Lava III

Mary Sheeran, Thomas Hallgren,
Emil Axelsson

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

Exercise: Zero detection

How to decompose the problem recursively?

Exercise: Zero detection

How to decompose the problem recursively?

First version

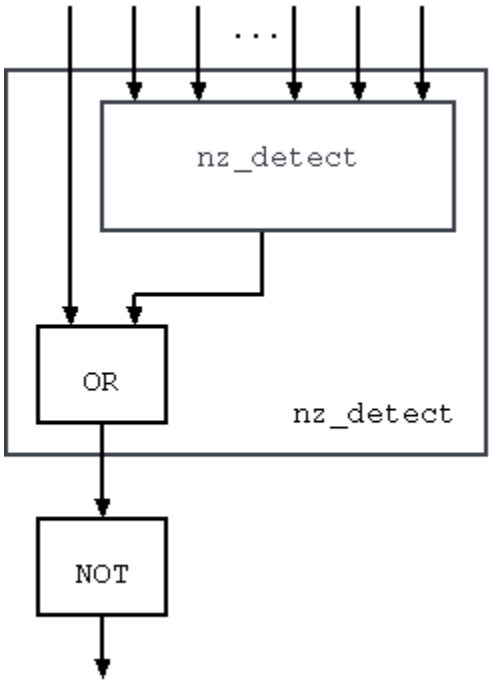
assume we have a circuit that works for n bits,
build a circuit that works for n+1 bits.
Result: a linear chain of 2-input gates

Second version

assume we have a circuit that works for n bits,
build a circuit that works for 2n bits.
Result: a balanced trees of 2-input gates

linear chain

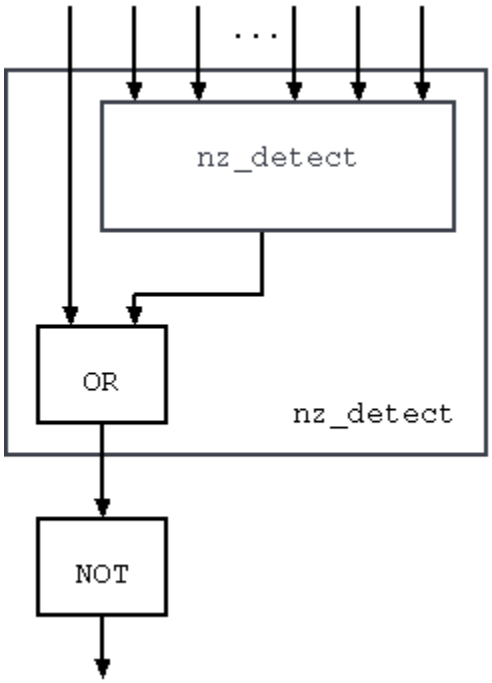
```
zero_detect as = inv nz
  where
    nz = nz_detect as

nz_detect [] = low
nz_detect (a:as) = out
  where
    out   = or2(a,out2)
    out2 = nz_detect as
```



linear chain

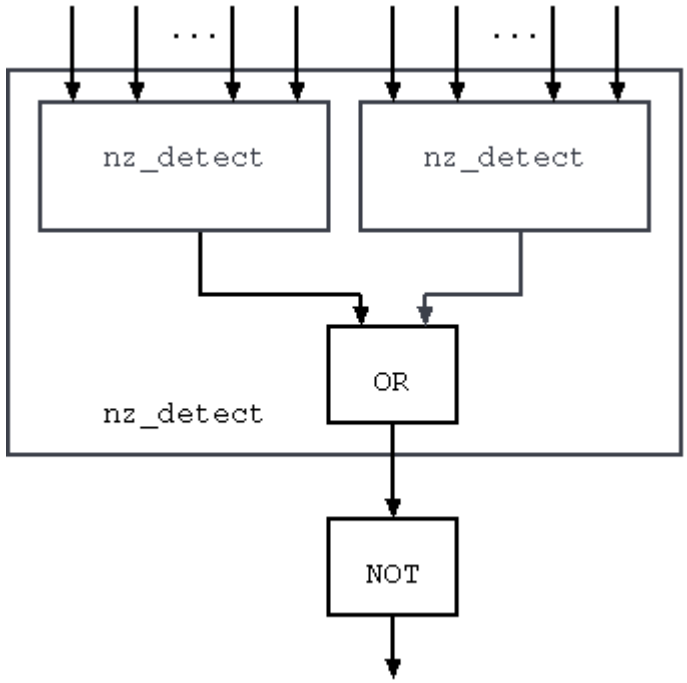
```
-- from Lecture 2
red :: ((a,b) -> a) -> (a, [b]) -> a
red f (a,[])        = a
red f (a, (b:bs)) = red f (f(a,b), bs)

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

```
zero_detect1 = lin or2 ->- inv
```

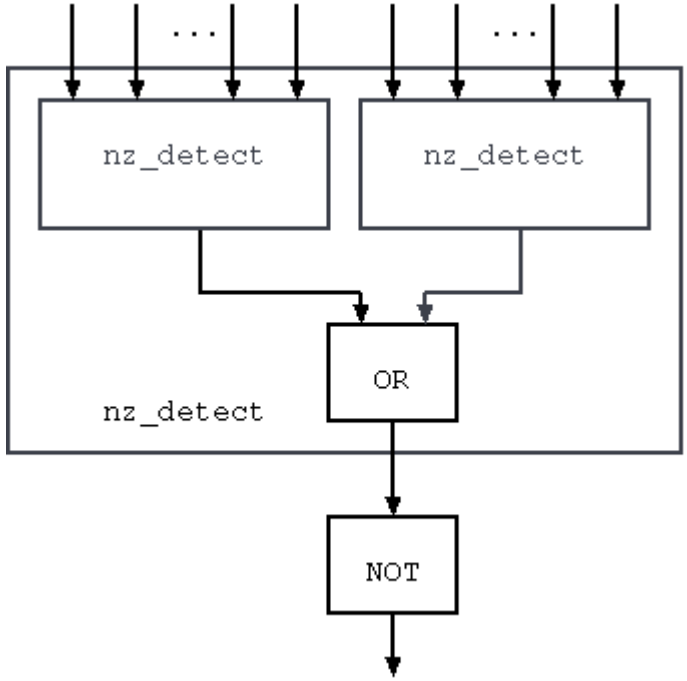


balanced tree

```
nz_detect1 []  = low
nz_detect1 [a] = a
nz_detect1 as  = out
  where
    (as1,as2) = halveList as
    out1 = nz_detect1 as1
    out2 = nz_detect1 as2
    out  = or2(out1,out2)
```

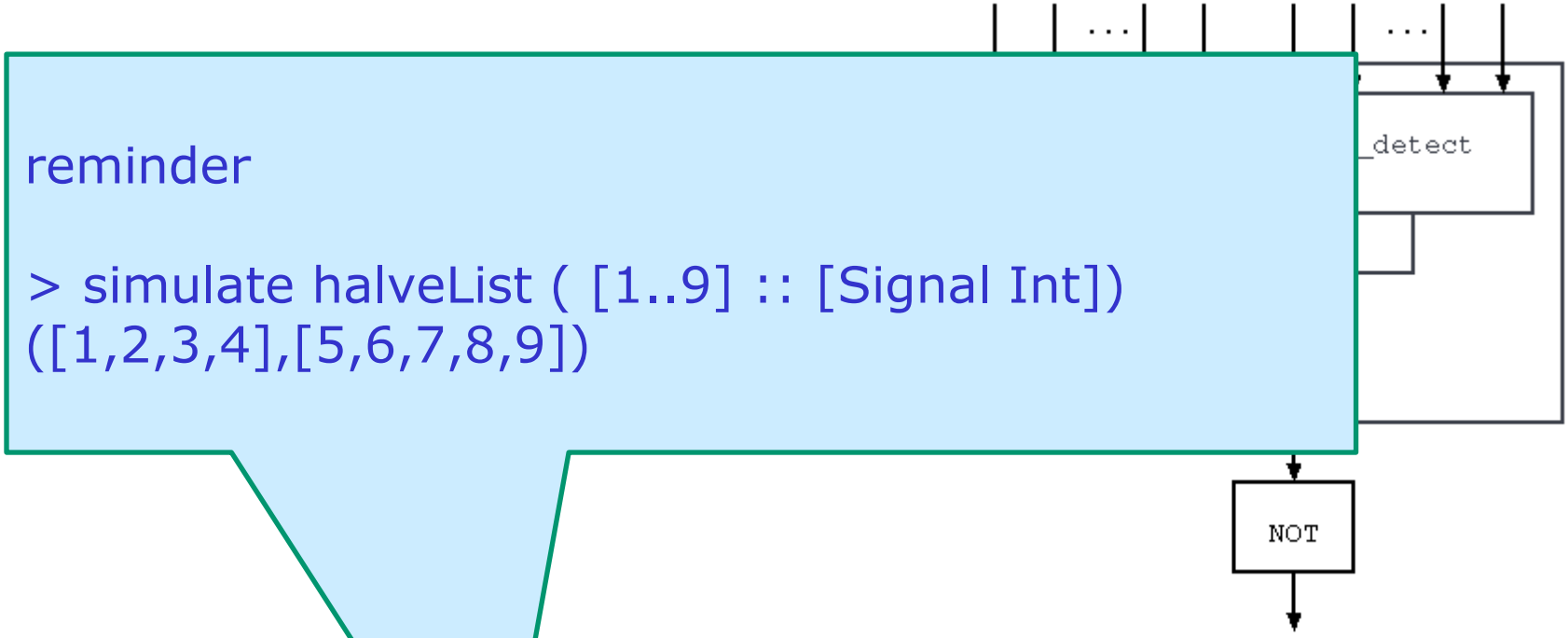


different style




```
nz_detect2 []    = low
nz_detect2 [a] = a
nz_detect2 as  = circ as
   where
        circ = halveList ->- (nz_detect2 -|- nz_detect2) ->- or2
```

different style



reminder

> simulate halveList ([1..9] :: [Signal Int])
([1,2,3,4],[5,6,7,8,9])

```
nz_detect2 []          w
nz_detect2 [a]
nz_detect2 as  =       rc as
   where
        circ = halveList ->- (nz_detect2 -|- nz_detect2) ->- or2
```

capturing the pattern for reuse

```
binTree c []   = error "binTree of empty list"
binTree c [a] = a
binTree c as  = circ as
    where
        circ = halveList ->- (binTree c -|- binTree c) ->- c
```

capturing the pattern for reuse

```
binTree c []   = error "binTree of empty list"
binTree c [a] = a
binTree c as  = ci
   where
      circ =
```

Q: Why do we need the second base case?

capturing the pattern for reuse

```
binTree c []   = error "binTree of empty list"
binTree c [a] = a
binTree c as  = ci
   where
      circ =
```

> simulate halveList [low]
([],[low])

Must make sure that inputs to recursive calls are smaller than original input

Comparing circuits

- The linear and tree-shaped versions have different logic depth
- Comparing behaviour with FV is easy (for fixed size boolean circuits, incl. Sequential)
- For comparing performance, we need to do some modelling of delay behaviour

Simple delay analysis: Depth computations

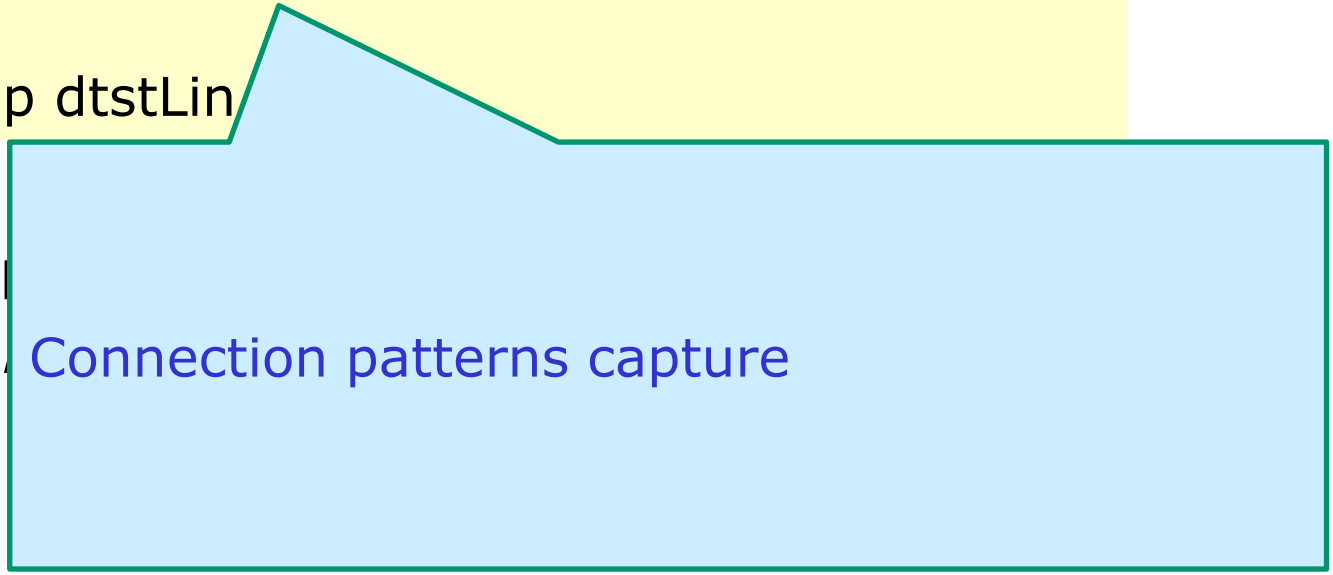
```
ldepth :: (Signal Int, Signal Int) -> Signal Int
ldepth (a,b) = max a b + 1

dtstTree n = simulate (binTree ldepth) (replicate n 0)

dtstT n = map dtstTree [1..n]
```

```
> dtstT 10
[0,1,2,2,3,3,3,3,4,4]
```

Simple delay analysis: Depth computations

```
dtstLin n = simulate (lin ldepth) (replicate n 0)

dtstL n = map dtstLin
```

```
*Main> dtstL
[0,1,2,3,4,5,
```



Simple delay analysis: Depth computations

```
lin     :: ((a, a) -> a) -> [a] -> a
binTree :: ((a, a) -> a) -> [a] -> a
```

Connection patterns do not constrain the ’a’ type

Enables reuse:

- Circuits of different type
  e.g. Singal Bool, (Singal Bool, Signal Bool)
- Non-functional analysis
  e.g. Singal Int or Integer

Simple delay analsysis: Modelling delay in a full adder

```
fAddI (a1s, a2s, a3s, a1c, a2c, a3c) (a1,(a2,a3)) = (s,cout)
  where
    s     =  maximum [a1s+a1, a2s+a2, a3s+a3]
    cout =   maximum [a1c+a1, a2c+a2, a3c+a3]

fI = fAddI (20,20,10,10,10,10)
```

Simple delay analsysis: Modelling delay in a full adder

```
-- from first lecture but generalising the type!
rcAdder2 :: ((a,(a,a)) -> (a,a)) -> (a,([a],[a])) -> ([a], a)
rcAdder2 fadd (c0, (as, bs)) = (sum, cOut)
  where
      (sum, cOut) = row fadd (c0, zipp (as,bs))


rcdeltst1 = simulate (rcAdder2 fI)
                  (0 :: Signal Int, (replicate 10 0, replicate 10 0))
```

```
> rcdeltst1
([20,30,40,50,60,70,80,90,100,110],100)
```

Simple delay analsysis: Modelling delay in a full adder

For feedback-free circuits, can also use Haskell directly:

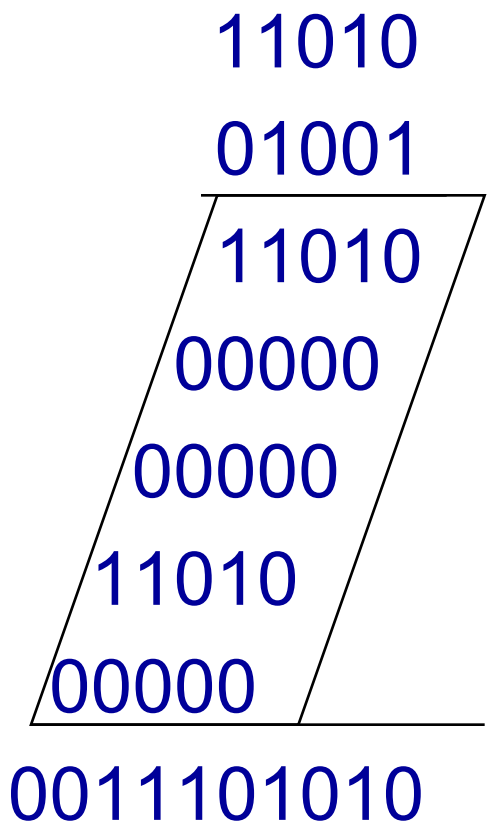
```
rcdeltst = rcAdder2 fI (0, (replicate 10 0, replicate 10 0))
```

Don’t try to mix the two approaches
Stay within Lava if you are not a Haskell expert!

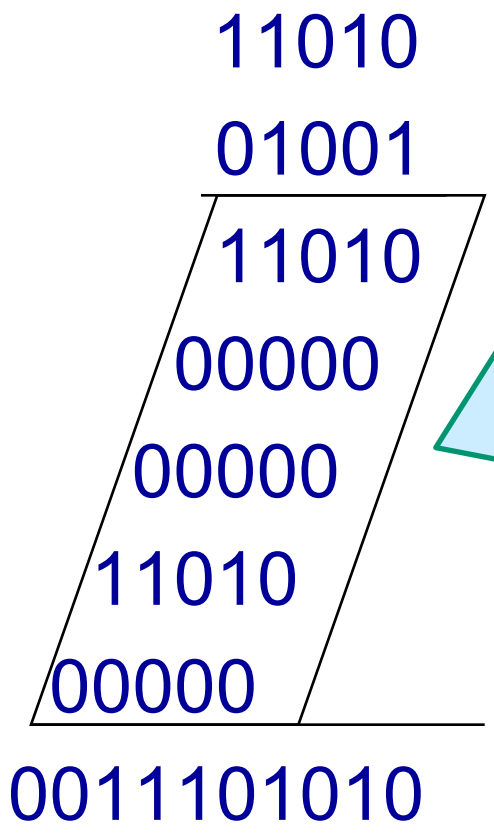
```
rcdeltst1 = simulate (rcAdder2 fI)
              (0 :: Signal Int, (replicate 10 0, replicate 10 0))
```

```
> rcdeltst1
([20,30,40,50,60,70,80,90,100,110],100)
```

Multiplication



Multiplication

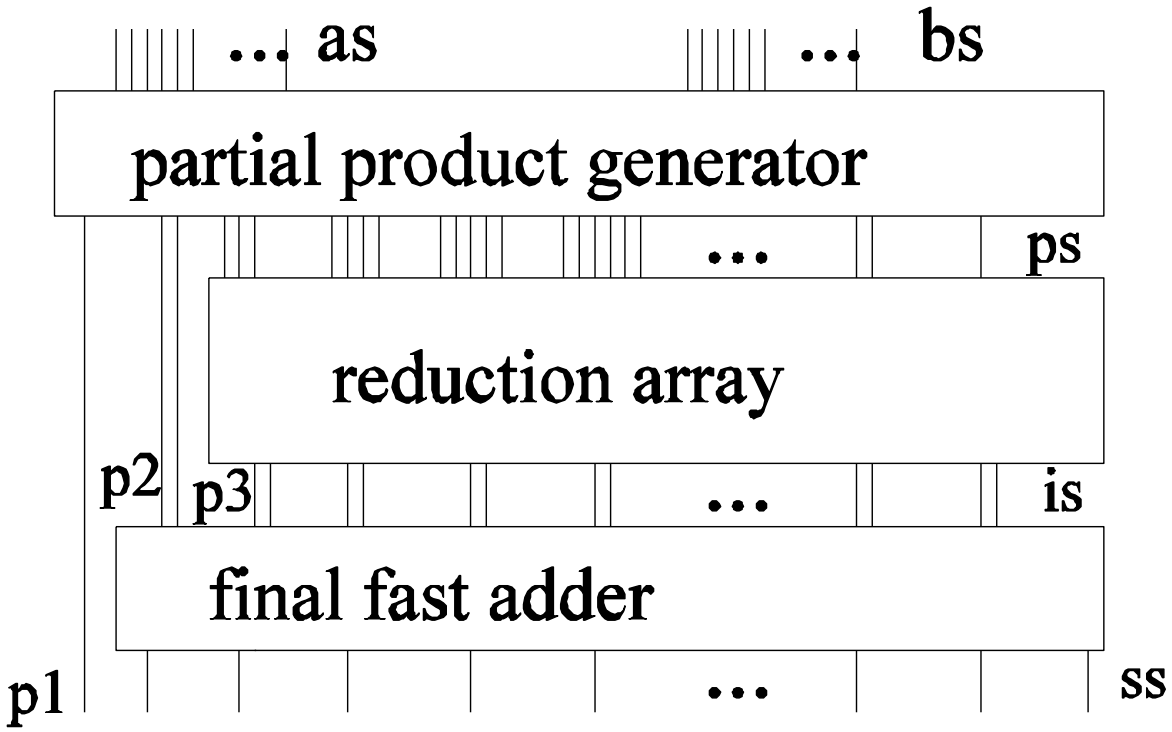


Partial products

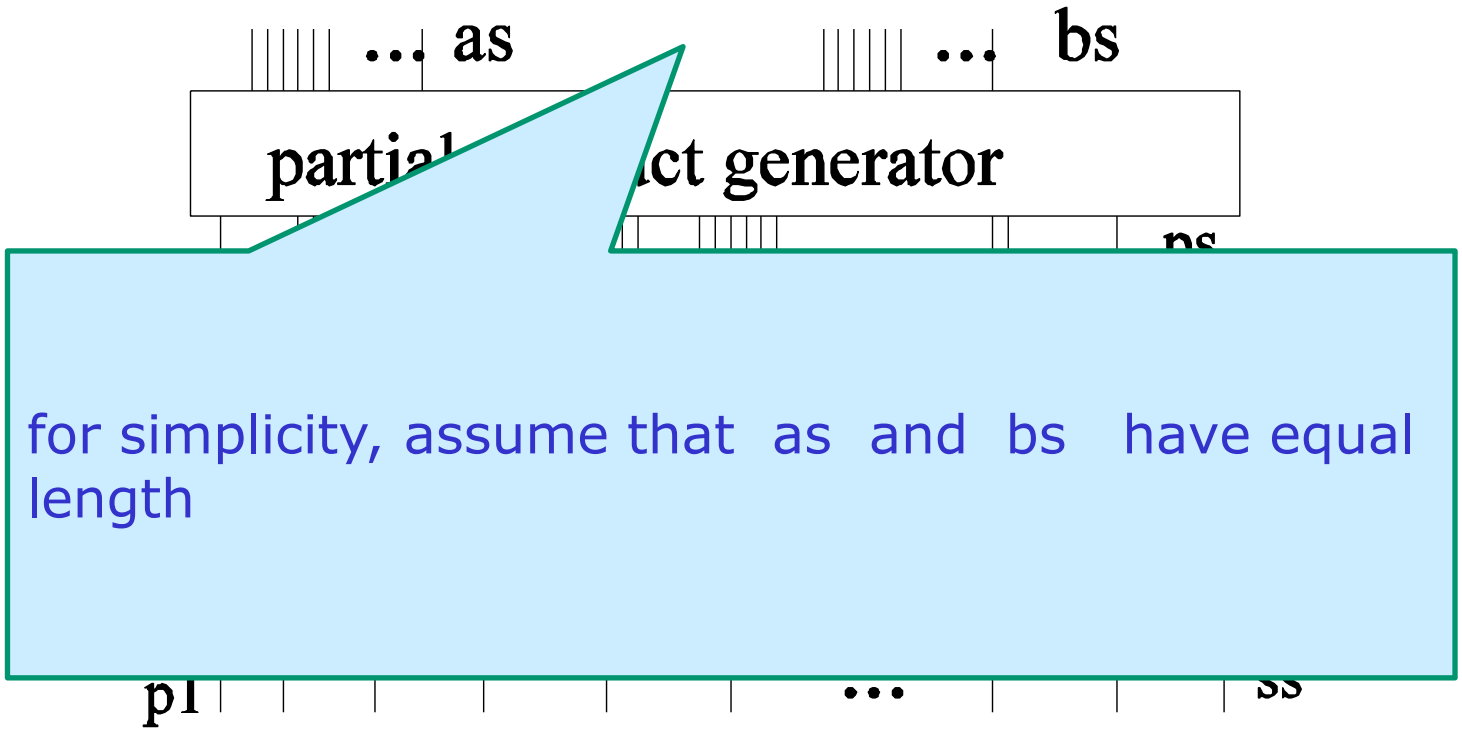
Making a multiplier is about adding up all these numbers (and that is what the Lava lab explores)

Here, we will look at a particular (slightly fancier) approach called column compression

Structure of multiplier



Structure of multiplier



Structure of multiplier

Reduction array reduces the partial products to two binary numbers (see Lava lab)

- Can be done in logic depth $O(log(n))$, where n is #partial products

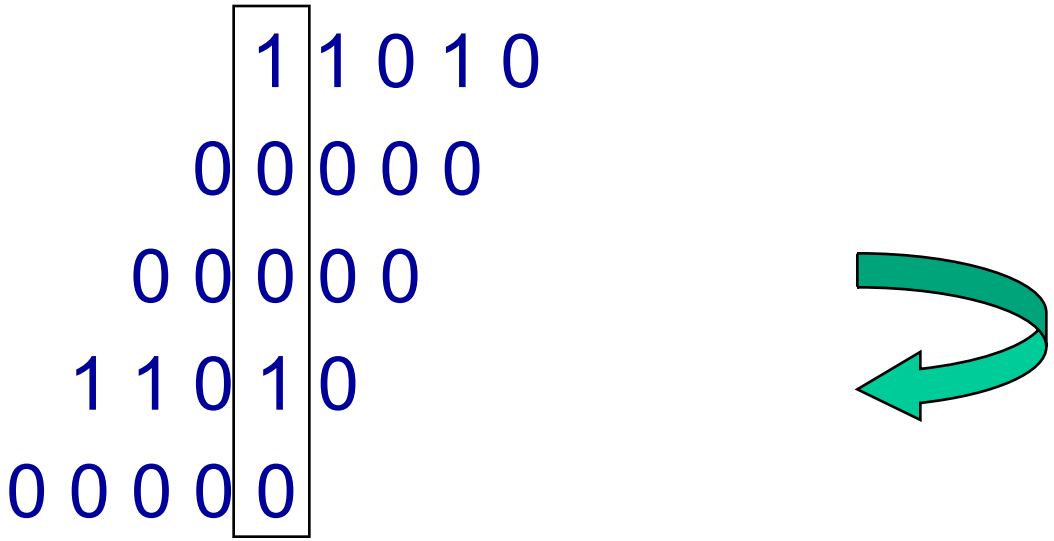
Final adder adds the two final numbers

- Can be done in logic depth $O(log(n))$, where n is the width of the numbers (~ #partial products)

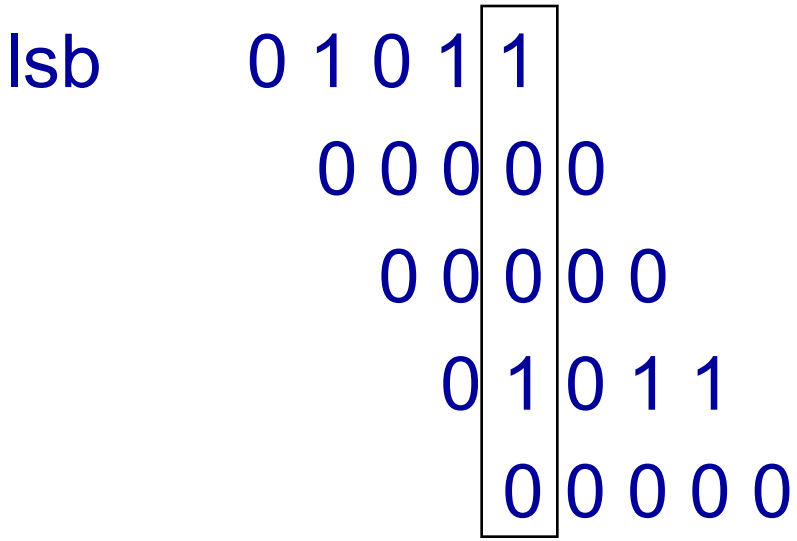
Total depth of $n \times n$ bit mult: $O(log(n))$

Multiplication

msb

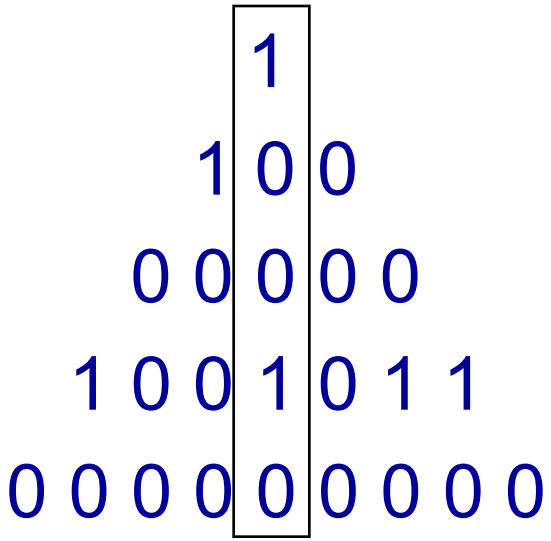


Multiplication



Multiplication

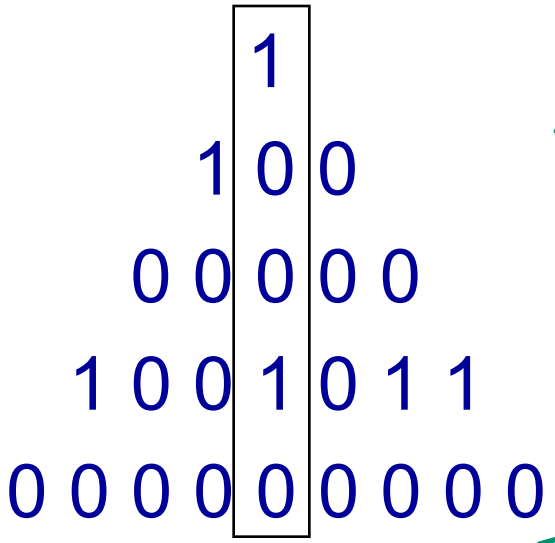
lsb



Process column by column instead of row by row

Multiplication

lsb



Process column by column instead of row by row

The bits in each column (and carries from the column before) can be processed in any order. Different decisions lead to different performance.

Generic components

See Lava3_mult.hs

```
type Components s =
    ( s                     -- low
    , s                     -- high
    , (s,s) -> s            -- and2
    , (s,s) -> (s,s)        -- halfAdd
    , (s,(s,s)) -> (s,s)    -- fullAdd
    )
```

Generic components

```
-- Components for logic simulation
logicComps :: Components (Signal Bool)
logicComps = (low,high,and2,halfAdd,fullAdd)

-- Components for delay estimation
delayComps :: Components (Signal Int)
delayComps = (0,0,an,ha,fa)
  where
    an (a,b)     = 1 + max a b
    ha (a,b)     = let x = 2 + max a b in (x,x)
    fa (c,(a,b)) = let x = 3 + maximum [a,b,c] in (x,x)
```

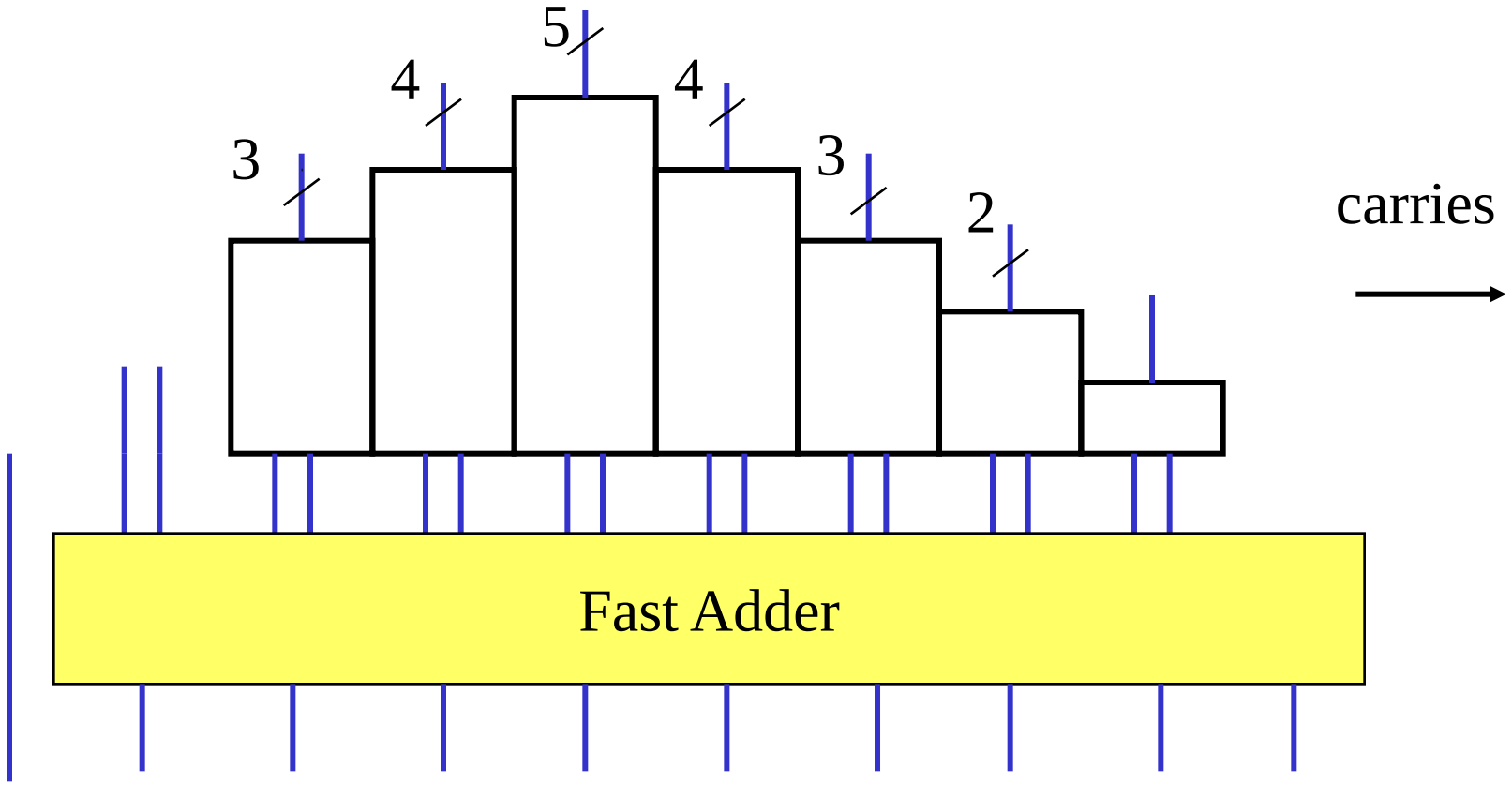
Partial products, grouped by weight

```
prods_by_weight (l,h,an,ha,fa) (as,bs)
   = [[an (a,b) | (a,m) <- number as,
                  (b,n) <- number bs,
                  m+n == w] | w <- [0 .. 2*length as-2]]
 where
   number cs = zip cs [0 .. length cs-1]
```

Complete multiplier

```
multBin :: Components s -> ([s],[s]) -> [s]
multBin comps (as,bs) = p1:ss
  where
    ([p1]:[p2,p3]:ps) = prods_by_weight comps (as,bs)
    is                = [(i1,i2) | [i1,i2] <- redArray comps ps]
    ss                = binaryAdder comps ((p2,p3):is)

redArray :: Components s -> [[s]] -> [[s]]
redArray comps ps = is
  where
    (is,[]) = row (compress comps) ([],ps)
```

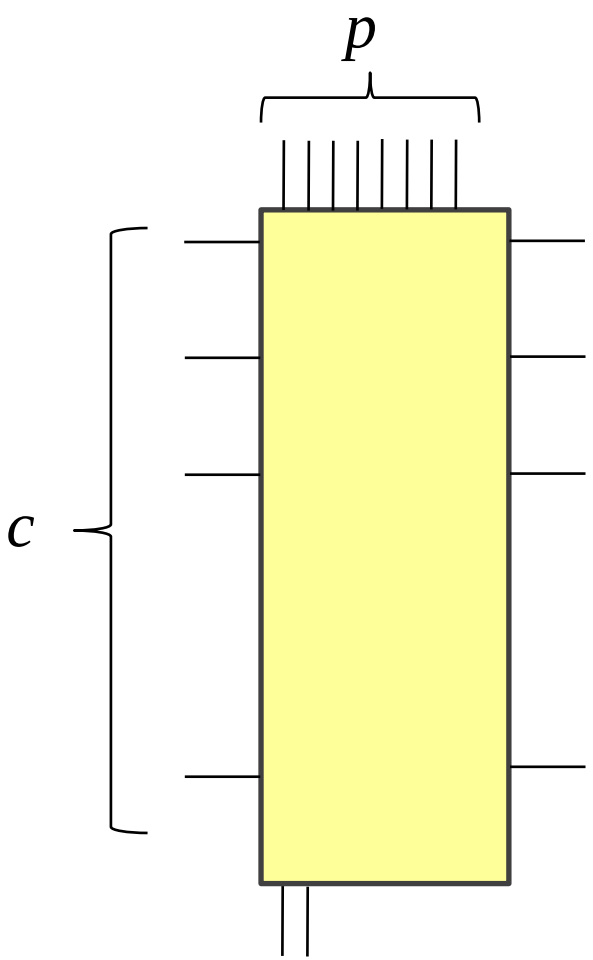
Reduction tree for multiplier



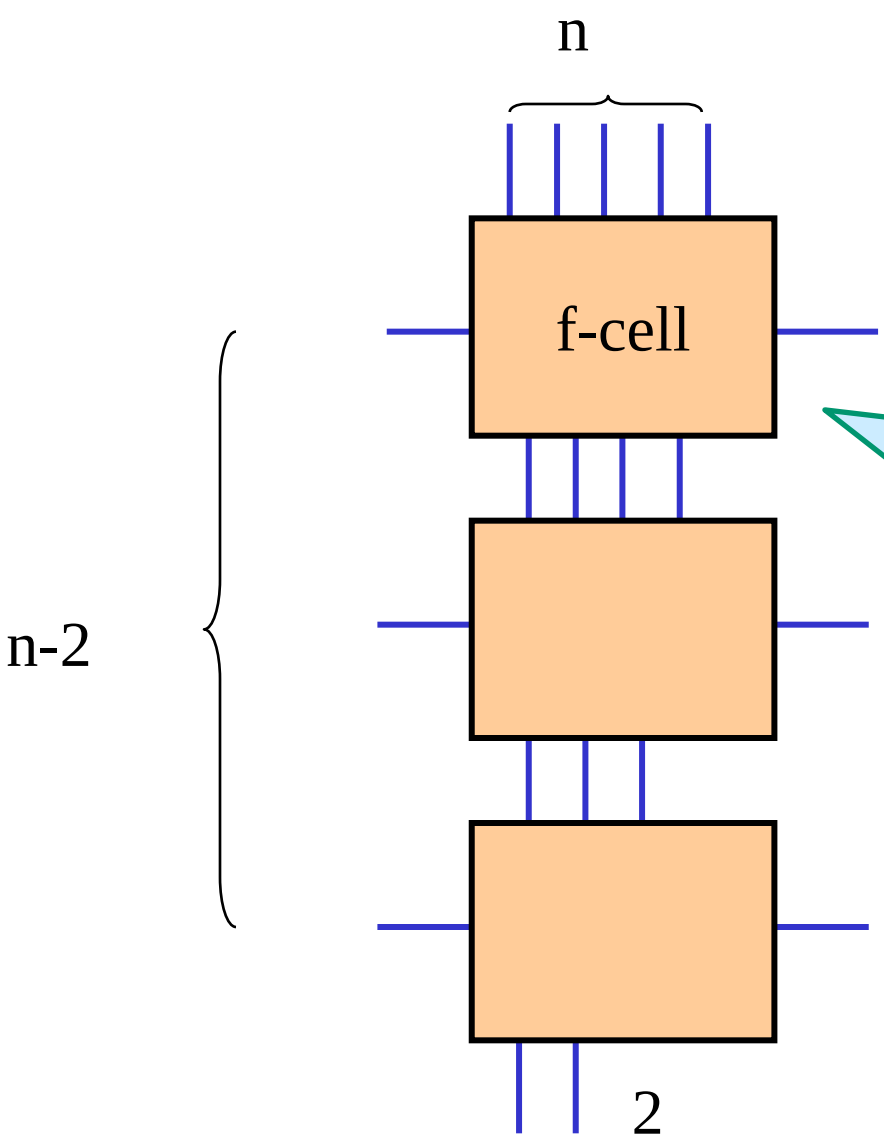
Missing piece: compress

Different cases depending on

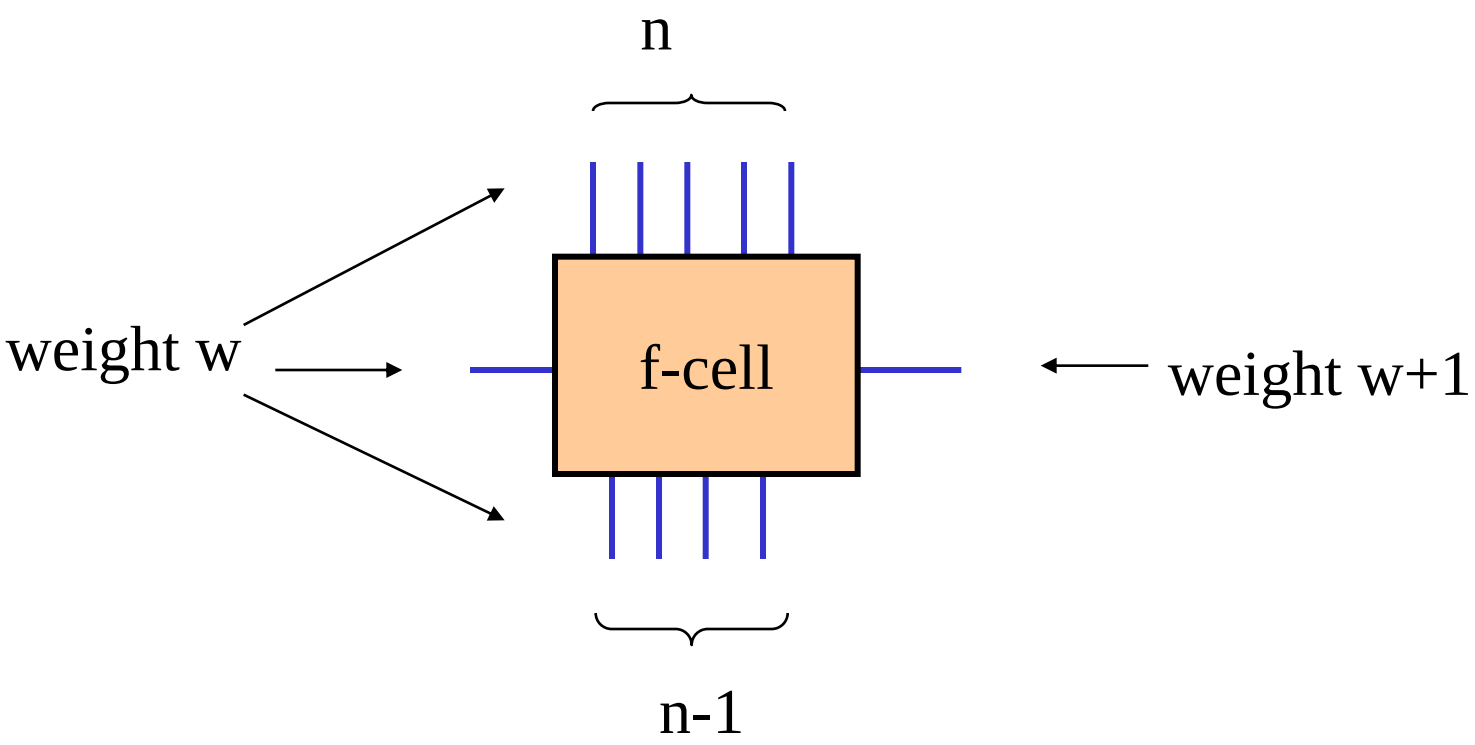
diff = *p*-*c*



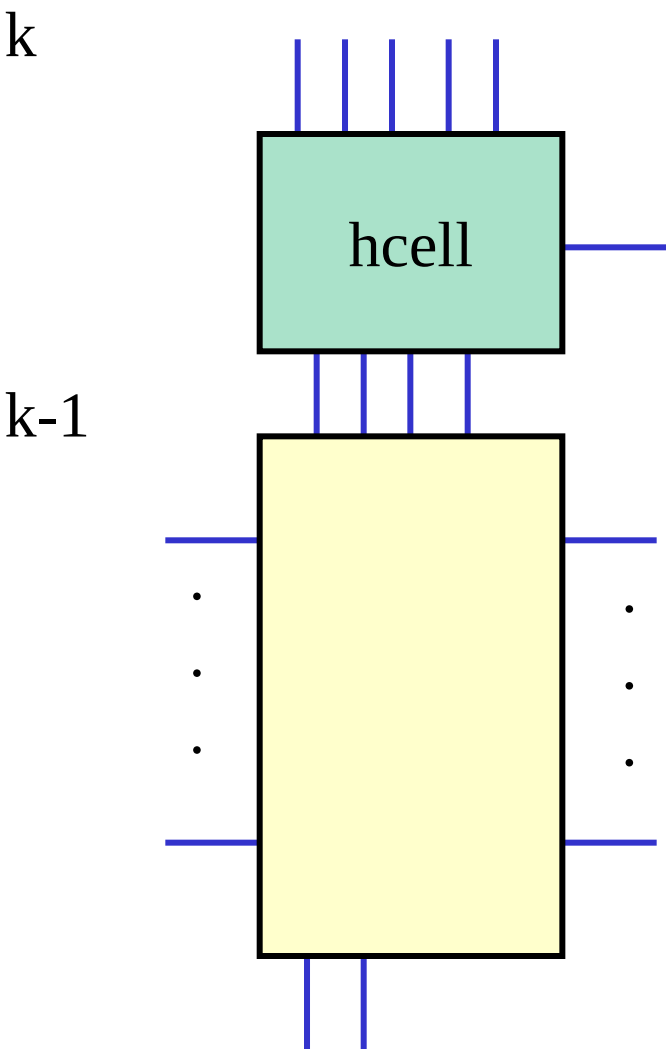
Compress (diff=2)



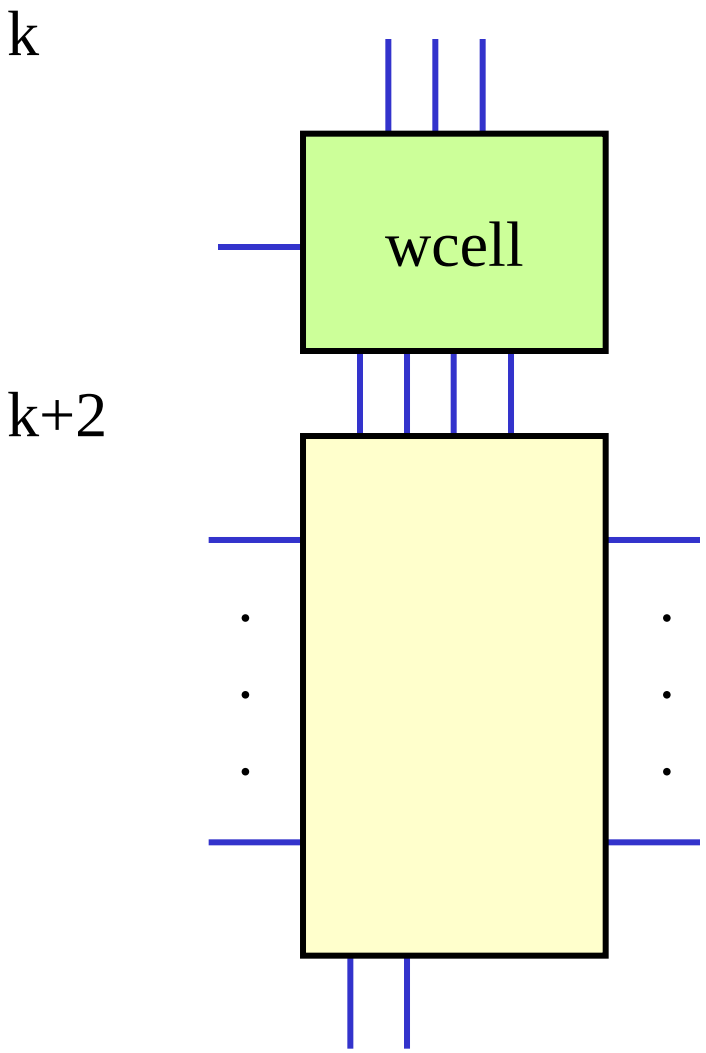
Use one full adder to compress three bits to one sum bit and a carry

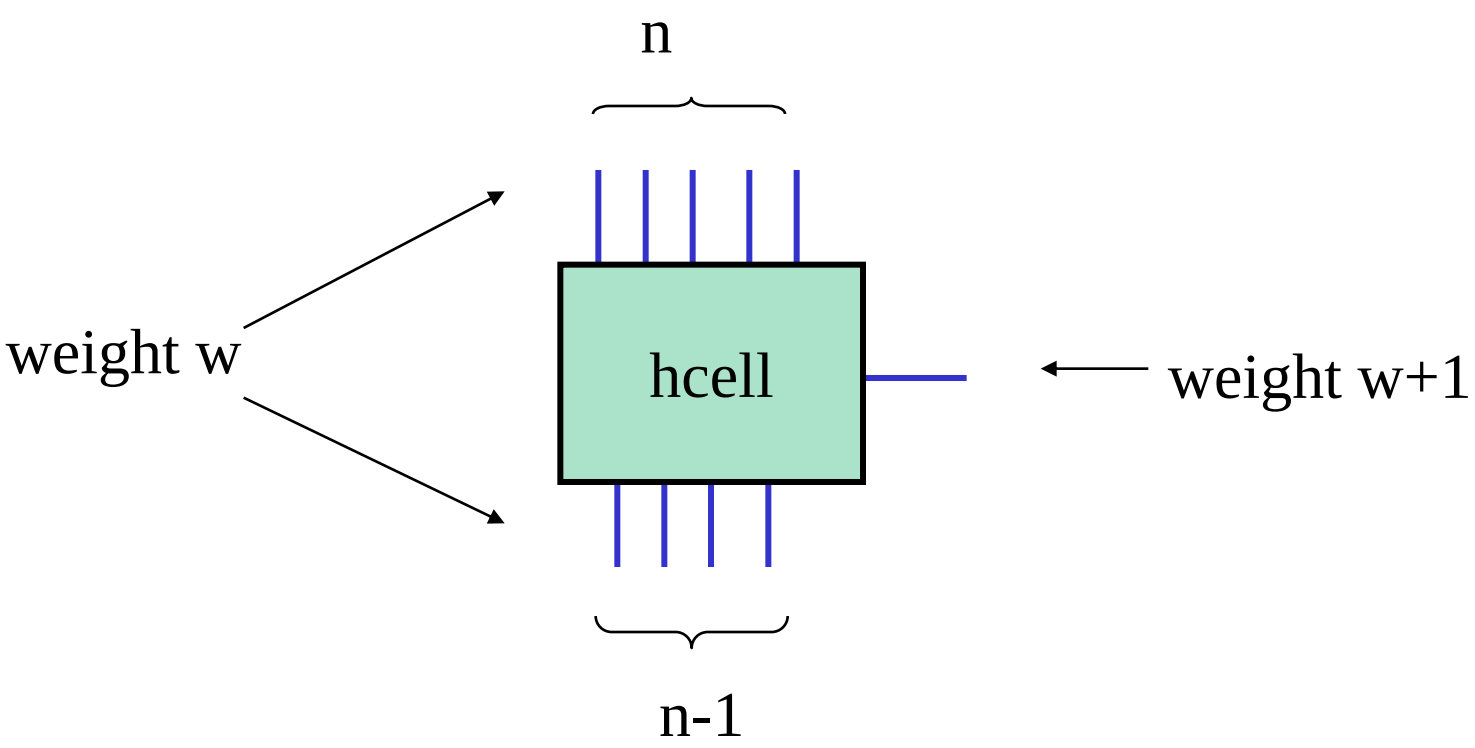

n
weight w
f-cell
weight w+1
n-1

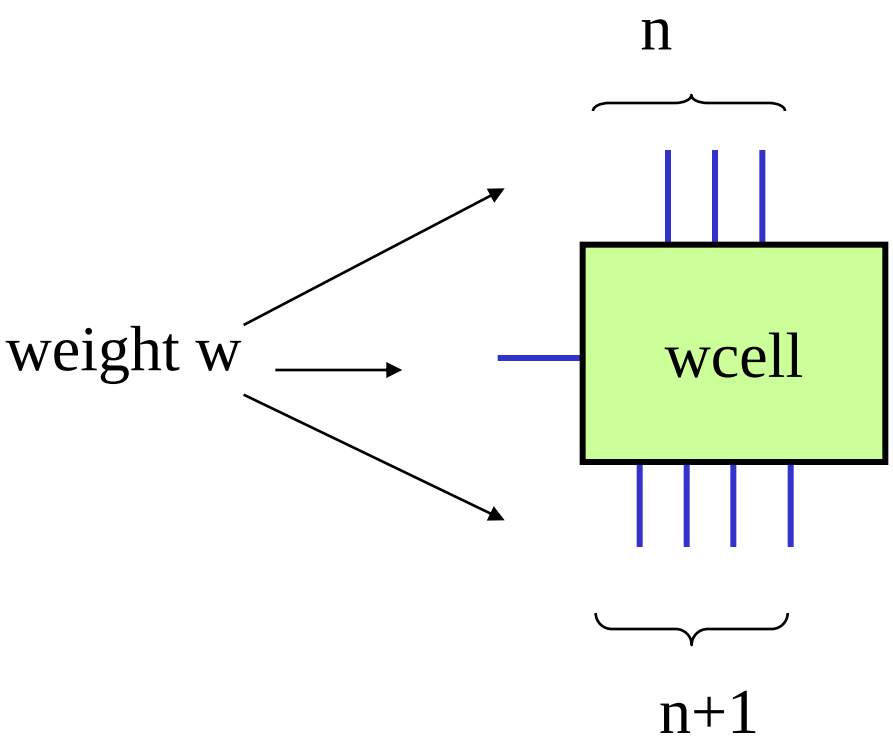
diff > 2



diff < 2



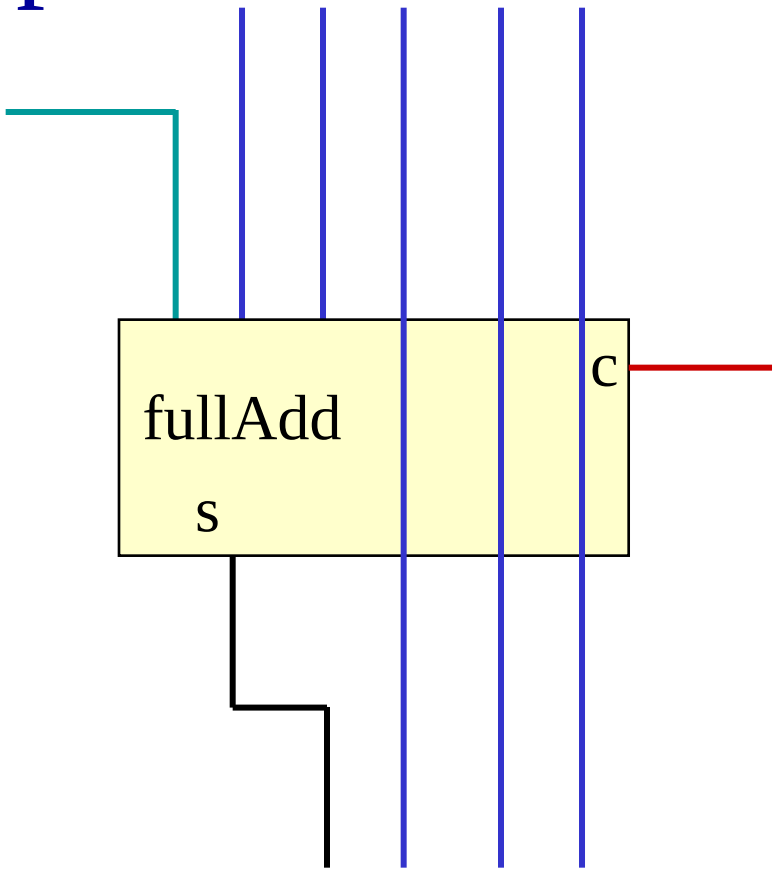

n
weight w
hcell
weight w+1
n-1


n
weight w
wcell
n+1

```
compress comps (as,bs)
    | diff >  2  = (compress comps |- hcell comps) (as,bs)
    | diff == 2 = column (fcell comps)                (as,bs)
    | diff <  2  = (compress comps -| wcell)          (as,bs)
  where
    diff = length bs - length as
```

(-| and |- are defined in Lava3_mult.hs.)

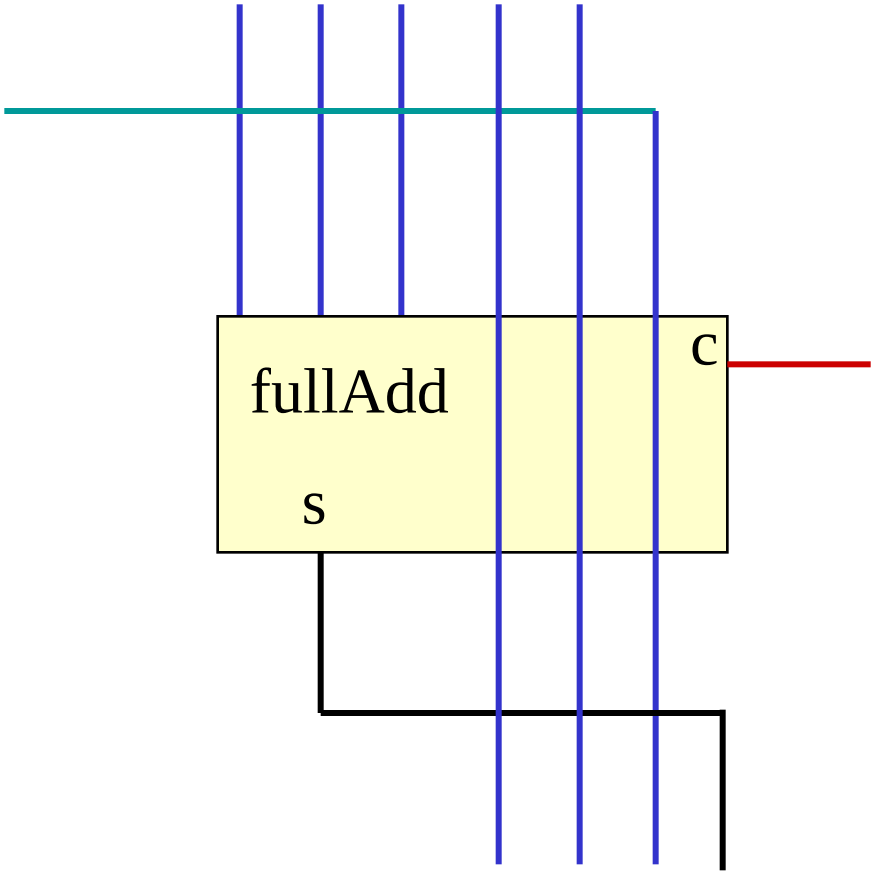
possible fcell



hcell similar using halfAdd

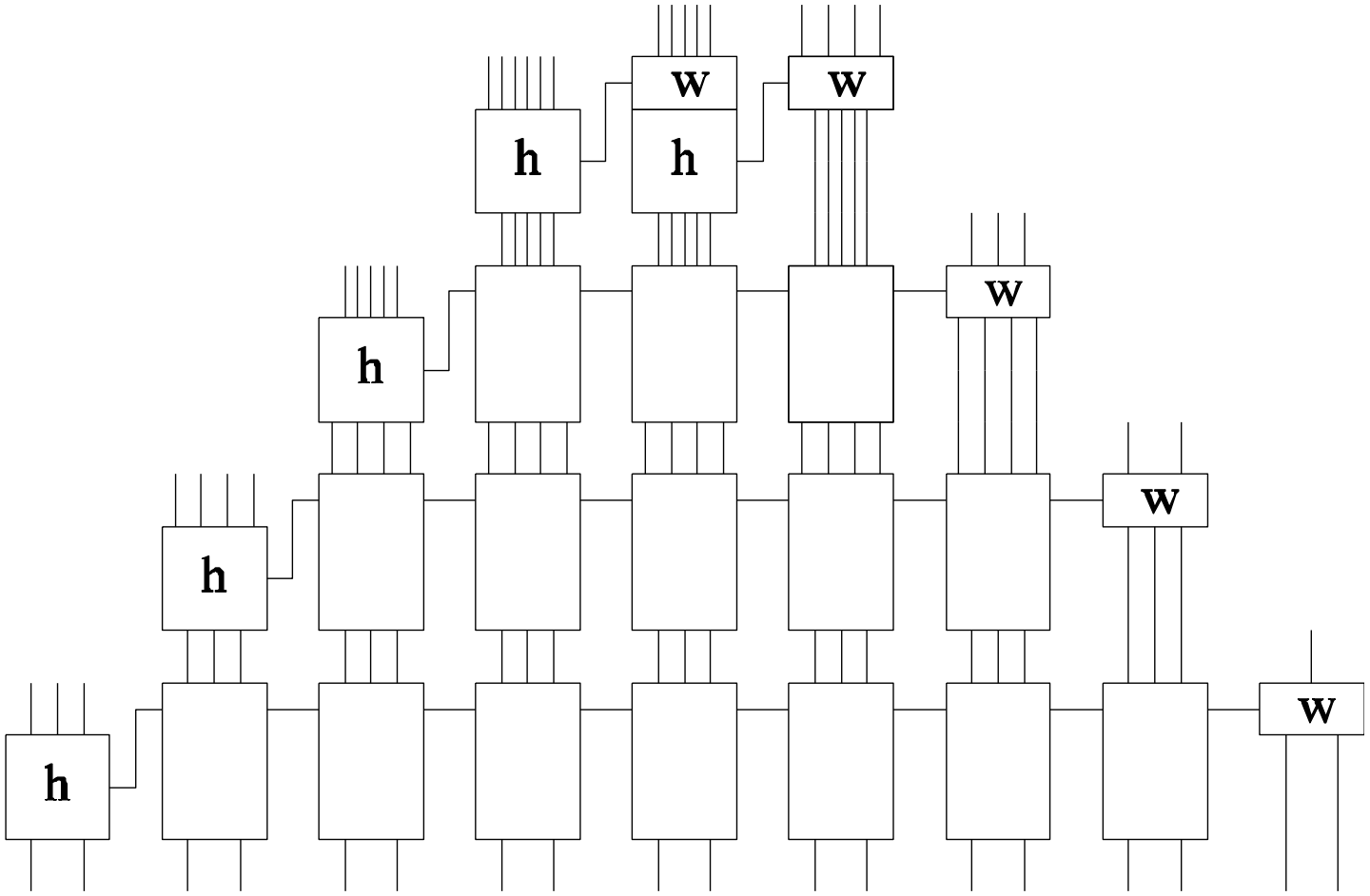
Gives standard array multiplier. Not great!

Dadda-like

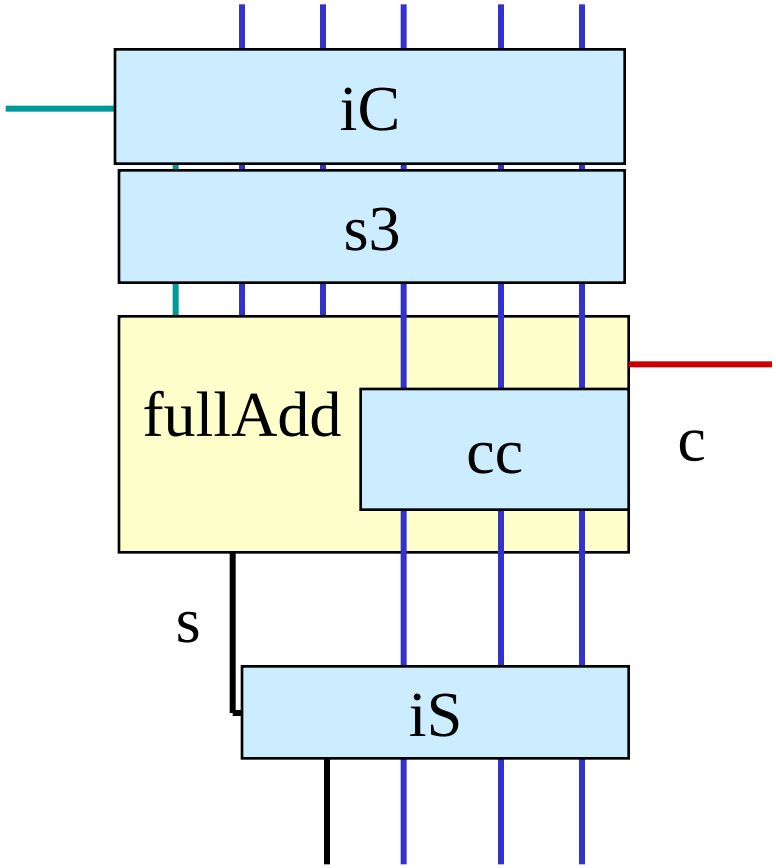


Excellent log depth reduction tree, but known for irregularity, difficult layout

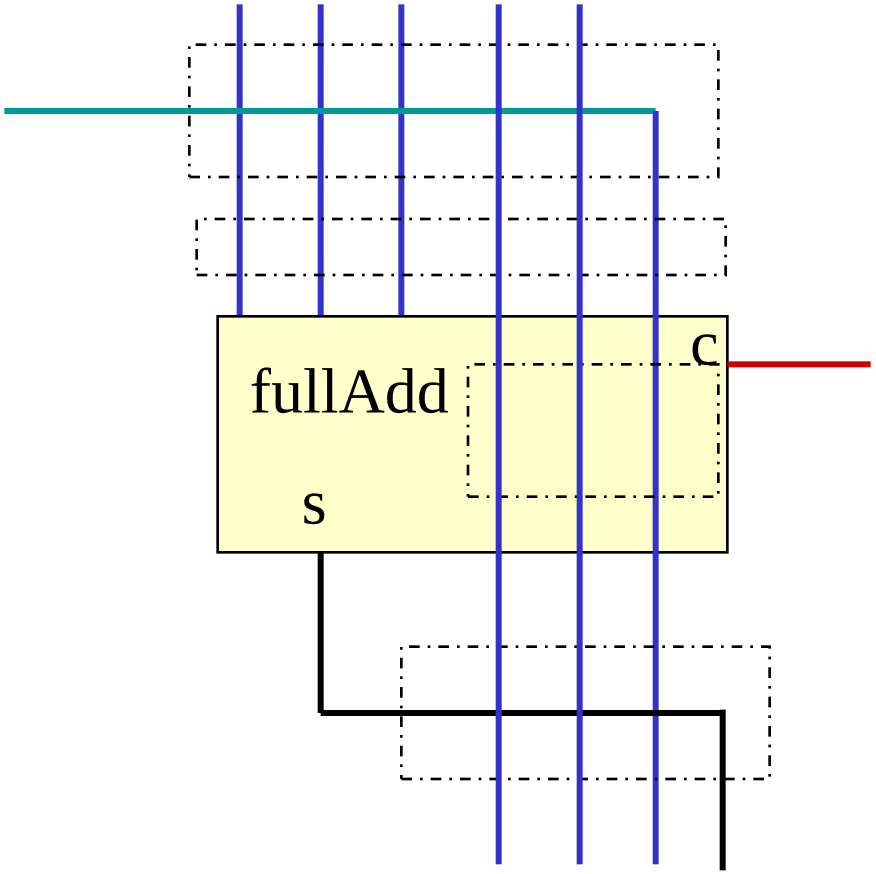
```
-- insert a bs = a:bs      -- Stick to front: array mult
insert a bs = bs ++ [a]  -- Stick to end: Dadda

hcell :: Components s -> [s] -> ([s],s)
hcell (l,h,an,ha,fa) (b1:b2:bs) = (insert s bs, c)
  where
    (s,c) = ha (b1,b2)

fcell :: Components s -> (s,[s]) -> ([s],s)
fcell (l,h,an,ha,fa) (ci,bs) = (insert s xs, c)
  where
    x1:x2:x3:xs = insert ci bs
    (s,c)           = fa (x1,(x2,x3))

wcell :: (s,[s]) -> [s]
wcell (a,bs) = insert a bs
```

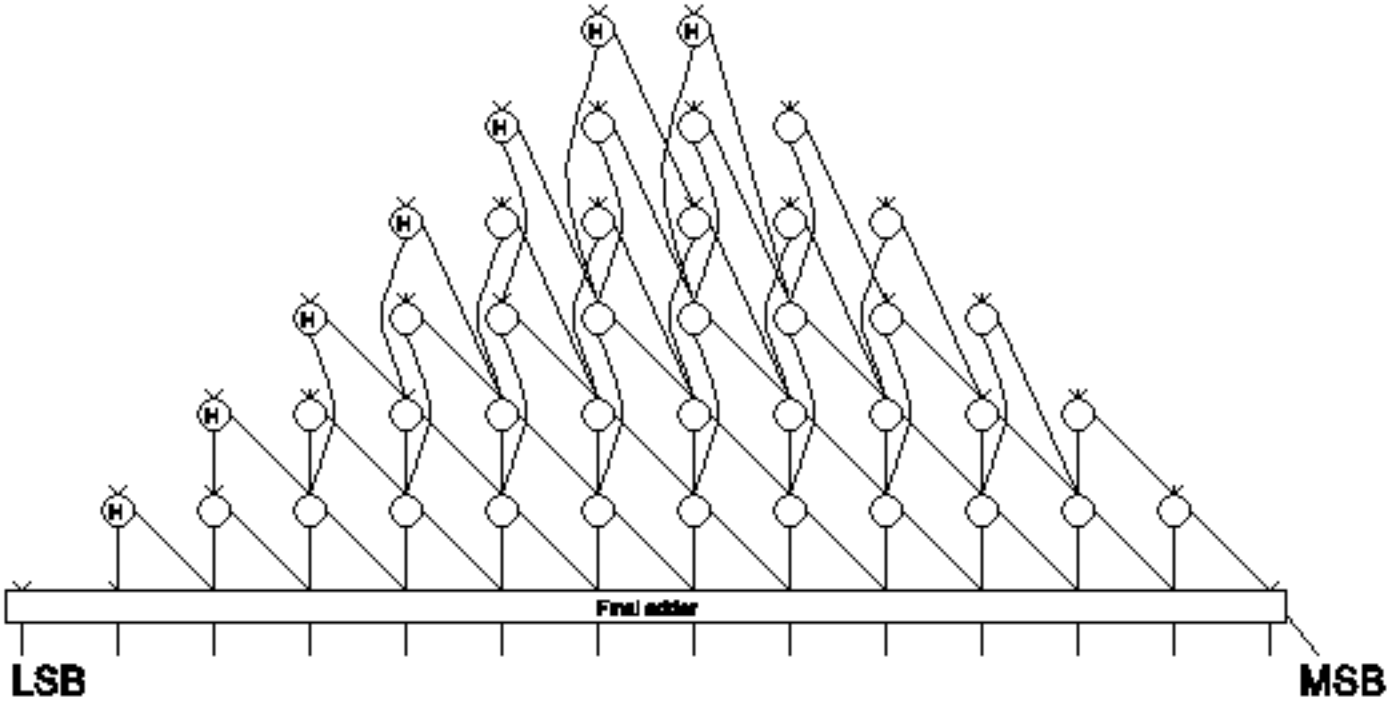

w
w
h
h
w
h
h
w
w
h
w

Exploring the choices:
Only need to vary wiring!



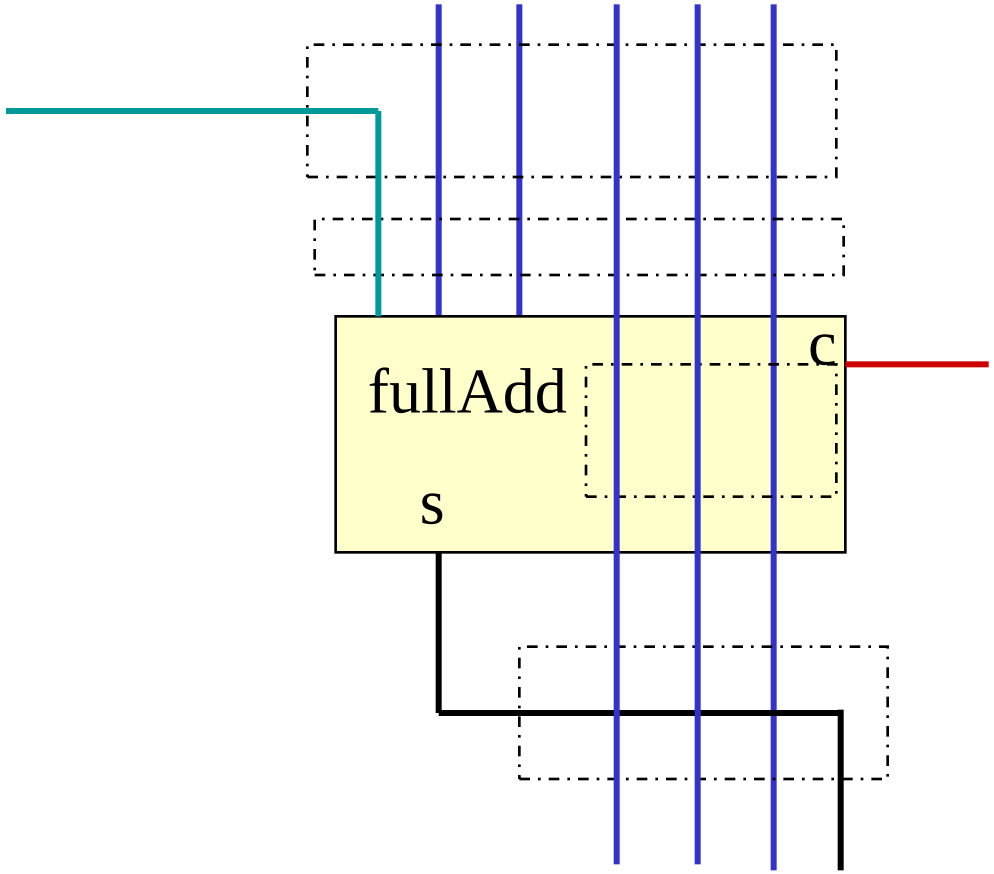
Dadda-like



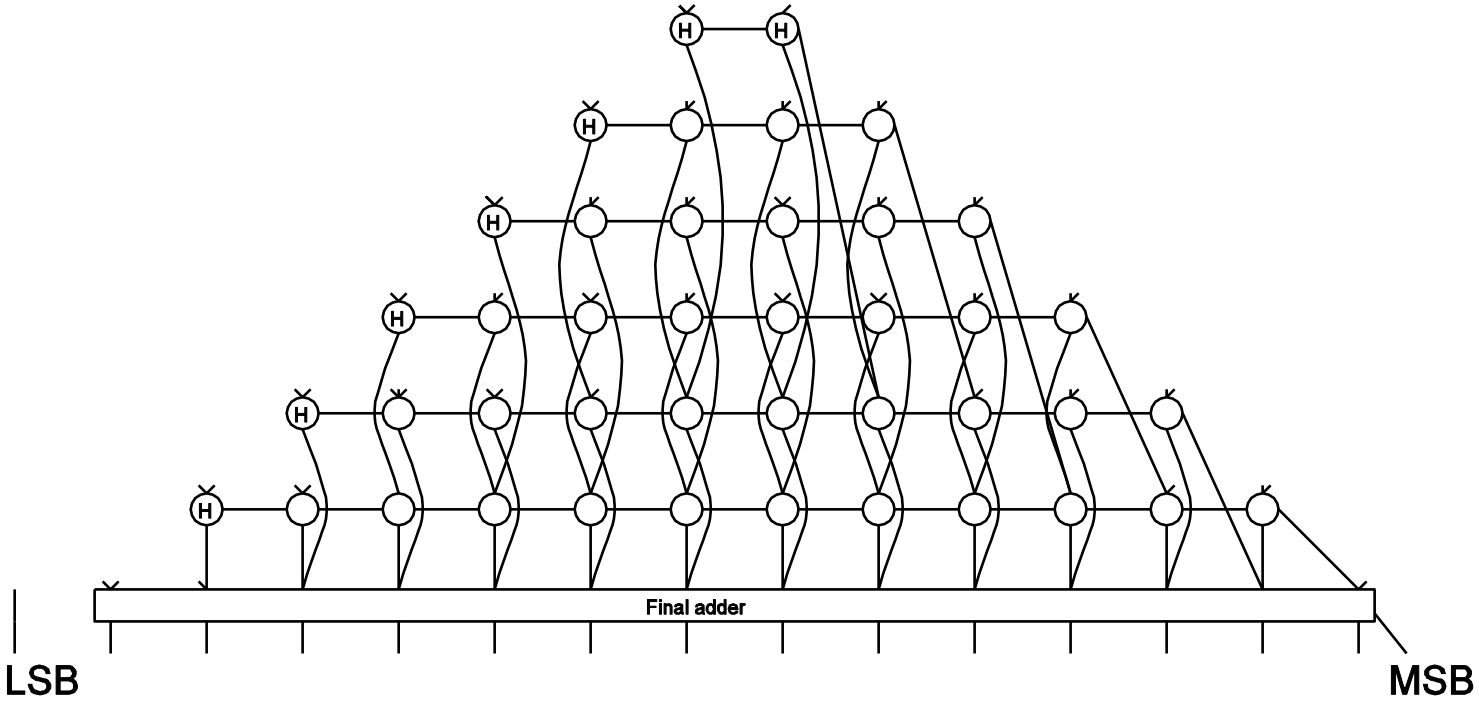


picture by Henrik Eriksson, Chalmers

Regular reduction tree (Eriksson et al. CE)



Nowhere near as good as Dadda, but inspired this work



picture by Henrik Eriksson, CE

The cool thing

The same description with just some different wiring cells gives a GREAT VARIETY of different multipliers

Layout of Dadda turned out to be easy!

One begins to see some order in the chaos...

The key point was finding the right connection pattern

Ideally, one would like to prove this extremely generic description correct! Open research question....

Verification

```
prop_mult n =
    forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      multBin logicComps (as,bs) <==> multi (as,bs)
```

smv (prop_mult 8) goes through in less than half a second. But size 16 doesn't. Why?

Delay analysis

```
delays :: Int -> Signal Int -> [Signal Int]
delays n inp = multBin delayComps (replicate n inp, replicate n inp)

totDelay :: Int -> Signal Int -> Signal Int
totDelay n inp = maximum (delays n inp)
```

Delay analysis

Lava3_mult.hs uses a linear final adder:

```
binaryAdder (l,h,an,ha,fa) abs = ss ++ [c]
  where
    (ss,c) = row fa (l,abs)
```

Replace with and gates to avoid linear depth in delay analysis:

```
binaryAdder (l,h,an,ha,fa) abs = map an abs
```

(Ideally should use a fast logarithmic adder.)

Delay analysis: Dadda

```
insert a bs = bs ++ [a]  -- Stick to end: Dadda

*Main> simulate (delays 8) 0
[1,2,4,7,8,10,11,13,14,14,14,14,14,14,8]

*Main> simulate (delays 16) 0
[1,2,4,7,8,10,11,13,14,14,16,17,17,17,17,19,20,20,20,20,20,20,20,
    20,17,17,17,17,14,14,8]
```

Delay analysis: Dadda

```
*Main> simulate (totDelay 4) 0
7
*Main> simulate (totDelay 8) 0
14
*Main> simulate (totDelay 16) 0
20
*Main> simulate (totDelay 32) 0
26
*Main> simulate (totDelay 64) 0
31
*Main> simulate (totDelay 128) 0
35
```

Delay analysis: array mult

```
insert a bs = a:bs  -- Stick to front: array mult


*Main> simulate (totDelay 4) 0
13
*Main> simulate (totDelay 8) 0
37
*Main> simulate (totDelay 16) 0
85
*Main> simulate (totDelay 32) 0
181
*Main> simulate (totDelay 64) 0
373
*Main> simulate (totDelay 128) 0
757
```

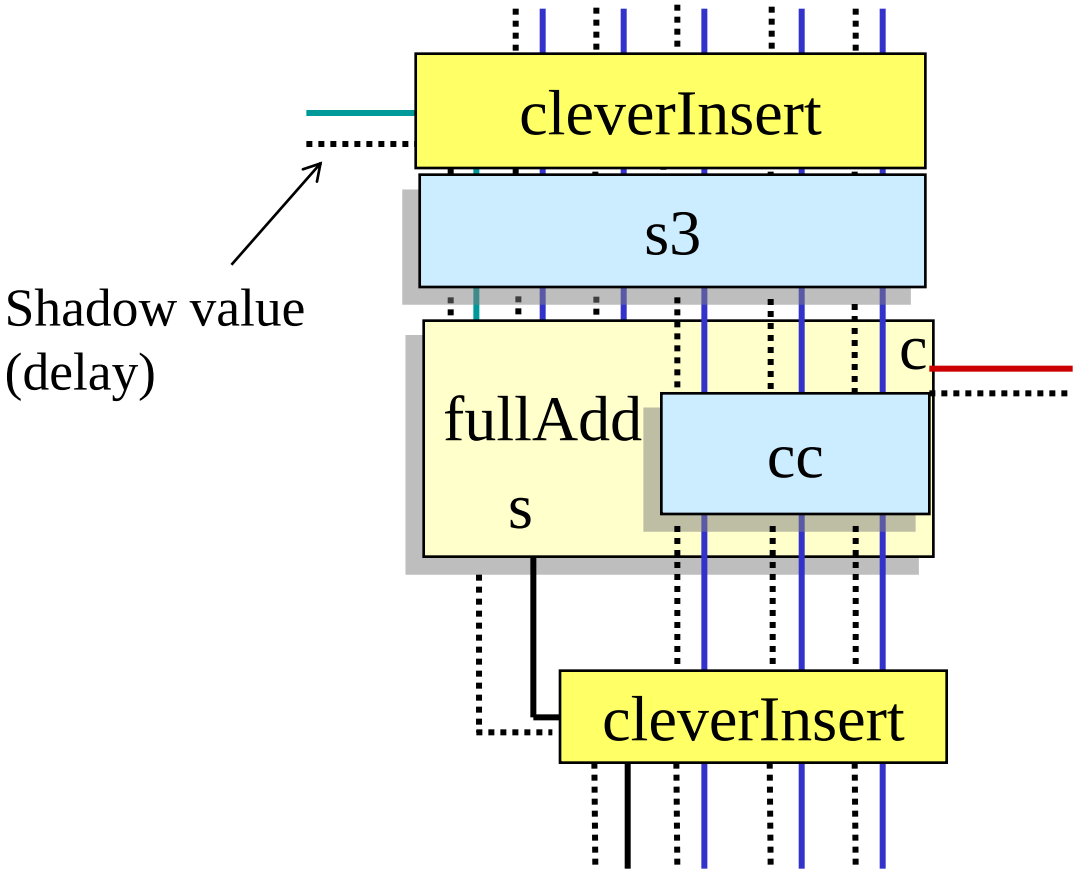
Next step: Choose wiring automatically

Predefined wiring components not optimal

Idea by Mary Sheeran:

- Pair up each wire with its estimated delay
  - ”Shadow value”
- Let the wiring components make sure that fast signals get processed first
- Resulting multiplier adapts both to context and internal delay

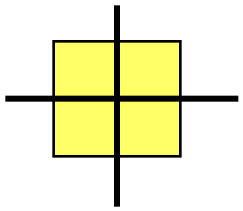
Idea: Harden the wiring during circuit generation using clever circuits. Shadow values estimate delay through wires and cells.

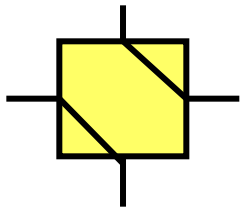


Parameterizing on s3 and cc lets us take wire delay into account

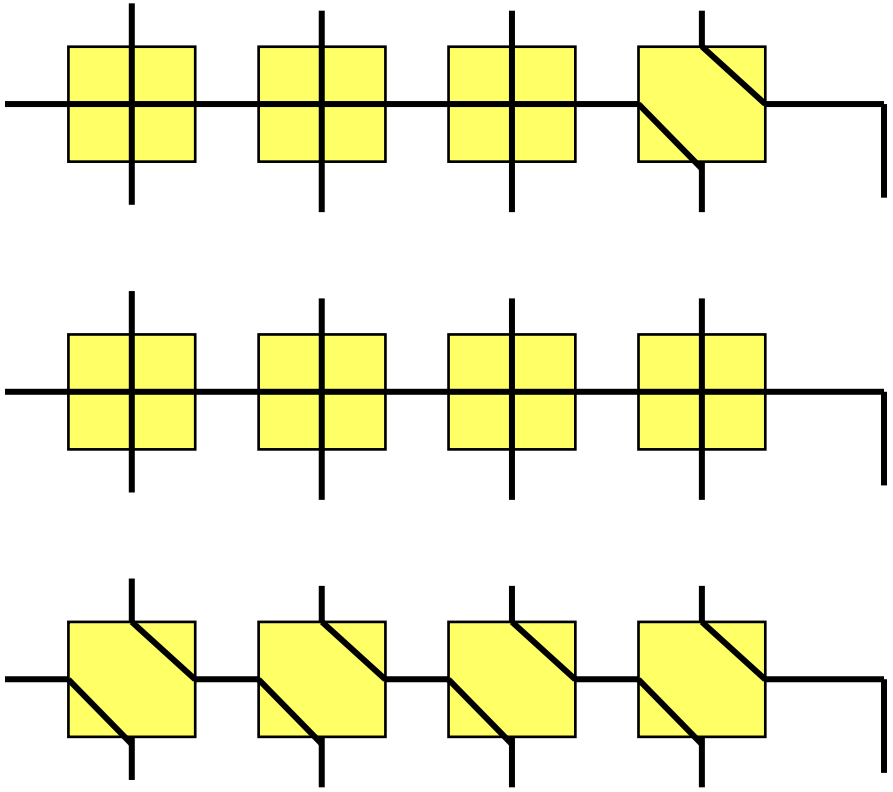
cswap((a,x),(b,y))

= if (x>y) then ((b,y),(a,x))else((a,x),(b,y))





cleverInsert = row cswap ->- apr



forms necessary wiring based on context (delays on shadow wires)

Result (multiplication)

Simple parameterised description of fast adaptive multiplier

Better than Dadda and TDM

Adaption to incoming delay profile can be arranged (clever circuits again)

Can also easily adapt description to take account of limitations on cross-cell tracks (see FMCAD04 paper)

Much remains to be done (e.g. insertion of buffers, fine delay modelling, transistor sizing, other layouts, the rest of the multiplier...). The approach feels right!

Reading

Published paper about this is at

http://www.cse.chalmers.se/~ms/fmcadMultSubmit.pdf

NOT required reading. Read if interested.

Next step: Wired (see links page)

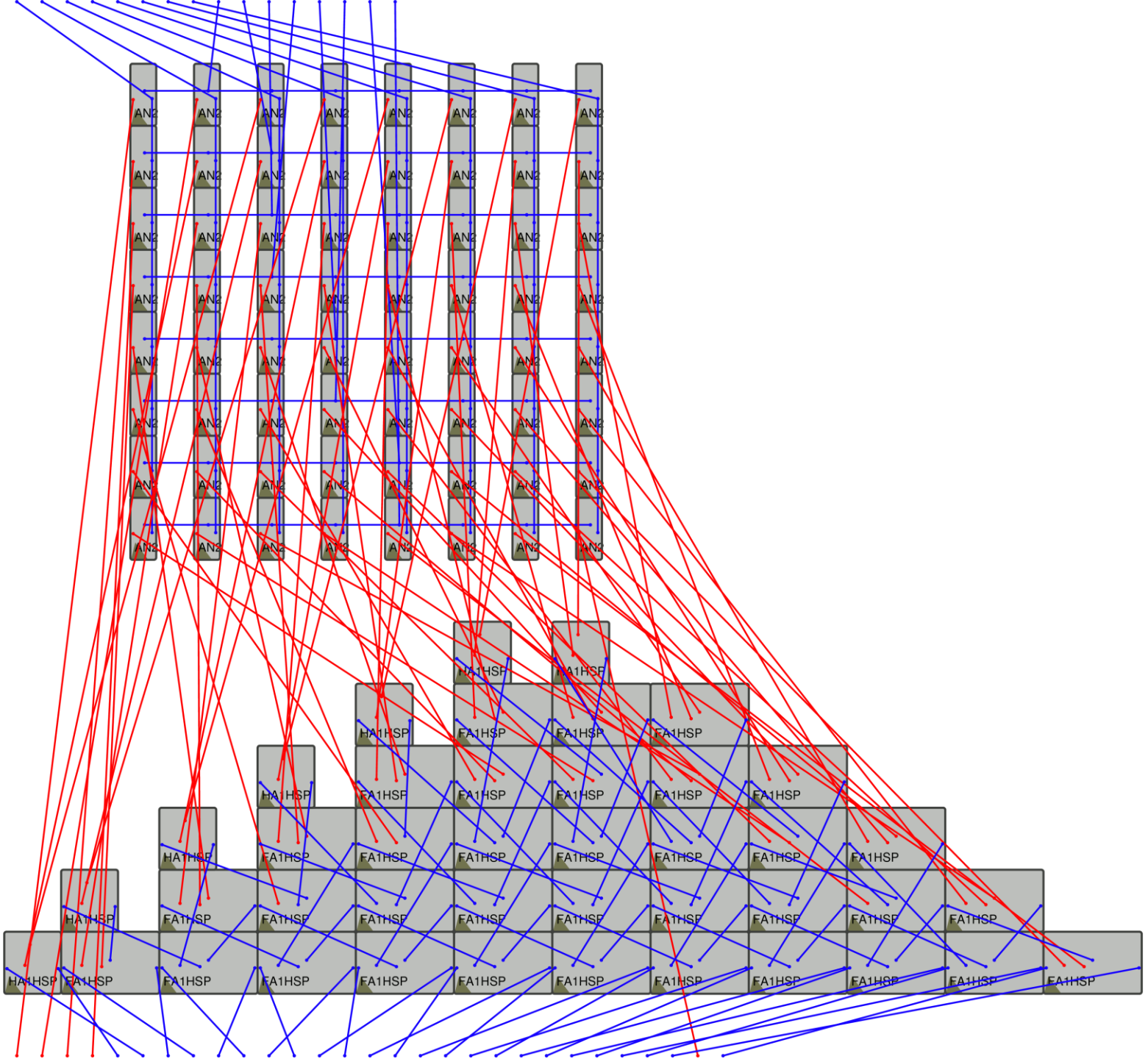
- Based on Lava
- Captures layout exactly
- Optional guiding of wire routing
- Built-in accurate wire-aware timing analysis
- Still embedded in Haskell
  - Can still use our bag of programming tricks
- Has been used in the VLSI design group (K. Subramaniyan) to study multiplier layout.

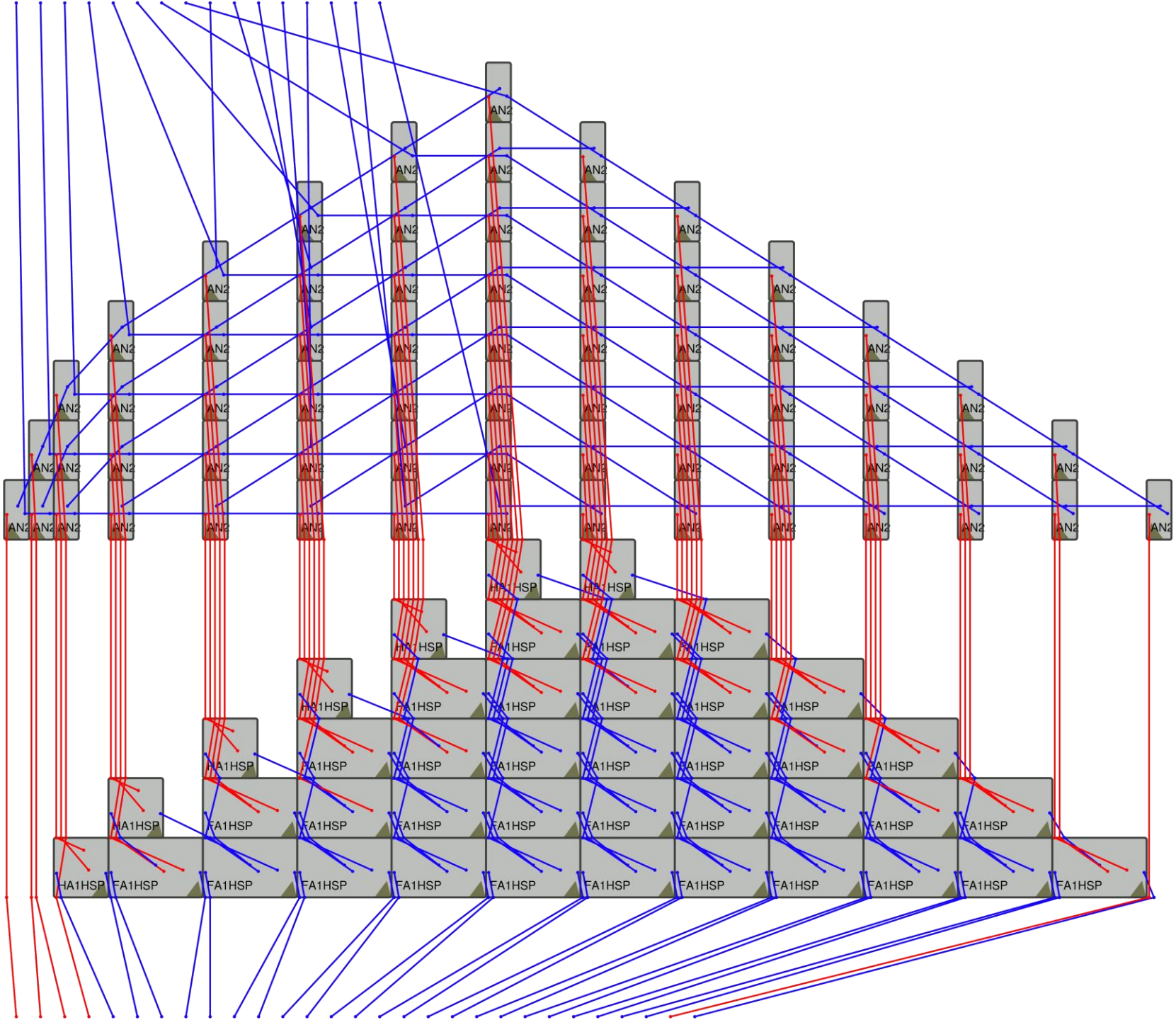
Multiplier in Wired

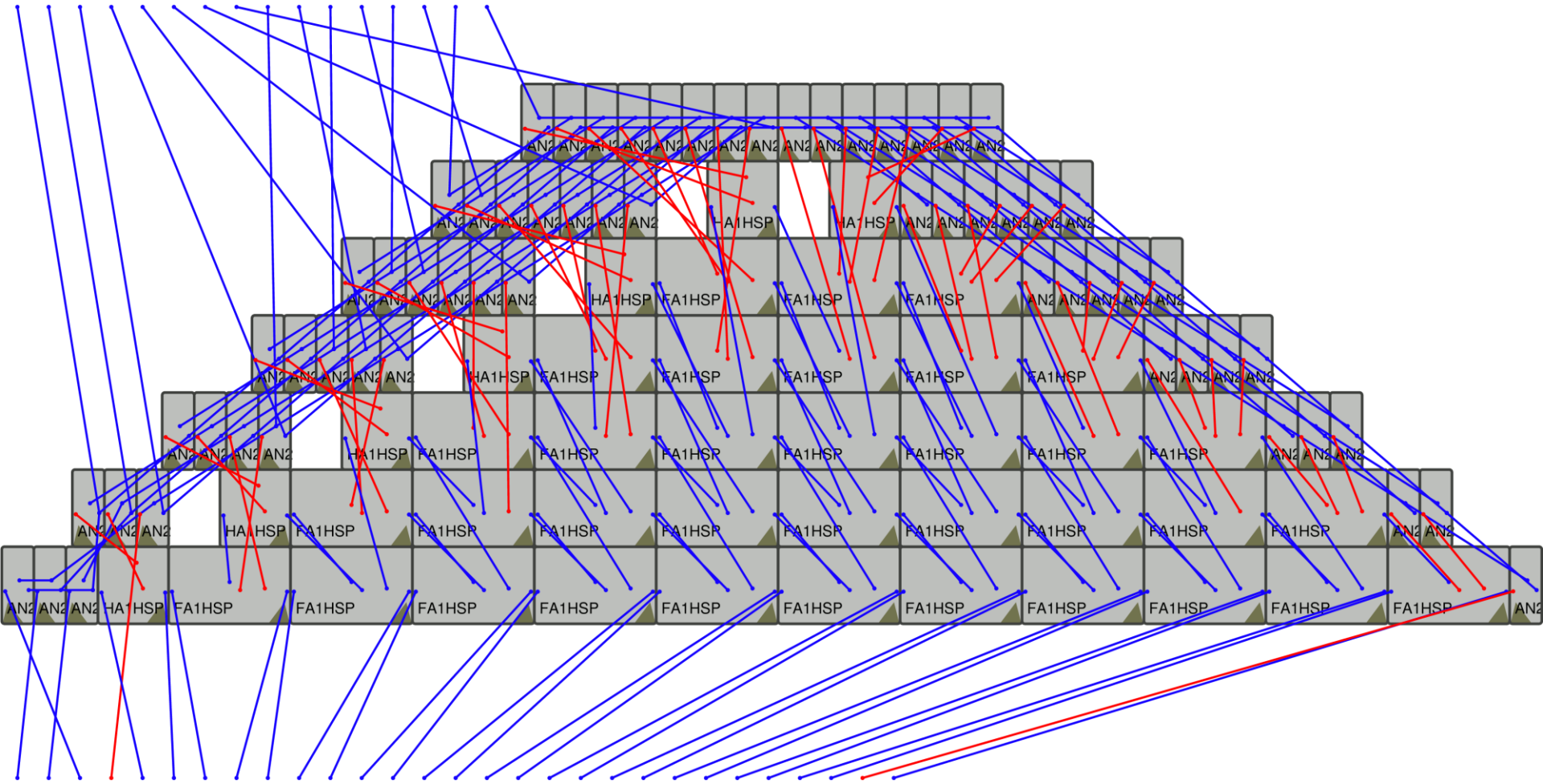
```
hpmTree :: [[Signal]] -> W [[Signal]]
hpmTree = (,) [] .>. (row (hpmColumn 2)) ->- mon fst

hpmMult :: ([Signal],[Signal]) -> W [[Signal]]
hpmMult = genCols ->- hpmTree
```

Multiplier in Wired

```
hpmTree :: [[Signal]] -> W [[Signal]]
hpmTree = (,) []
          .>. (rightwards . row (downwards . hpmColumn 2))
          ->- mon fst

hpmMult :: ([Signal],[Signal]) -> W [[Signal]]
hpmMult = downwards .
     ( bus
   ->- space r
   ->- genCols
   ->- label "pp"
   ->- space r
   ->- hpmTree
   ->- space r
   ->- bus
    )
```


AN2
HA1HSP
FA1HSP


AN2
HA1HSP
FA1HSP


AN2
HA1HSP
FA1HSP

Putting the designer in control

Connection patterns are essential first step (and give some layout awareness when wanted)

We write circuit generators rather than circuit descriptions. Full power of Haskell is available to the user (but we have some useful idioms to reduce the fear).

Circuit generators are short and sweet and LOOK LIKE circuit descriptions.