# Lava III

Mary Sheeran, Thomas Hallgren

# Exercise: Zero detection

Examples of recursively defined circuits

First version

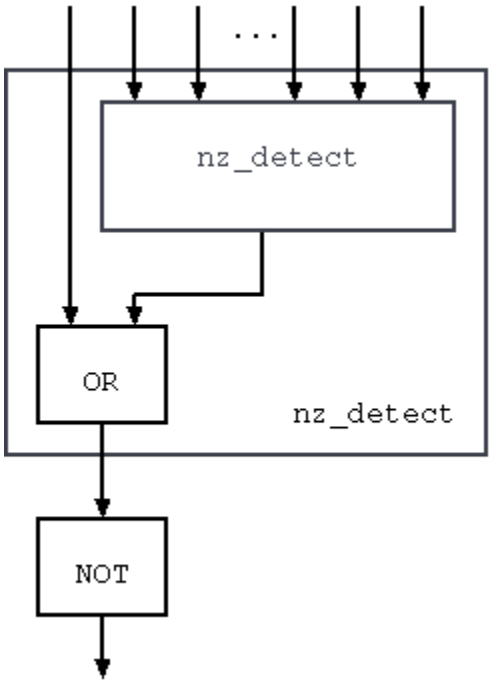
assume we have a circuit that works for n bits,
build a circuit that works for n+1 bits.
Result: a linear chain of 2-input gates

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

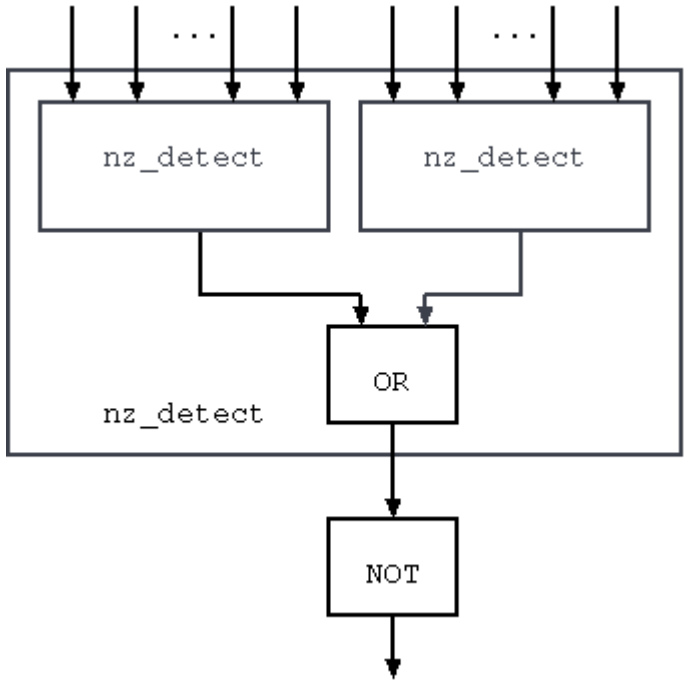
# linear chain

```
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
```

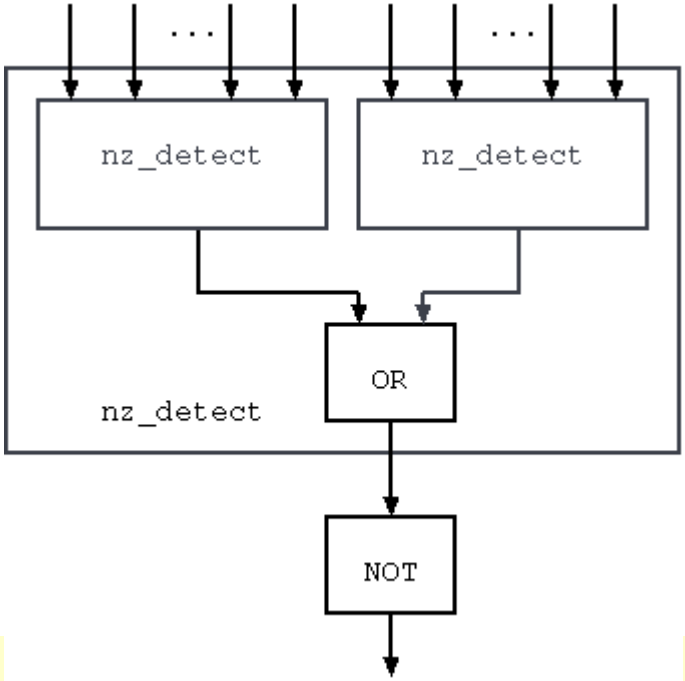


# balanced tree

```
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

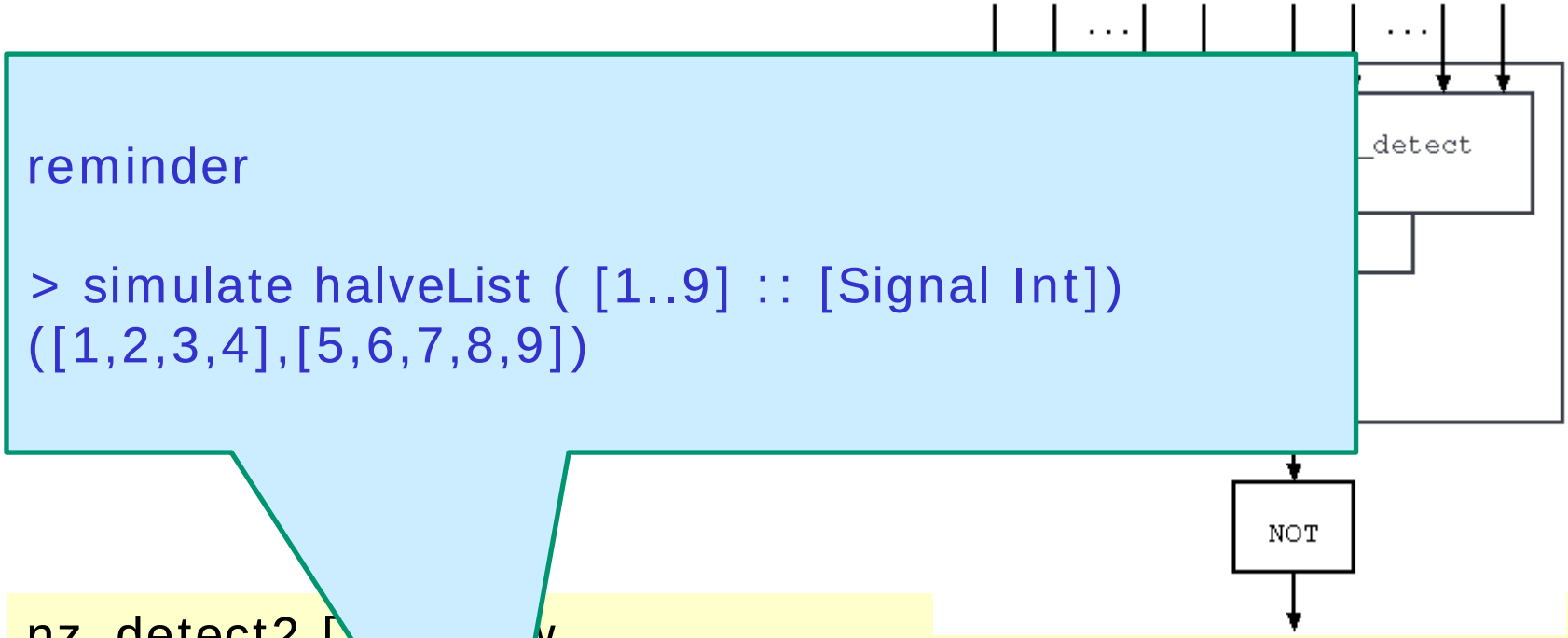



```
nz_detect2 []   = low
nz_detect2 [a] = a
nz_detect2 as  = circ as
  where
      circ = halveList ->- (nz_detect2 -|- nz_detect2) ->- or2
```

# 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  = c
    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  = c
    where
        circ
```

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

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

# Comparing circuits

Comparing behaviour with FV is easy (for fixed size boolean circuits, inc. 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

```
-- 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"

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

```
*Main> dtstL 10
[0,1,2,3,4,5,6,7,8,9] >
```

# Simple delay analysis: Depth computations

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

lin f (a:
lin _ []

dtstLin
```

This kind of analysis is an argument for defining parameterised circuits (rather than hard-wiring in the components)

```
*Main> dtstL 10
[0,1,2,3,4,5,6,7,8,9] >
```

# 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:

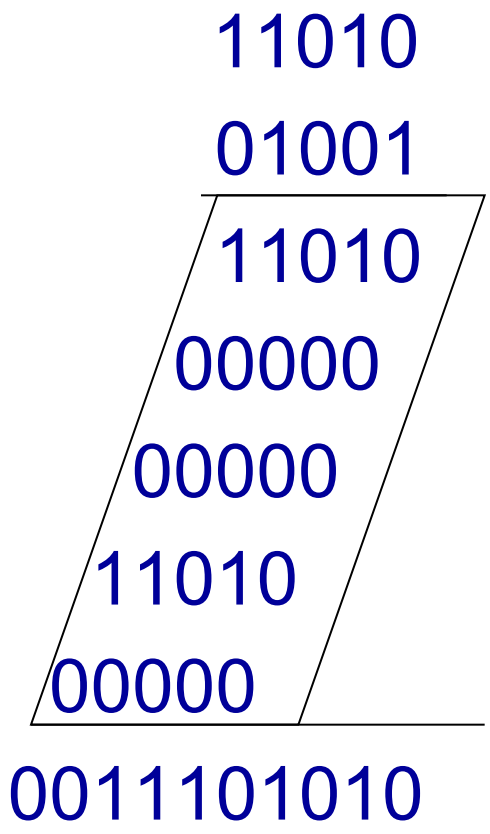
```
rcdeltst = rcAdder2 fI (0, (replicate 10 0, replicate 10 0))
```

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

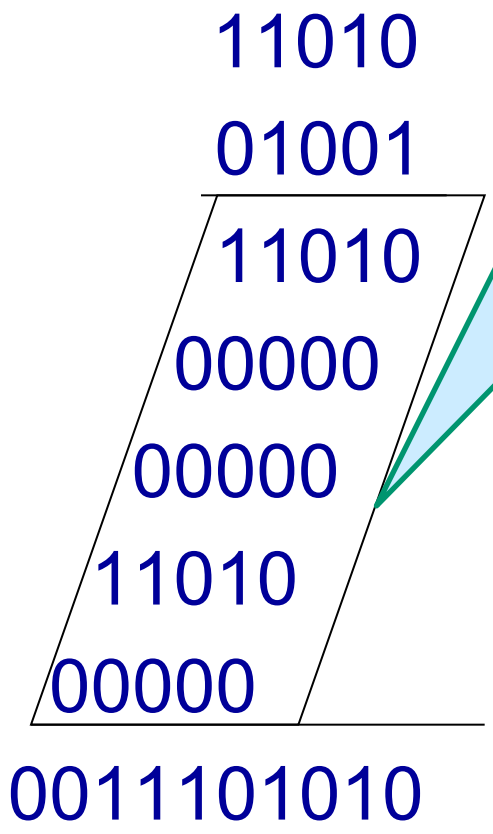
```
rcdelts        simulate (rcAdder2 fI)
                  (0 :: Signal Int, (replicate 10 0, replicate 10 0))
```

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

# Multiplication



# Multiplication



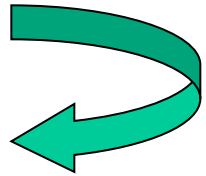

Making a multiplier is about adding up all these numbers (and that is what the Lava lab explores)

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

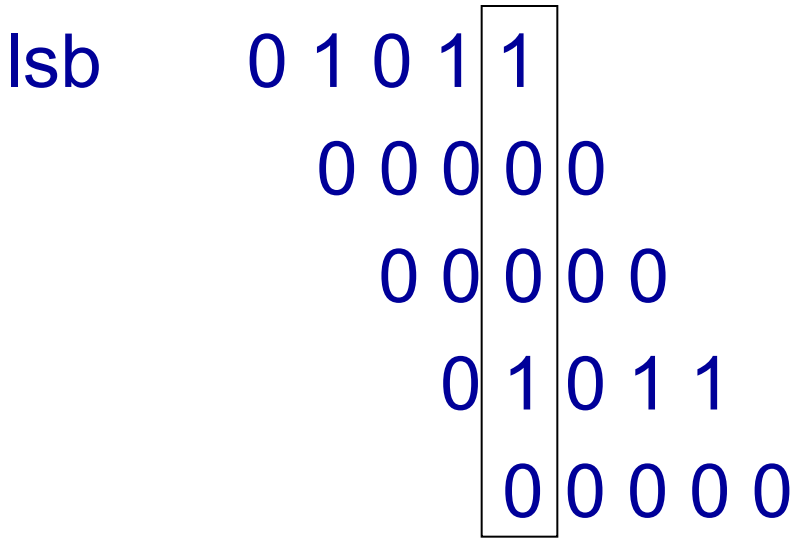
# Multiplication

msb

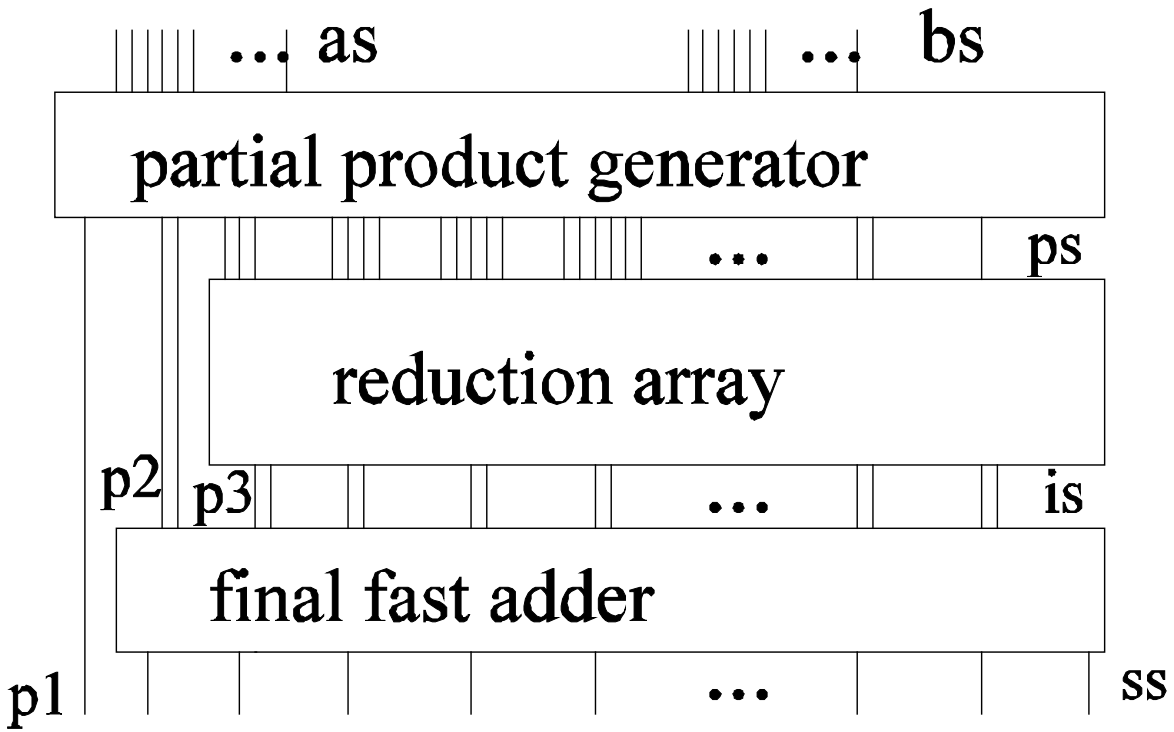
| | | | | | |
|---|---|---|---|---|---|
| | | | 1 | 1 0 1 0 | |
| | | 0 | 0 | 0 0 0 | |
| | 0 | 0 | 0 | 0 0 | |
| 1 | 1 | 0 | 1 | 0 | |
| 0 | 0 | 0 | 0 | 0 | |



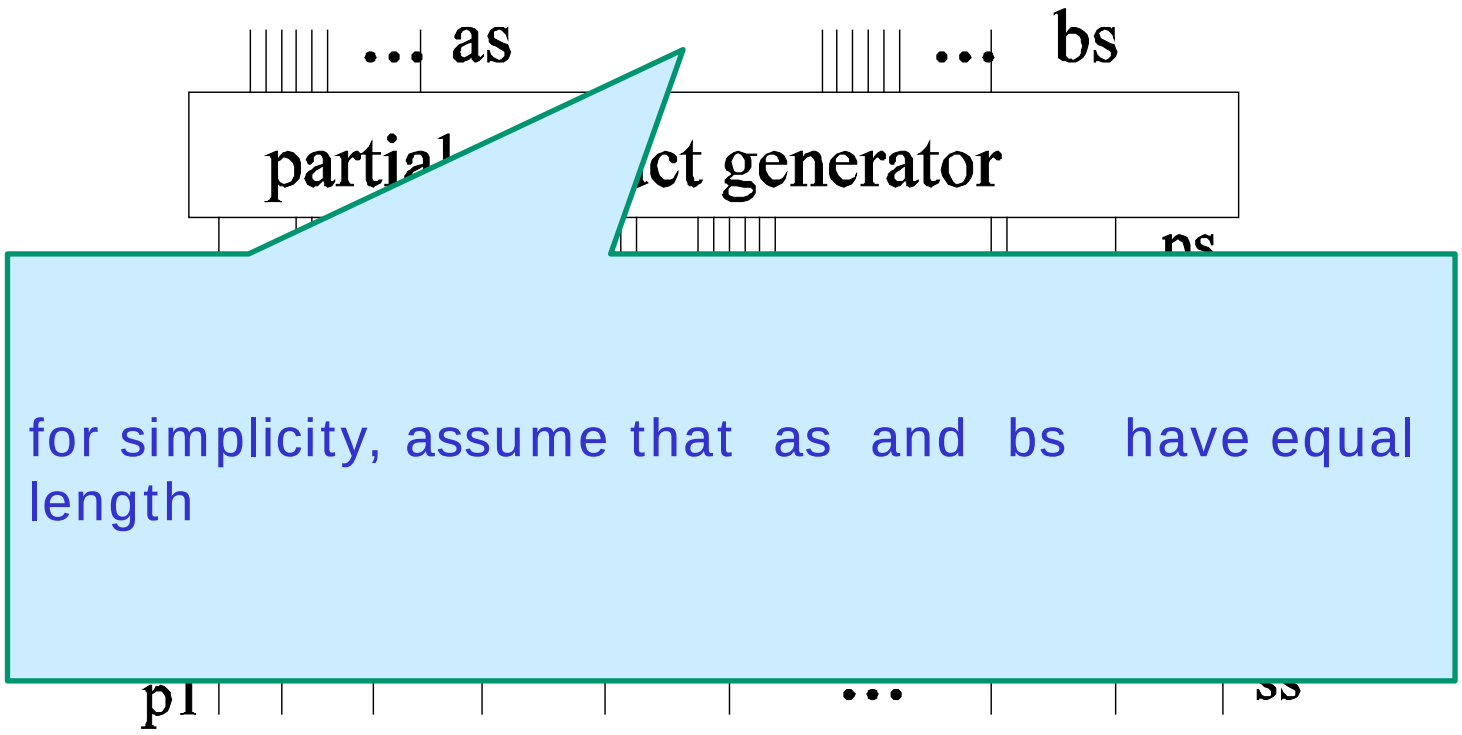
# Multiplication



# Structure of multiplier

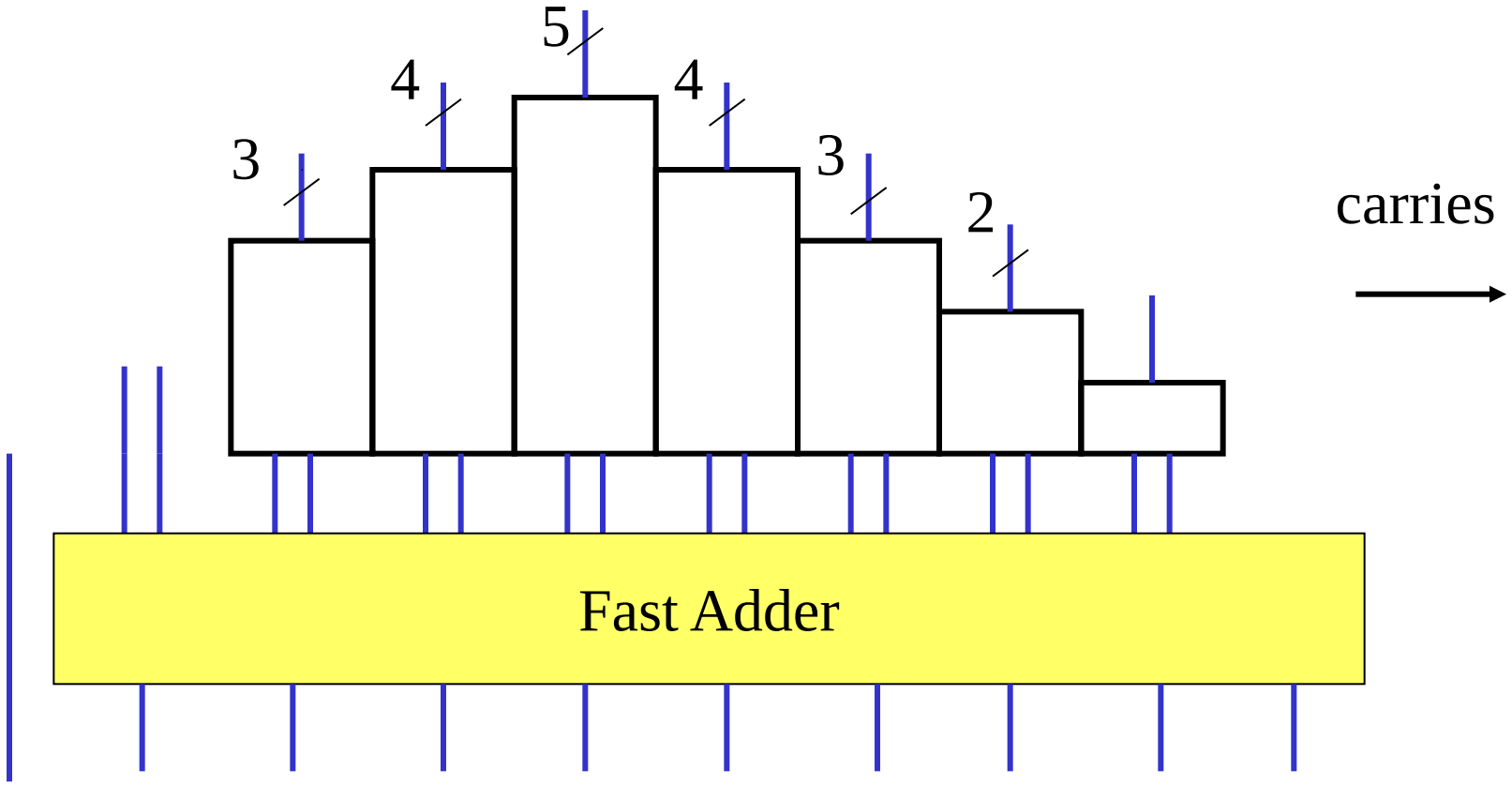


# Structure of multiplier



```
multBin comps (as,bs) = p1:ss
  where
     ([p1]:[p2,p3]:ps) = prods_by_weight (as,bs)
     is                = redArray comps  ps
     ss                = binaryAdder  ([p2,p3]:is)

redArray comps ps = is
  where
    (is,[]) = row (compress comps) ([],ps)
```

# Reduction tree for multiplier



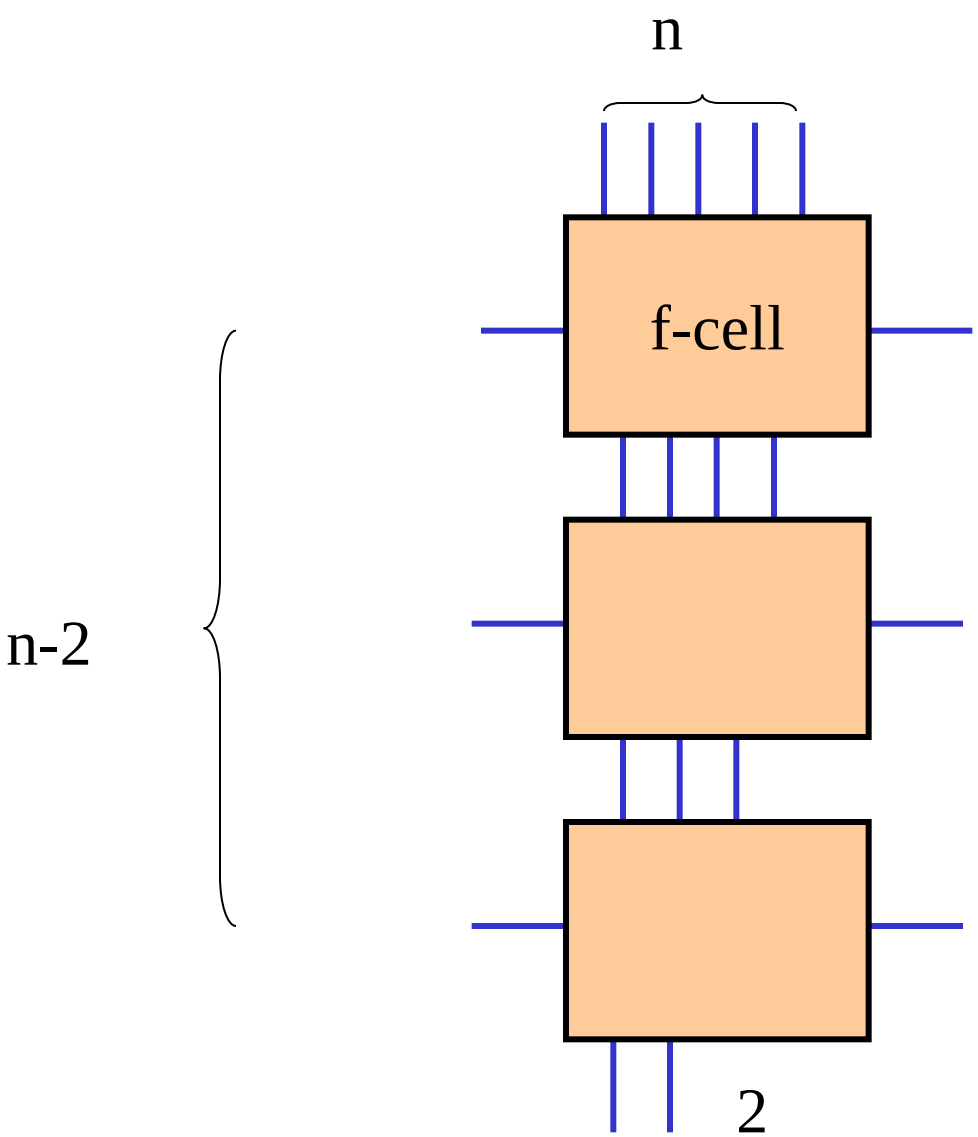
Will concentrate on the reduction tree (a row of compress cells)

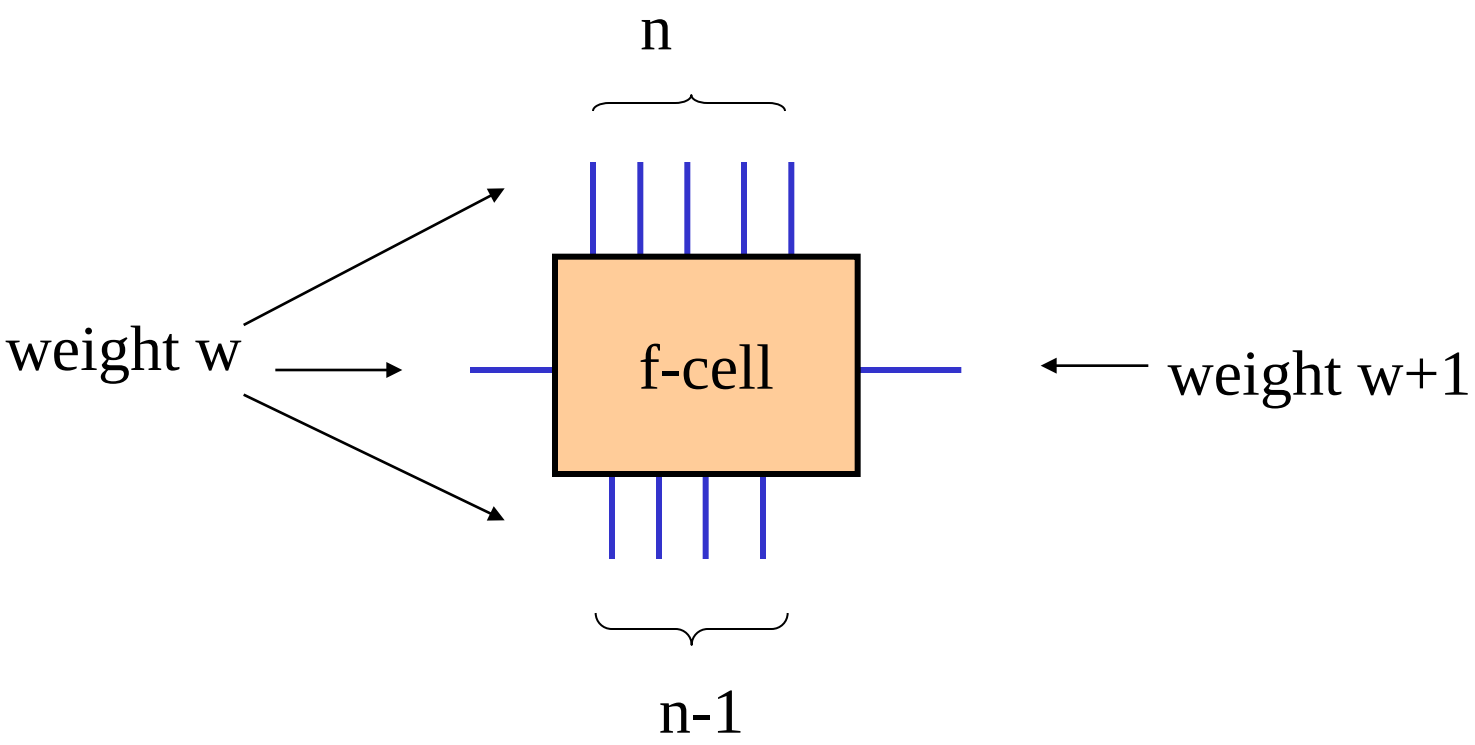
Partial products generated using and gates. May also include recoding to reduce size of tree (cf. Booth)

# (for reference)

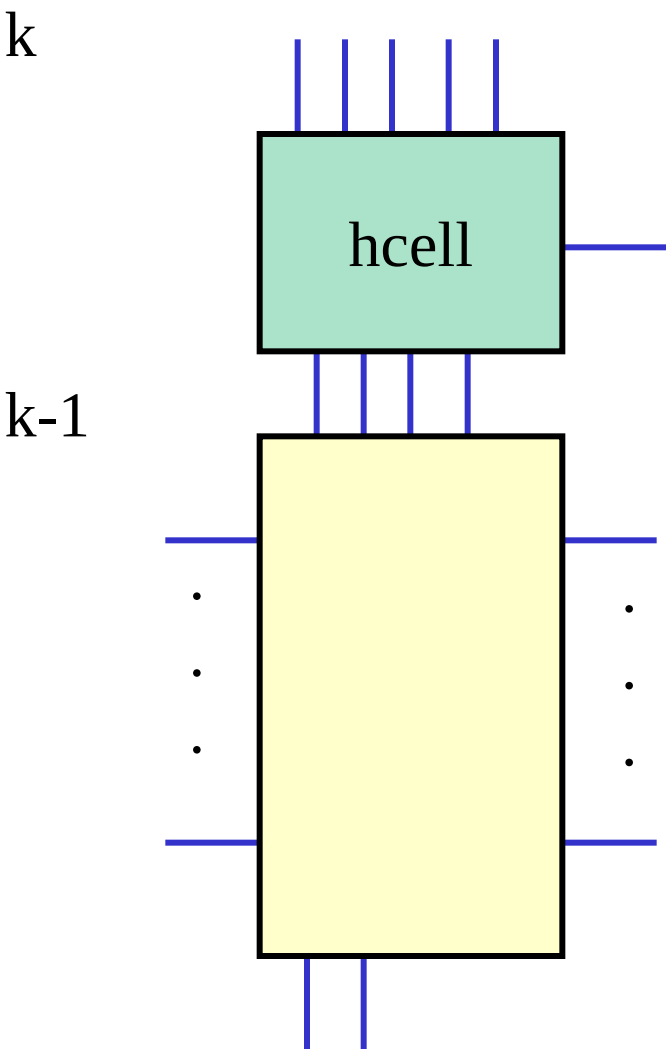
```
prods_by_weight (as,bs)
    = [[and2(a,b) | (a,m)<- number as,
                    (b,n) <- number bs,
                    m+n == i] | i <- [0..(2*(length as)-2)]]
 where
    number cs = zip cs [0..((length cs)-1)]
```

# Compress (diff=2)

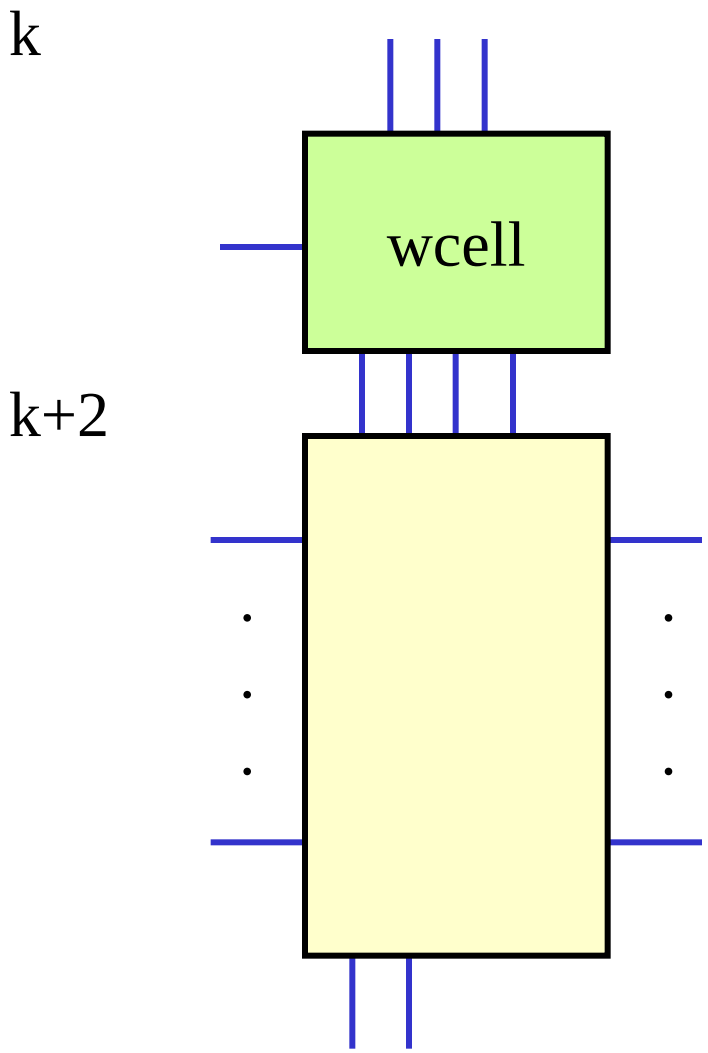


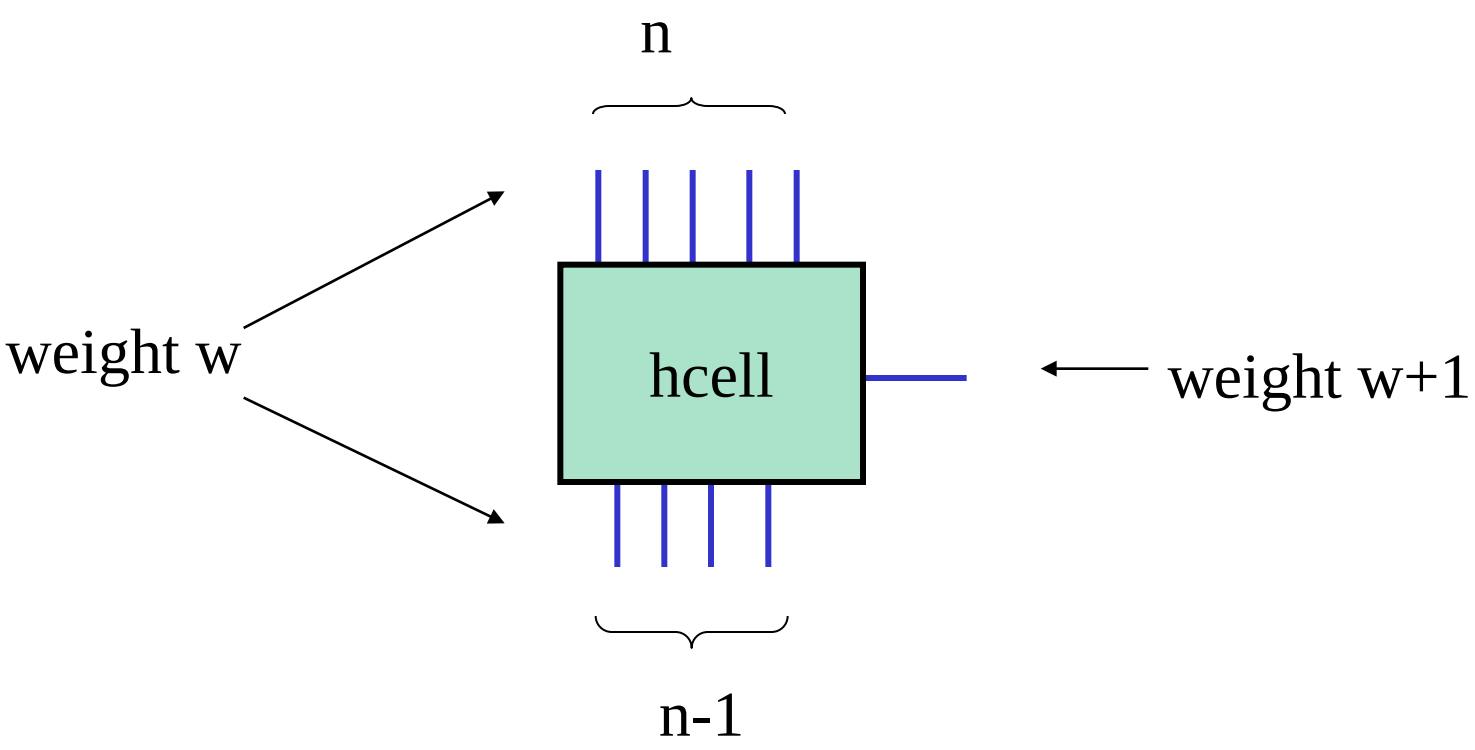

n
weight w
f-cell
weight w+1
n-1

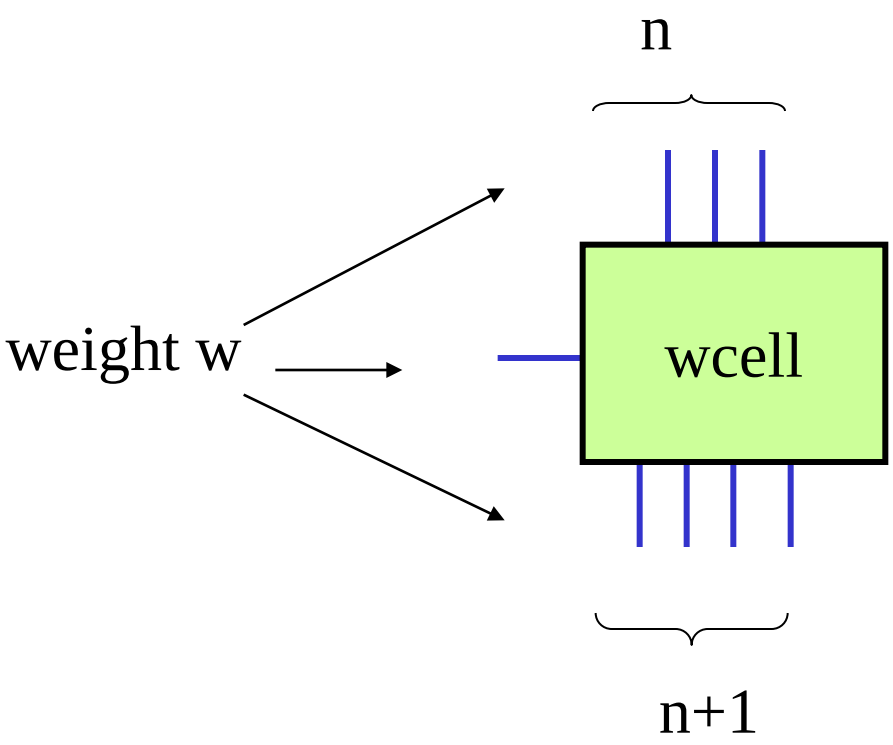
# diff > 2




# diff < 2

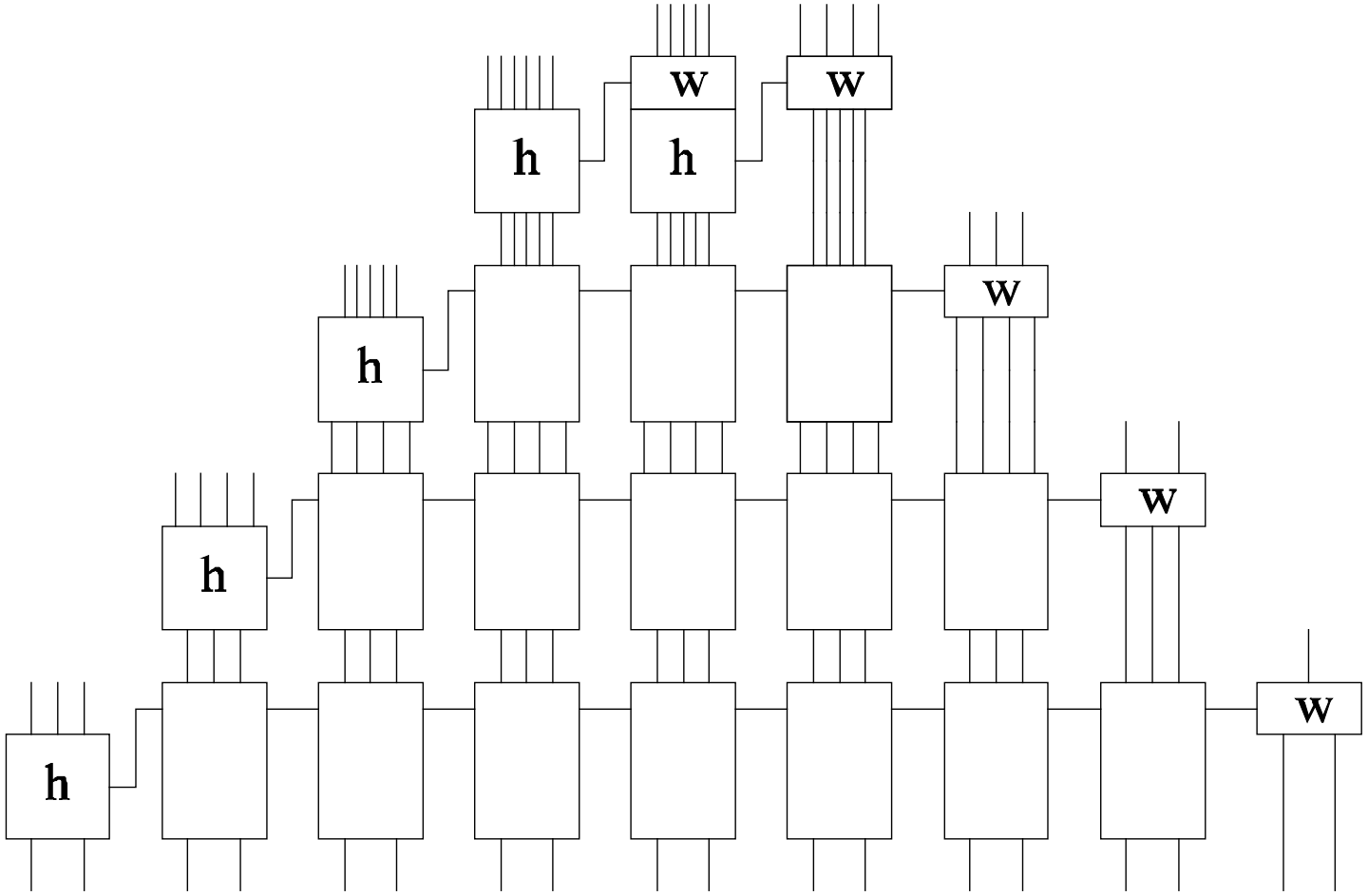



n
weight w
hcell
weight w+1
n-1

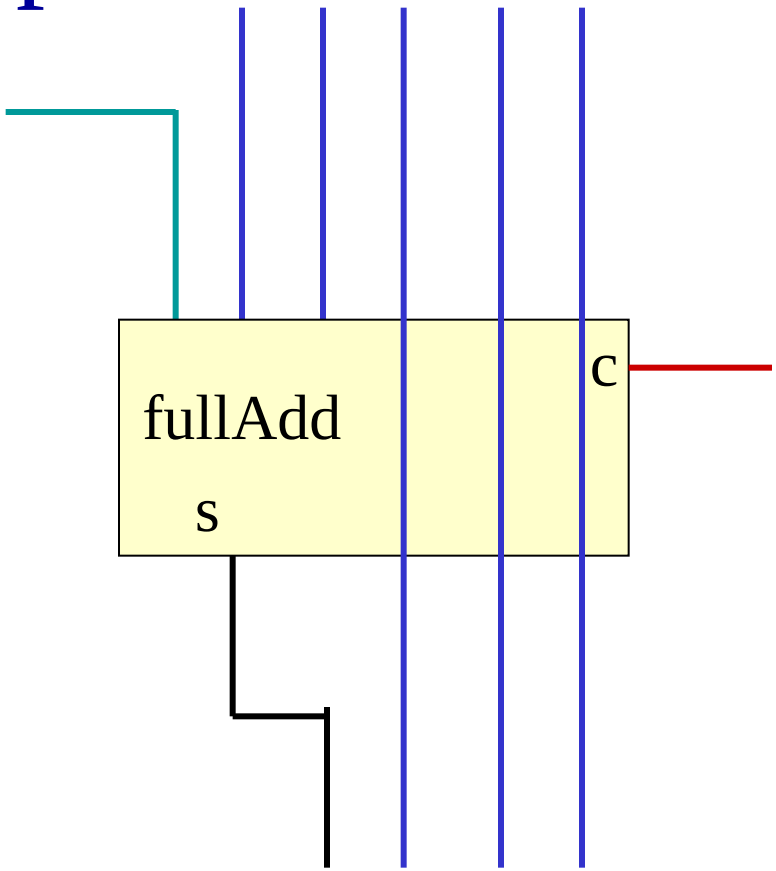

n
weight w
wcell
n+1

```
compress bbs (as,bs) =  comp (as,bs)

  where

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

```
(hAdd,fAdd,iS,iC,w,s2,s3)  =  bbs

fcell  = iC ->- s3 ->- ((fAdd ->- list2Pair)`beside14`
                                (iS `below5` (swap ->- fsT w)))

hcell  = s2 ->- ((hAdd ->- list2Pair) `beside14`
                        (iS `below5` (swap ->- fsT w)))

wcell = iC
```

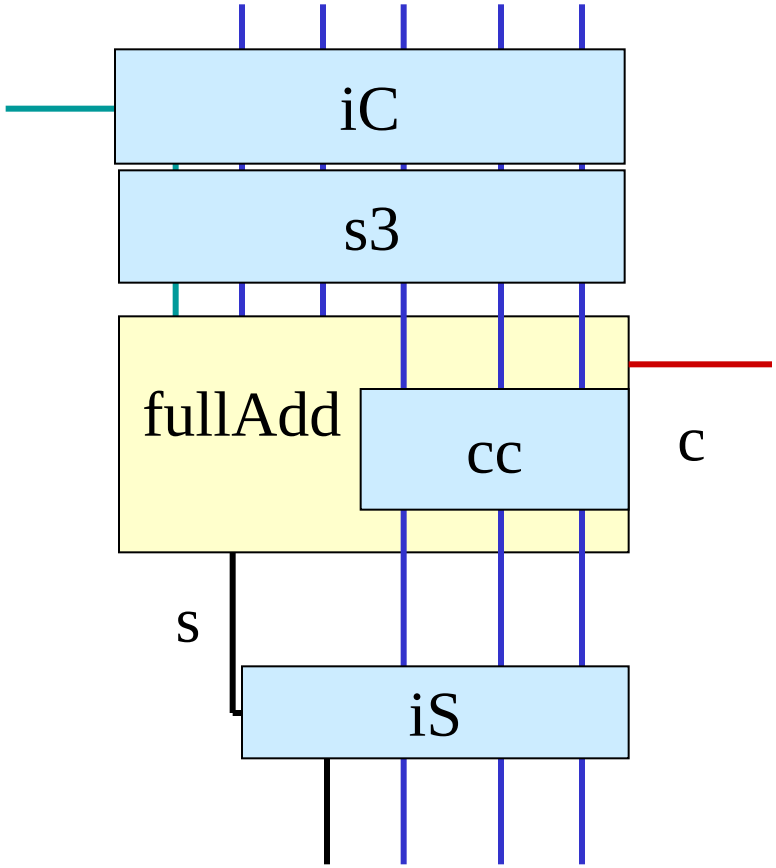

w
w
h
h
w
h
w
h
w
h
w

# possible fcell




halfAdd cells similar. Gives standard array multiplier. Not great!

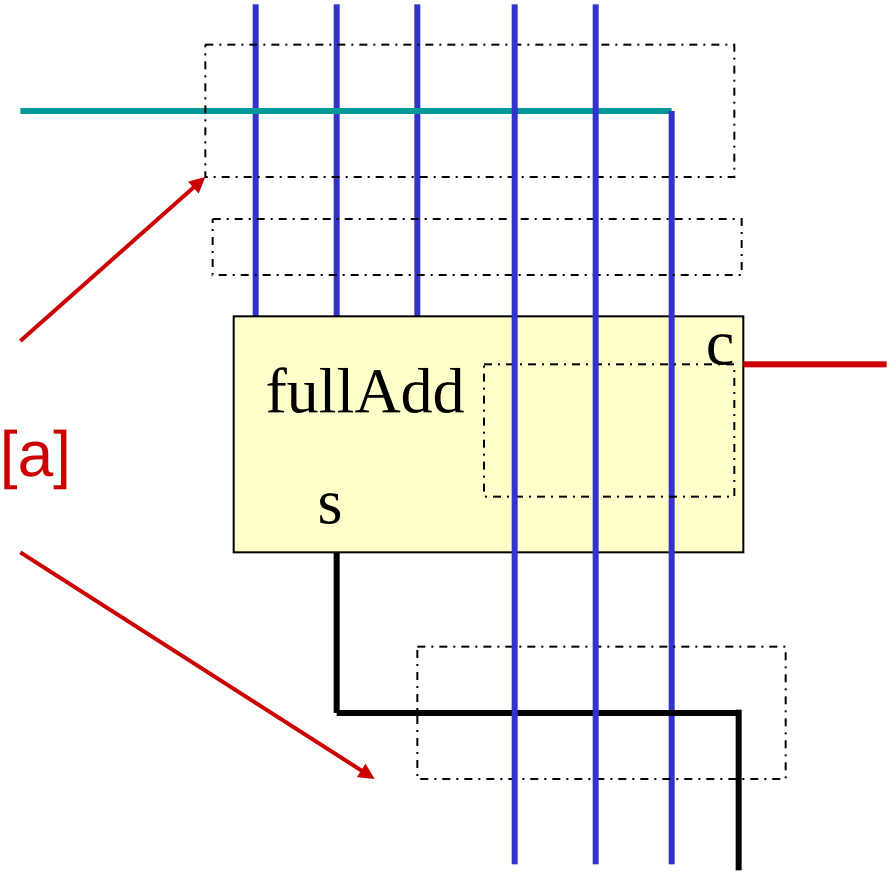
# Only need to vary wiring!
# Make it explicit



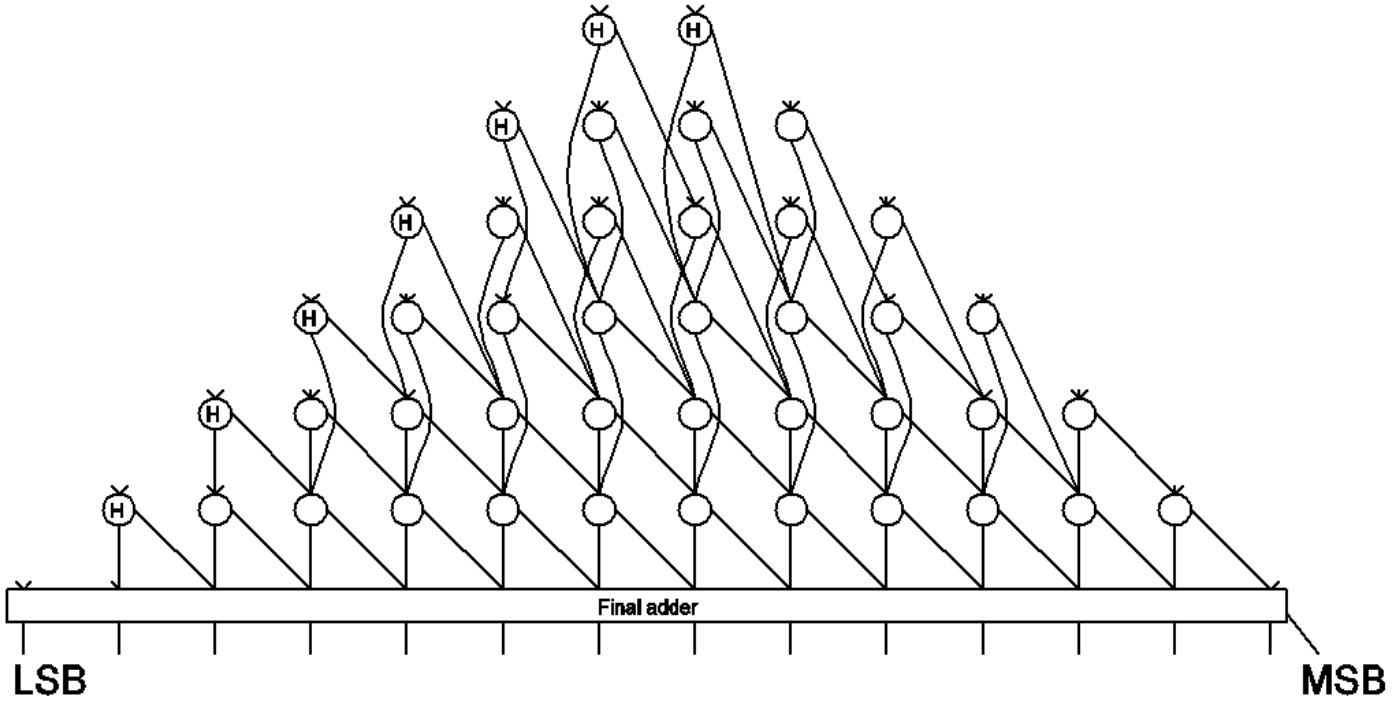
```
(hAdd,fAdd,iS,iC,w,s2,s3)  =  bbs

fcell  = iC ->- s3 ->- ((fAdd ->- list2Pair)`beside14`
                                (iS `below5` (swap ->- fsT w)))

hcell  = s2 ->- ((hAdd ->- list2Pair) `beside14`
                        (iS `below5` (swap ->- fsT w)))

wcell = iC
```

# Dadda-like

toEnd (a,as) = as++[a]

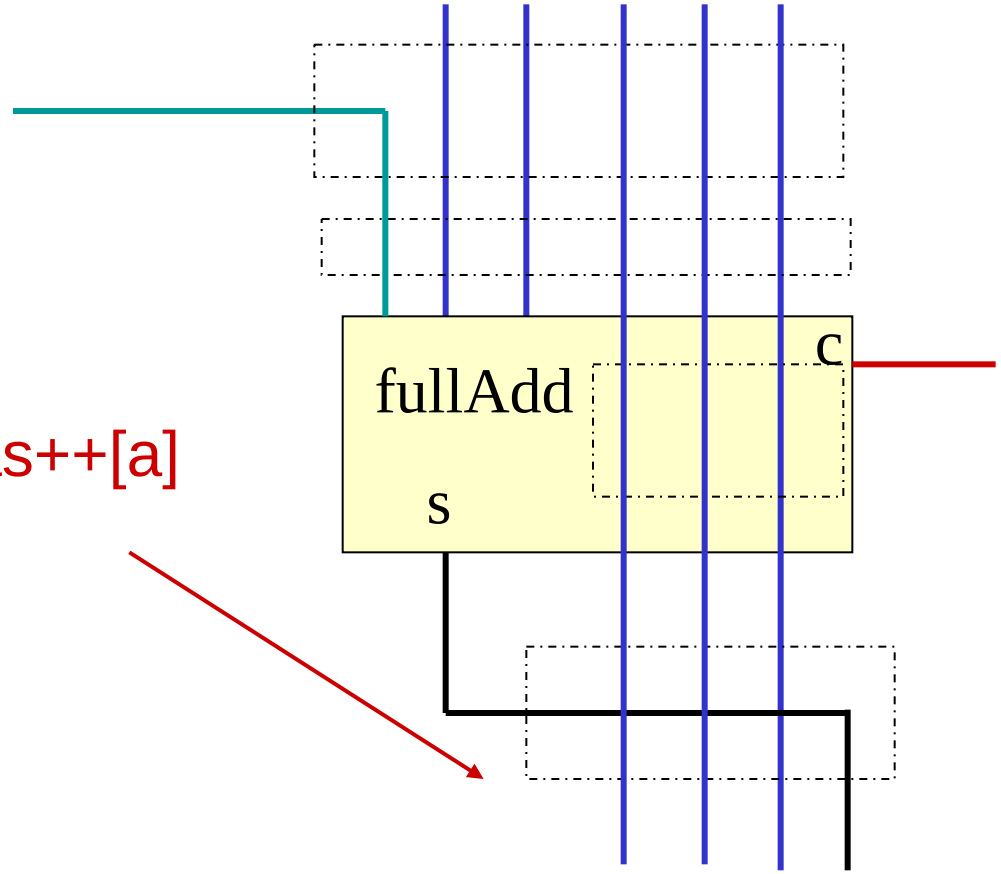



Excellent log depth reduction tree , but known for irregularity, difficult layout



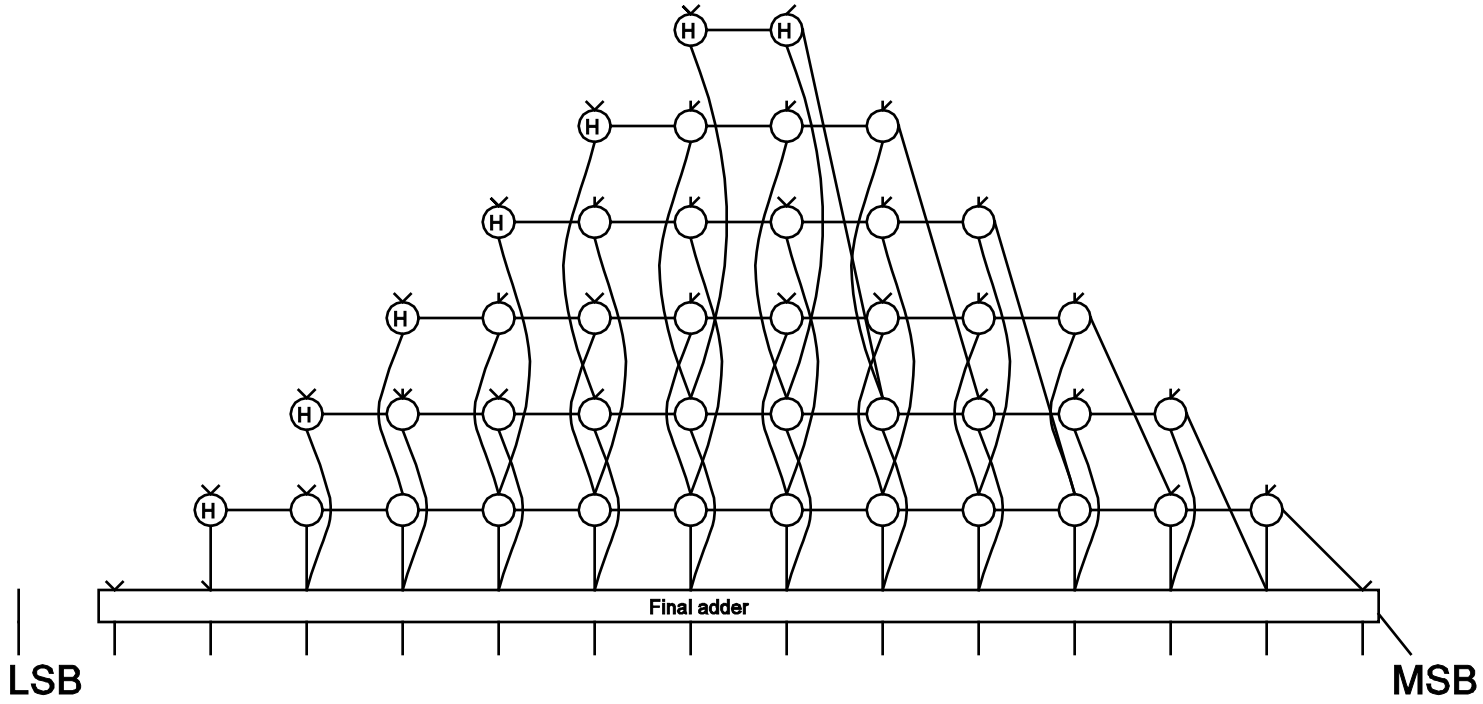

picture by Henrik Eriksson, Chalmers

# Regular reduction tree (Eriksson et al. CE)




toEnd (a,as) = as++[a]

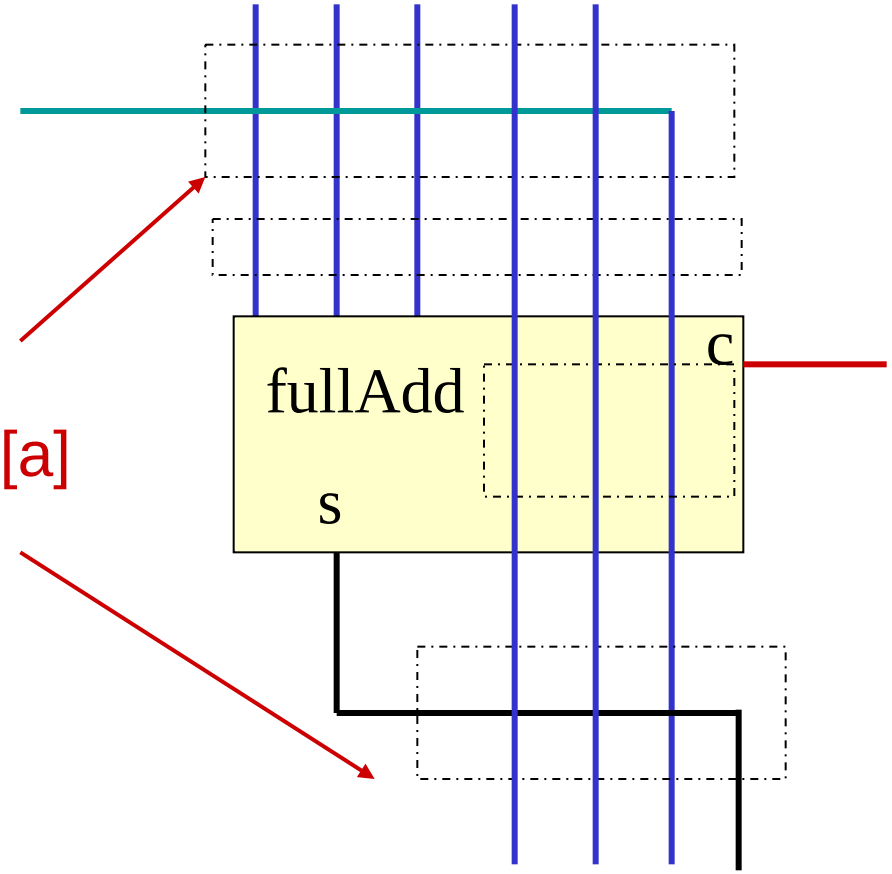
Nowhere near as good as Dadda, but inspired this work




picture by Henrik Eriksson, CE

# Back to Dadda

toEnd (a,as) = as++[a]

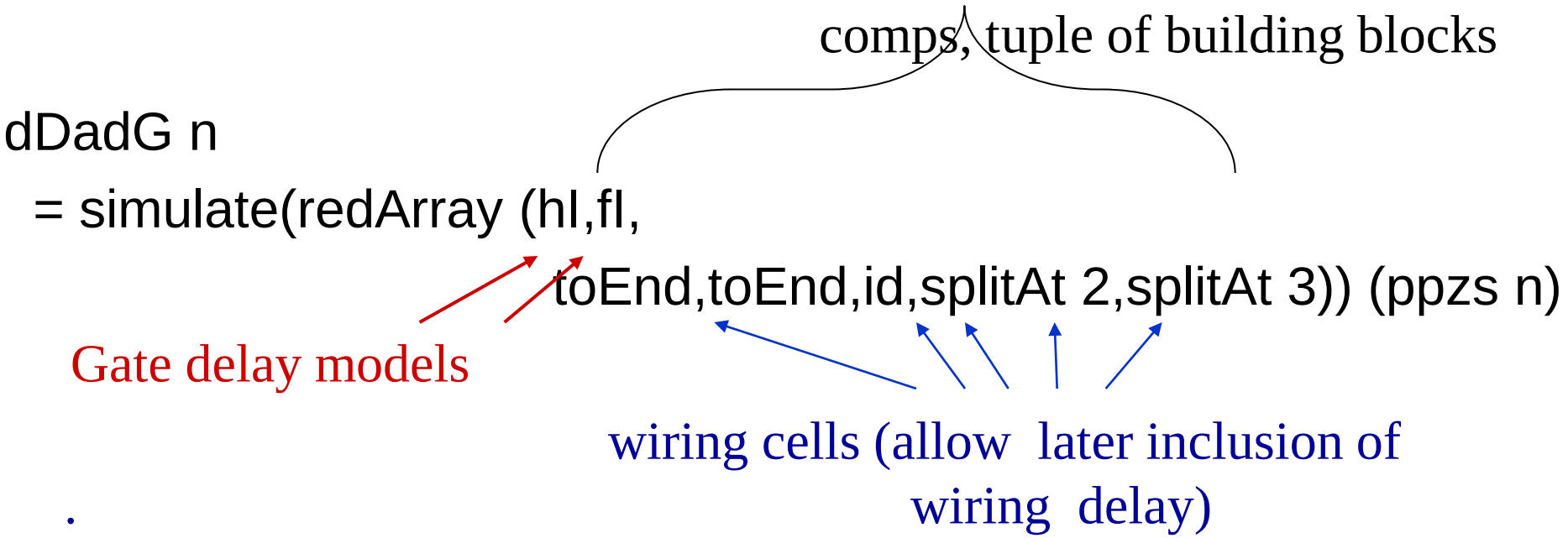


# Simple delay analysis (again)

```
fullAddL [a,b,cc] = [s,c]
  where (s,c) = fullAdd (a,(b,cc))

fAddl (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]


fl :: [Signal Int]  ->  [Signal Int]
fl as = fAddl (20,20,10,10,10,10) as
```

(Have changed the full-adder interface to be “list to list”. Was handier in this example.)

# Checking gate delay




.

(will return to splitAt shortly)

# Checking gate delay (as before)

Main> dDadG 16

[[0,10],[5,20],[20,30],[30,40],[40,50],[50,50],[50,60],[60,70],[70,70],
[70,70],[70,80],[70,80],[80,90],[90,90],[90,90],[90,90],[90,90],[90,90],
[80,90],[80,80],[70,80],[70,80],[70,70],[60,70],[60,60],[50,60],[50,50],
[40,20],[0,20]]

# Checking gate delay (as before)

Main> dDadG 54
[[0,10],[5,20],[20,30],[30,40],[40,50],[50,50],[50,60],[60,70],[70,70],[70,70],[70,80],[70,80],[80,90],
[90,90],[90,90],[90,90],[90,100],[90,100],[90,100],[100,110],[110,110],[110,110],[110,110],[110,110],
[110,110],[110,120],[110,120],[110,120],[110,120],[120,120],[120,130],[130,130],[130,130],[130,130],
[130,130],[130,130],[130,130],[130,130],[130,130],[130,140],[130,140],[130,140],[130,140],[130,140],
[140,140],[140,140],[140,150],[150,150],[150,150],[150,150],[150,150],[150,150],[150,150],[150,150],
[150,150],[150,150],[150,150],[150,150],[150,150],[150,150],[140,140],[140,140],[140,140],[140,140],
[140,140],[130,140],[130,140],[130,140],[130,140],[130,140],[130,130],[130,130],[130,130],[130,130],
[130,130],[130,130],[120,120],[120,120],[120,120],[120,120],[110,120],[110,120],[110,120],[110,110],
[110,110],[110,110],[110,110],[100,100],[100,100],[100,100],[90,100],[90,100],[90,90],[90,90],[80,90],
[80,80],[70,80],[70,80],[70,70],[60,70],[60,60],[50,60],[50,50],[40,20],[0,20]]

# Use of predefined Haskell functions

splitAt is a library function from ”the standard prelude”. See

http://www.haskell.org/definition/haskell98-report.pdf

Reading the standard prelude is a good way to learn! Saves you from reinventing commonly used functions (for example on lists). Your code gets shorter and easier for me to read. (Starting from scratch will not be penalised, if correct!)

# an ordinary Haskell function

```
Main> :t splitAt
splitAt :: Int -> [a] -> ([a],[a])

Main> splitAt 7 [1..10]
([1,2,3,4,5,6,7],[8,9,10])

Main> splitAt 7 [1..3]
([1,2,3],[])

Main> splitAt 2 [1..10]
([1,2],[3,4,5,6,7,8,9,10])
```

# Verifying the multiplier

```
multDadda (as,bs) = ps
  where
       ps = multBin(halfAddL,fullAddL,
                          toEnd,toEnd,id,splitAt 2,splitAt 3)

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

```
prop_mults mymult n
  = forAll (list n) $ \as ->
     forAll (list n) $ \bs ->
      propEQ multi mymult (as,bs)
```

OR

```
prop_mults mymult n
  = forAll (list n) $ \as ->
     forAll (list n) $ \bs ->
      multi(as,bs) <==> mymult (as,bs)
```

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

See section 4.2 of Lava tutorial (replace verify by smv)

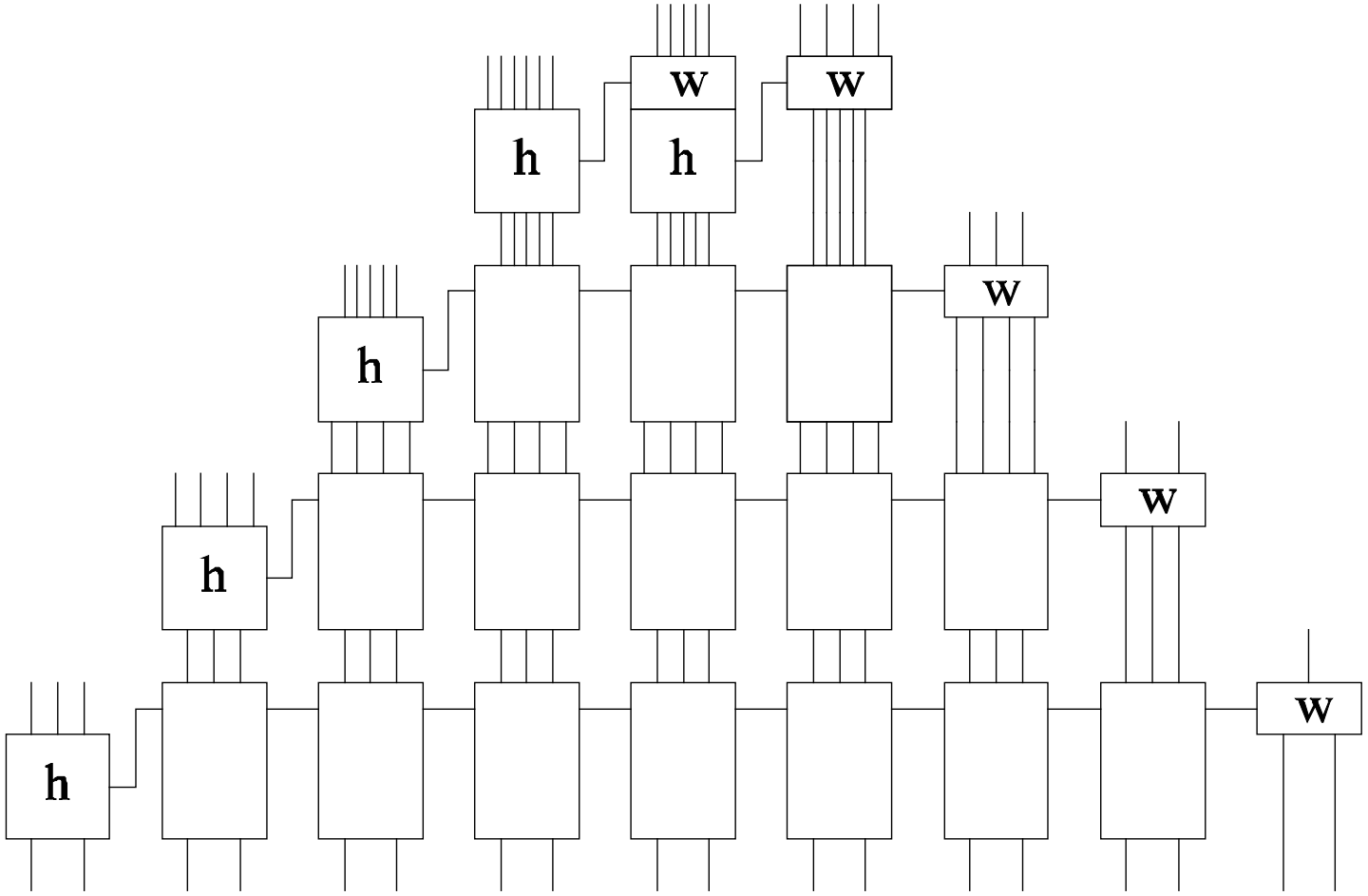
# The cool thing

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

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....


w
w
h
h
w
h
w
h
w
h
w

# Note

Layout for the Dadda-like tree is no more difficult than for any of the others. Important in practice!

We call it the High Performance Multiplier reduction tree (Henrik, Per, Mary :)

Henrik Eriksson, CE, had first idea and then my mult. descriptions suggested something similar. This led to a layout strategy, which Henrik followed.

Next step is to generate layout from Wired (wire-aware version of Lava)

# Promising, but we can do better!

Choose what wiring cells to use dynamically, during circuit generation, rather than in advance

Base choice on delay behaviour of both wires and components

# Shadow Values

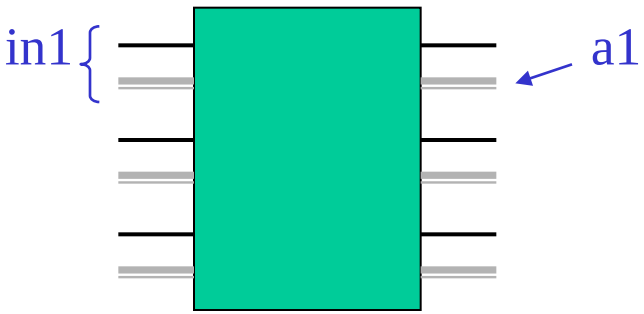
Main> tomarked (map (*2)) [(1,True),(3,False),(5,True)]

[(2,True),(3,False),(10,True)]

Can use same idea to prune unwanted parts of circuits. Pair dummy ”wires” with False and then use

pattern (tomarked s)

# Clever Components

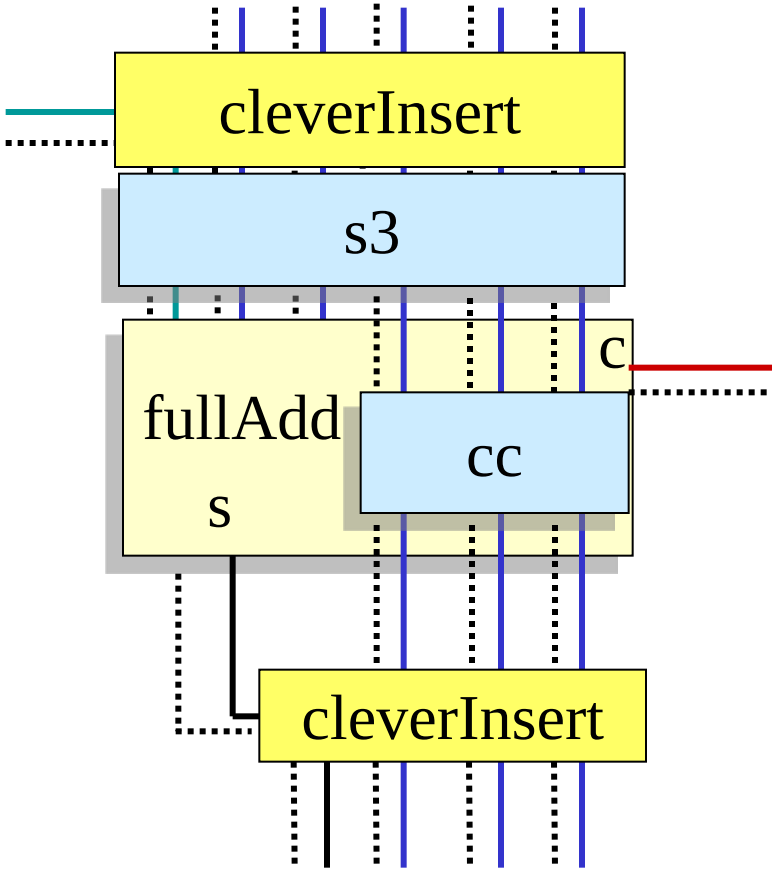



decide what component to be based on shadow values input (A,used here)

can even try several components and decide which to be by looking at shadow values produced!! (B,used to make small median circuits)

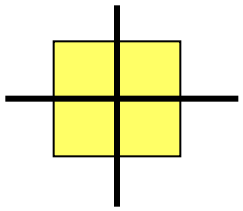
Try it and see during generation

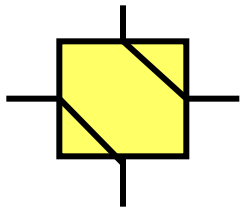
Idea: Harden the wiring during circuit generation using clever circuits. Shadow values estimate delay through wires and cells.



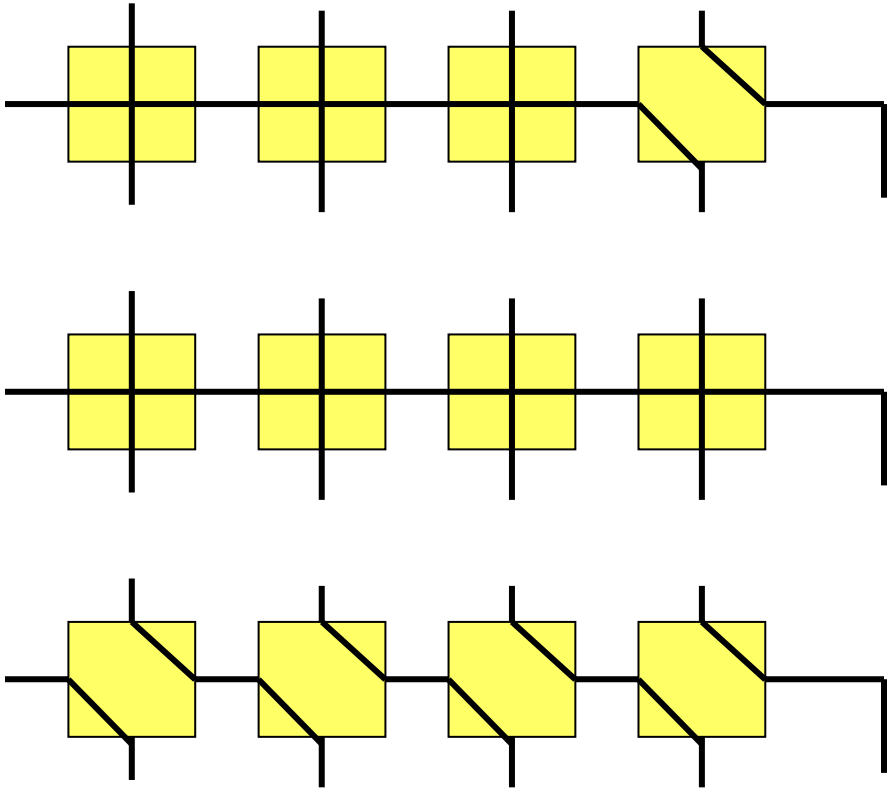
cswap((a,x),(b,y))

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





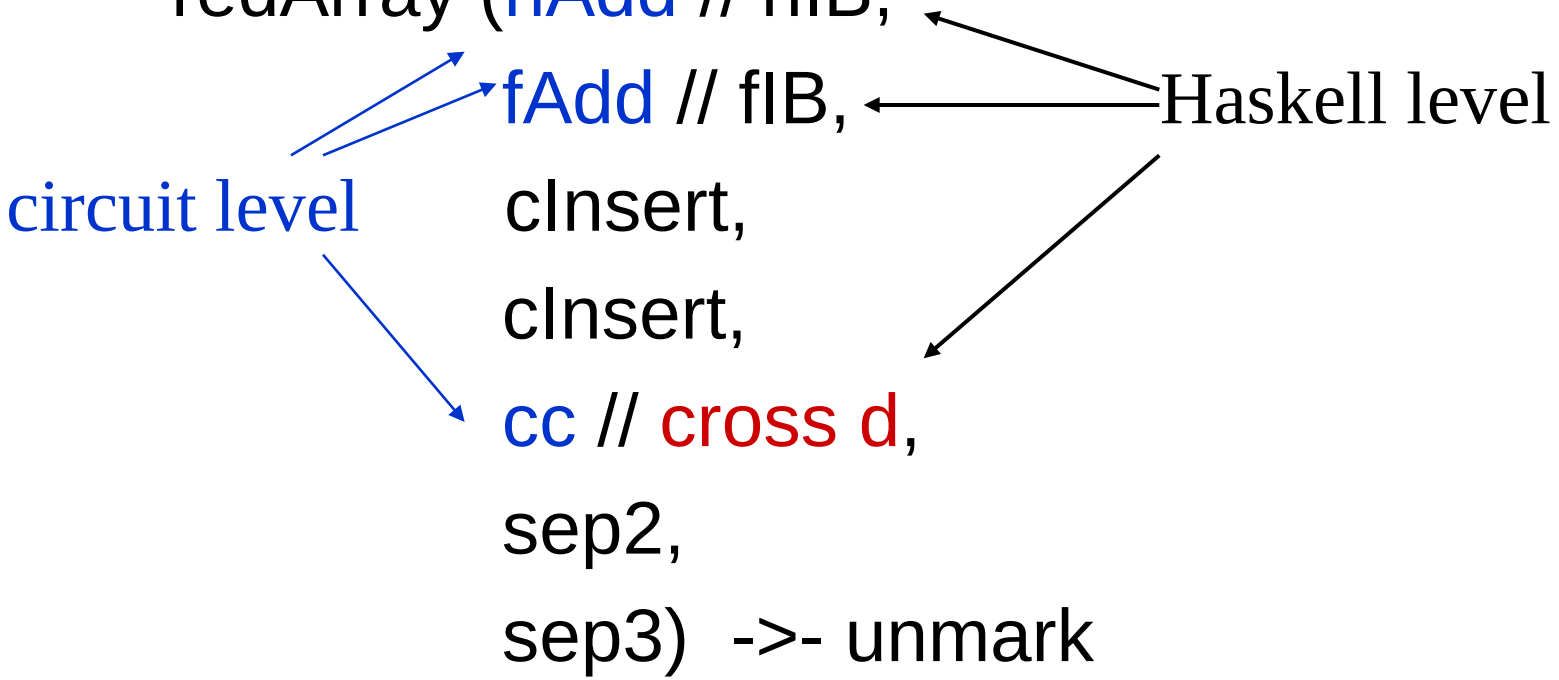
cleverInsert = row cswap ->- apr



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

# Structure of circuit generator remains unchanged

```
adapt (hAdd, fAdd, cc) (d,pds)
=  mmark pds ->-
       redArray (hAdd // hIB,
                 fAdd // fIB,
                 cInsert,
                 cInsert,
                 cc // cross d,
                 sep2,
                 sep3)  ->- unmark
```



# Better than Dadda

```
Main> getDiff delDaddaGW delAdGW 16

([[0,0],[-12,12],[12,0],[0,2],[2,0],[0,12],
   [12,4],[4,3],[3,12],[12,8],[8,9],[9,7],
   [7,3],[3,9],[9,11],[11,7],[7,6],[6,5],[5,5],
   [5,5],[20,3],[19,2],[3,3],[4,3],[22,2],[20,2],
   [21,0],[43,-24],[0,0]],[])
```

# Better than TDM

Main> getDiff delTDMGW delAdGW 54

([[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,4],[4,0],
[0,0],[0,0],[0,1],[1,4],[4,0],[0,4],[4,0],[0,0],[0,6],[6,6],[6,3],[3,4],
[4,7],[7,2],[2,2],[2,3],[3,4],[4,-3],[-3,8],[8,8],[8,12],[12,6],[6,9],[9,5],
[5,8],[8,2],[2,7],[7,3],[3,7],[7,2],[2,5],[5,6],[6,5],[5,12],[12,17],[17,14],
[14,11],[11,13],[13,10],[10,11],[11,18],[18,14],[14,10],[10,9],[9,11],[11,13],
[13,13],[13,16],[16,16],[16,16],[16,16],[17,17],[18,18],[18,18],[17,18],
[17,17],[17,16],[16,2],[2,3],[3,3],[3,6],[6,6],[6,7],[8,7],[8,8],[8,12],
[13,12],[13,13],[5,13],[11,5],[12,1],[2,2],[2,2],[2,6],[6,6],[7,6],[6,7],
[6,6],[-1,6],[0,1],[2,2],[2,2],[1,2],[1,1],[-1,1],[0,-1],[0,0],[0,0],[0,0],
[0,0],[0,0],[0,0],[0,0]],[])

# Result (multiplication)

Simple parameterised description of fast adaptive multiplier

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)

Captures layout exactly

Can still use our bag of programming tricks
(still embedded in Haskell)

Quick but relatively accurate design exploration

Being pursued in the VLSI design group (K. Subramaniyan)

# Obvious questions

This is very low level. What about higher up, earlier in the design?

(Tentative assertion: these were general programming idioms with possible application at other levels of abstraction.)

What about the cases when such a structural approach is inappropriate? Datapath vs. control

Can we make refinement work?

Can we design appropriate GENERIC verification methods?

# 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. Everything is done behind the scenes by symbolic evaluation. 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.

# It’s all about programming

Non-standard interpretation used after generation (as we have long done) and now also to guide synthesis

Clever circuits a good idiom. Can control choice of components, wiring and topology. Greatly increase expressive power of the connection patterns approach.

Having a full functional language available is a great thing once one has had some practice. More idioms to be discovered (for example multi-format circuits)

Ideas compatible (I believe) with Intel’s IDV

# We can’t only think about function

Clever circuits give a way to allow non-functional properties to influence design (even early on). Makes blocks context sensitive. (Can make modelling finer)

Vital as we move to deep sub-micron

Separation of concerns becoming less and less possible

We need to study the algebra of the connection patterns with this in mind

# You should think about

The two different design flows that you have seen

What was good and bad about them

YOUR opinions based on your experience (which is influenced by previous expertise)