Lava 4

brief info for TH exam

Code for this lecture provided in EFile12.hs on the Assignments page.

The file DrawPP12.hs is for drawing pictures.

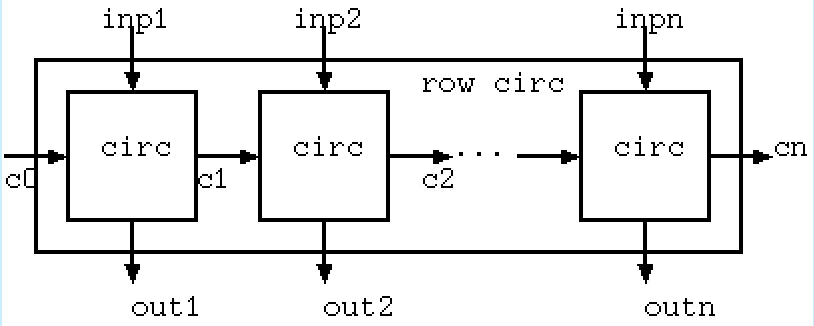
Q: How can we speed this up?

```
adder0 :: [(Bit, Bit)] -> ([Bit], Bit)
adder0 abs = row fullAdd (low,abs)
```

Q: How can we speed this up?

```
adder0 :: [(Bit, Bit)] -> ([Bit], Bit)
adder0 abs = row fullAdd (low,abs)
```

row is a connection pattern

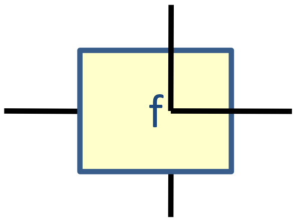


A: Compute carries separately

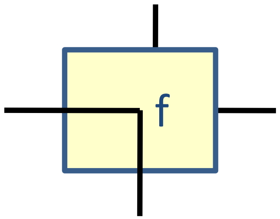
will show a sequence of functions that each have same behaviour

Some useful stuff

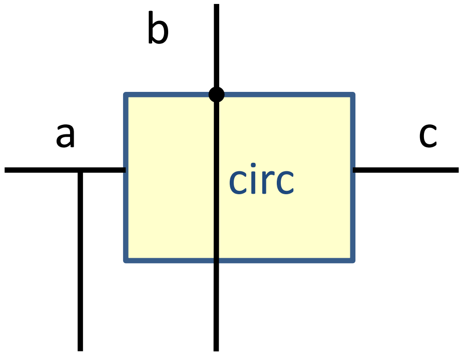
fsT f = (f -|- id)



snD f = (id -|- f)

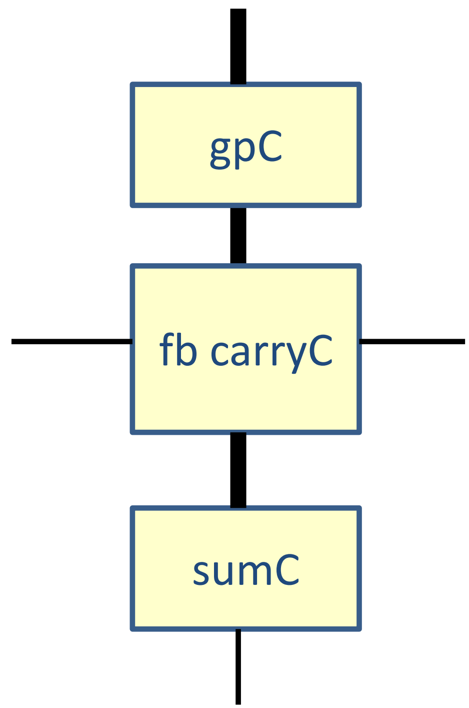


fb :: ((a,b) -> c) -> (a,b) -> ((a,b),c)
fb circ ab = (ab, circ ab)



another view of full adder

```
fullAdd1 :: (Bit,(Bit,Bit)) -> (Bit,Bit)
fullAdd1 = snD gpC ->- fb carryC ->- fsT sumC
```

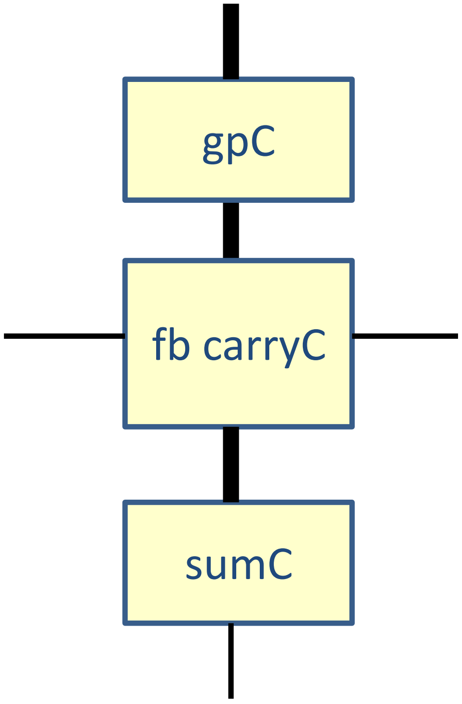


```
gpC:: (Bit,Bit) -> (Bit,Bit)
gpC (a,b) = (a <&> b,a <#> b)

sumC :: (Bit,(Bit,Bit)) -> Bit
sumC (cin, (_,p)) = cin <#> p

carryC :: (Bit,(Bit,Bit)) -> Bit
carryC (cin, (g,p)) = g <|> (cin <&> p)
```

Why this structure?

carryC can be taylored to have short delay from carry in to carry out

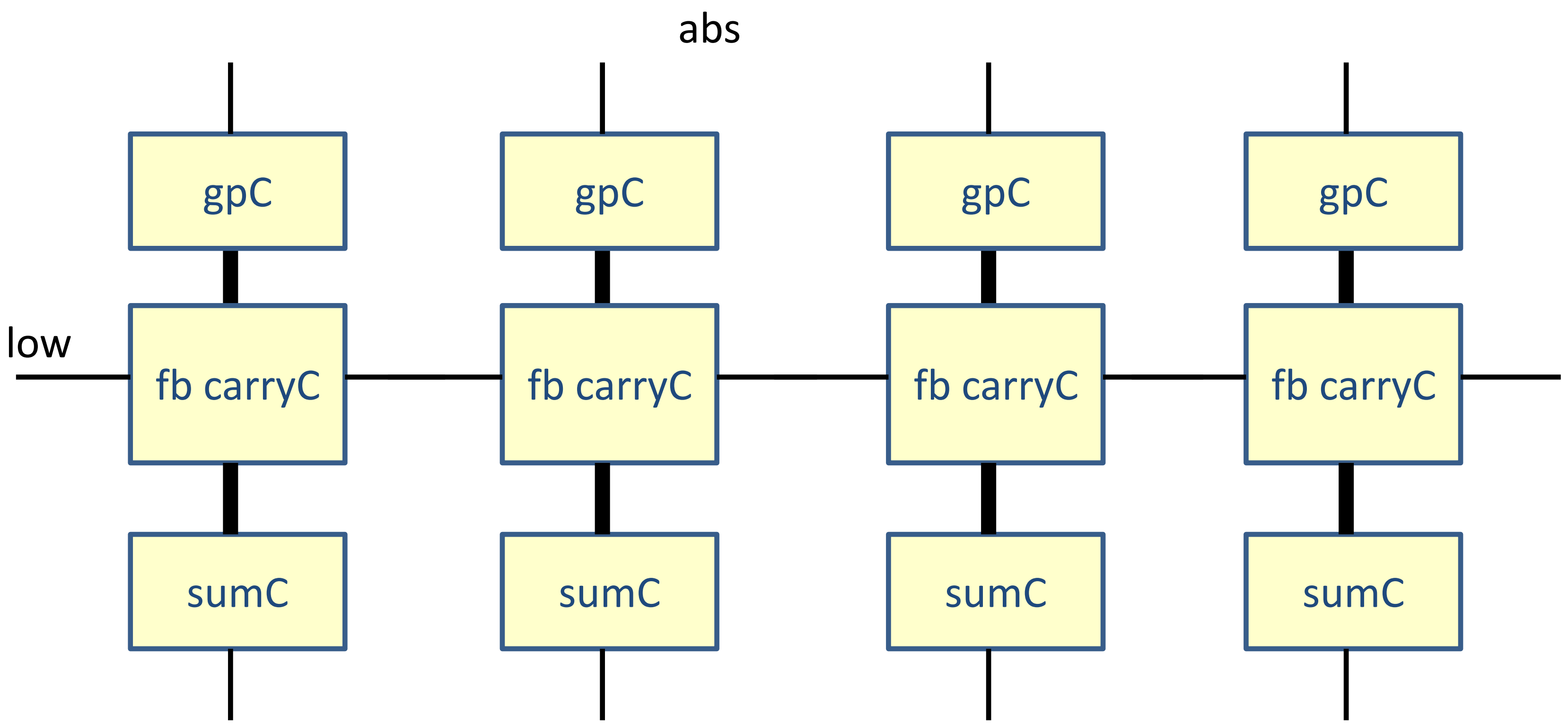
Carrys can be computed in parallel (see later)



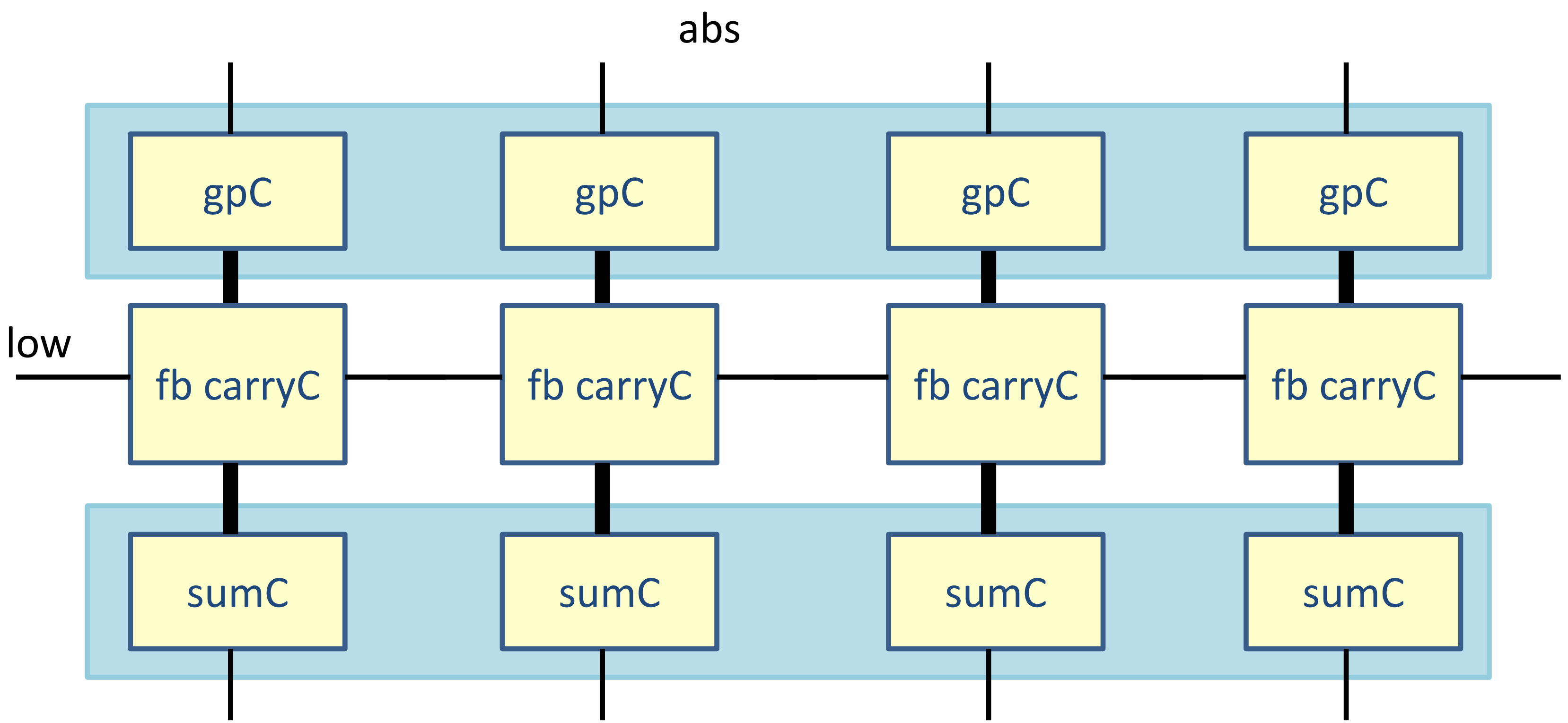
Can we rewrite this?

```
adder1 :: [(Bit, Bit)] -> ([Bit], Bit)
adder1 abs = row fullAdd1 (low,abs)
```

One row


abs
low
gpC
gpC
gpC
gpC
fb carryC
fb carryC
fb carryC
fb carryC
sumC
sumC
sumC
sumC

Same as two maps and a row



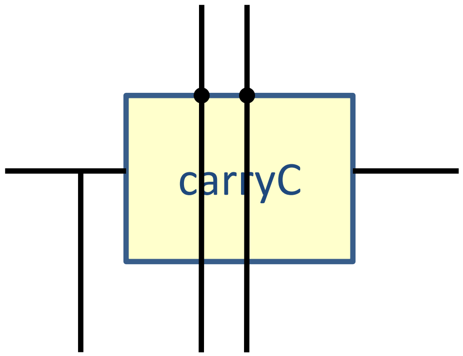
Two maps and a row

```
adder2 :: [(Bit, Bit)] -> ([Bit], Bit)
adder2 abs = (ss,cout)
  where
    gps         = map gpC abs
    (rs,cout) = row (fb carryC) (low,gps)
    ss          = map sumC rs
```

Isolate the carry chain

fb :: ((a,b) -> c) -> (a,b) -> ((a,b),c)
fb circ ab = (ab, circ ab)

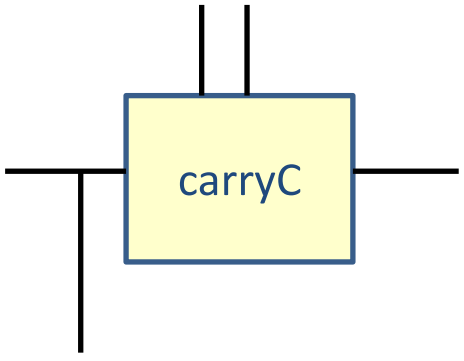
fb carryC



Isolate the carry chain

f1 :: ((a,b) -> c) -> (a,b) -> (a,c)
f1 circ (a,b) = (a, circ (a,b))

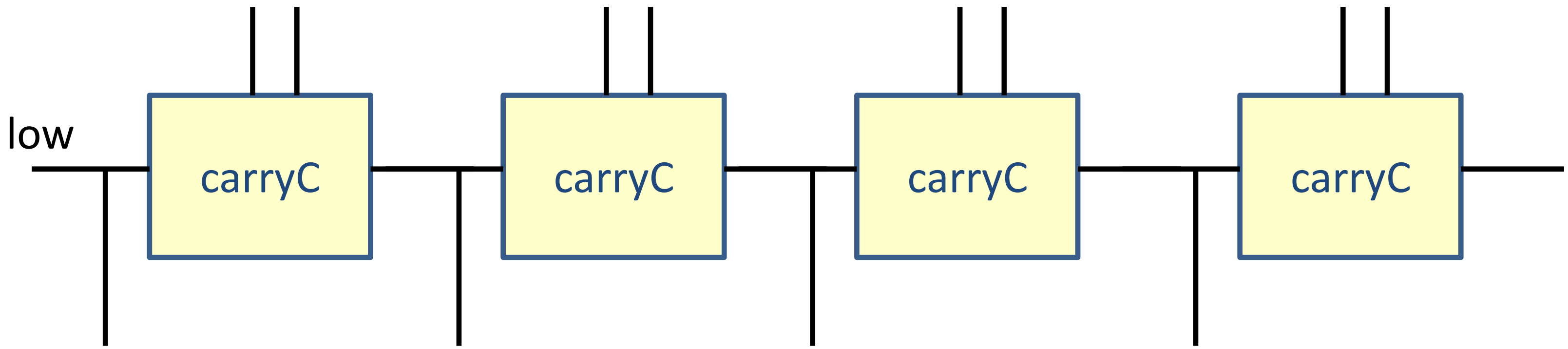
f1 carryC



slight reorg

```
adder3 :: [(Bit, Bit)] -> ([Bit], Bit)
adder3 abs = (ss,cout)
  where
    gps        = map gpC abs
    (cs,cout) = row (f1 carryC) (low,gps)
    rs         = zip cs gps
    ss         = map sumC rs
```

isolated the carry calculation

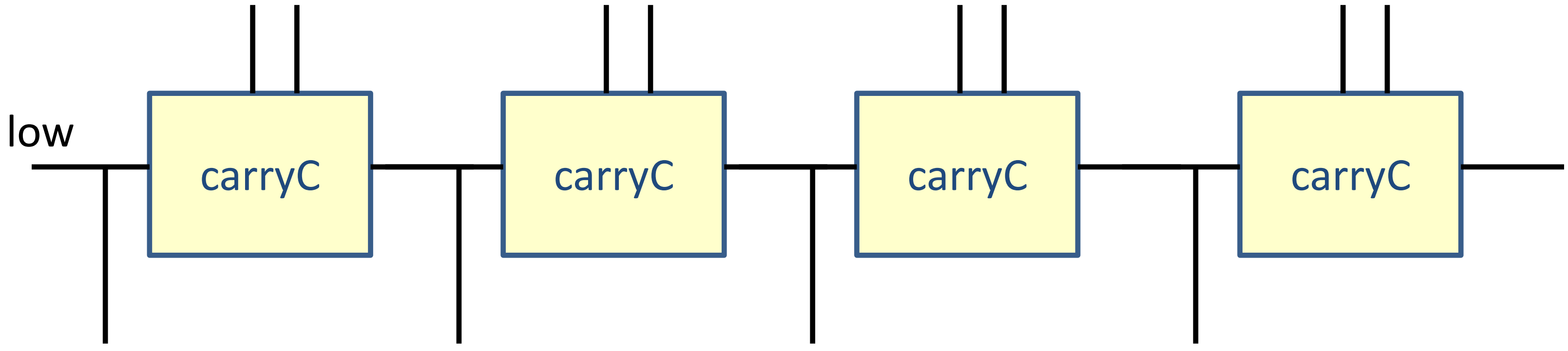


Remember

We can replace linear array by binary tree for associative operator

Similar game can be played for row

isolated the carry calculation



But carryC can't be associative:

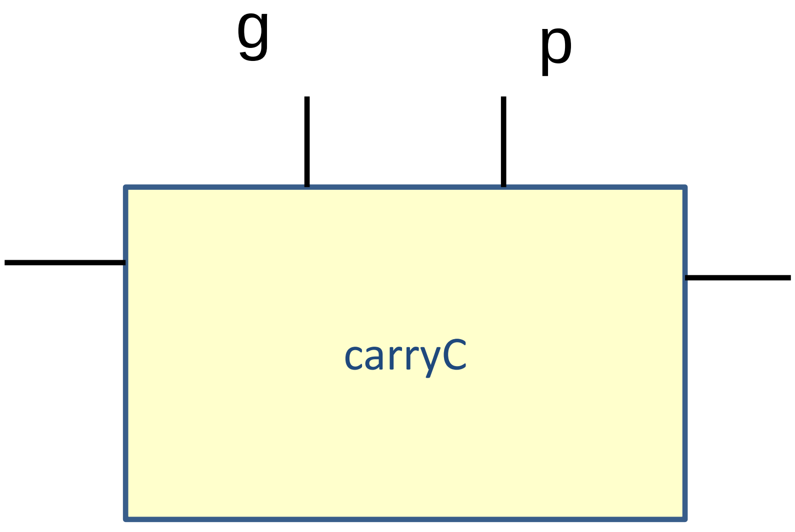
f1 carryC :: (Bit, (Bit, Bit)) -> (Bit, Bit)

(Need type (a,a) -> a)

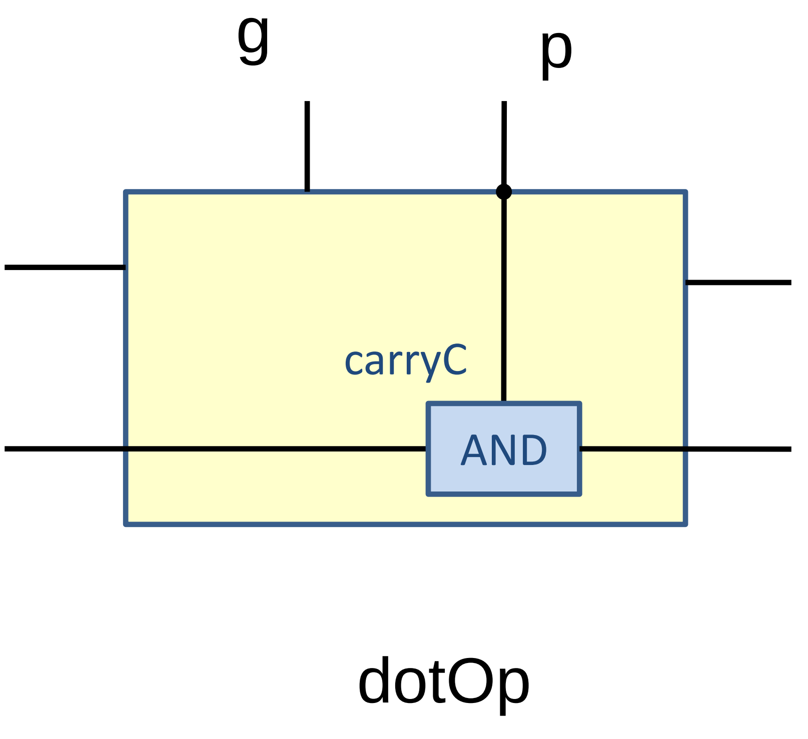
Brent and Kung’s insight

If we can find an associatve operator that still computes the same thing when placed in a row, then we will be able to do much better than a linear array!

enhance carryC



enhance carryC



enhance carryC

```
dotOp :: ((Bit,Bit), (Bit,Bit)) -> (Bit,Bit)
dotOp ((g1,p1), (g2,p2)) = (carryC (g1, (g2,p2)), p1 <&> p2)
```

enhance carryC

```
dotOp :: ((Bit,Bit), (Bit,Bit)) -> (Bit,Bit)
dotOp ((g1,p1), (g2,p2)) = (carryC (g1, (g2,p2)), p1 <&> p2)
```

Need also to compensate for this change so that the entire circuit retains its function

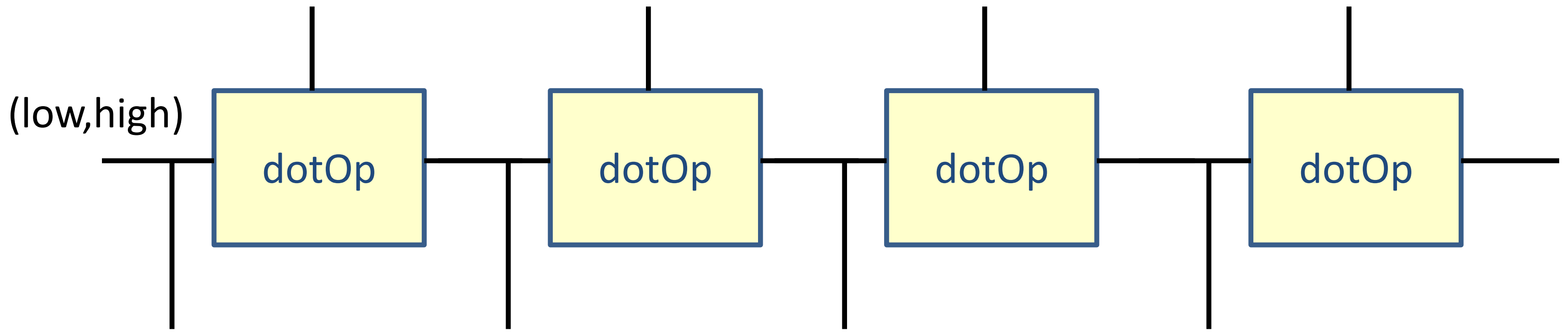
enhance carryC

```
adder4 :: [(Bit, Bit)] -> ([Bit], Bit)
adder4 abs = (ss,cout)
  where
    gps         = map gpC abs
    (cs,cout) = (row (f1 dotOp) ->- (map fst -|- fst)) ((low,high), gps)
    rs           = zip cs gps
    ss          = map sumC rs
```

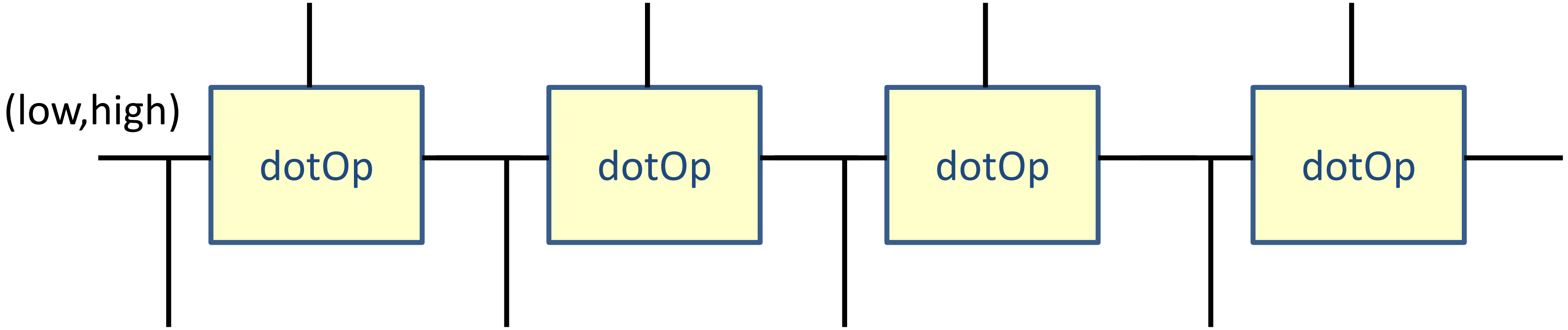
enhance carryC

```
adder4 :: [(Bit, Bit)] -> ([Bit], Bit)
adder4 abs = (ss,cout)
  where
    gps         = map gpC abs
    (cs,cout) = (row (f1 dotOp) ->- (map fst -|- fst)) ((low,high), gps)
    rs          = zip cs gps
    ss          = map sumC rs
```

New situation (associative op.)



New situation (associative op.)



Matches the famous Prefix Problem!

Prefix

Given inputs x1, x2, x3 … xn

Compute x1, x1*x2, x1*x2*x3, … , x1*x2*…*xn

Where * is an arbitrary associative (but not necessarily commutative) operator

Why interesting?

Microprocessors contain LOTS of parallel prefix circuits

- not only binary and FP adders
- address calculation
- priority encoding etc.

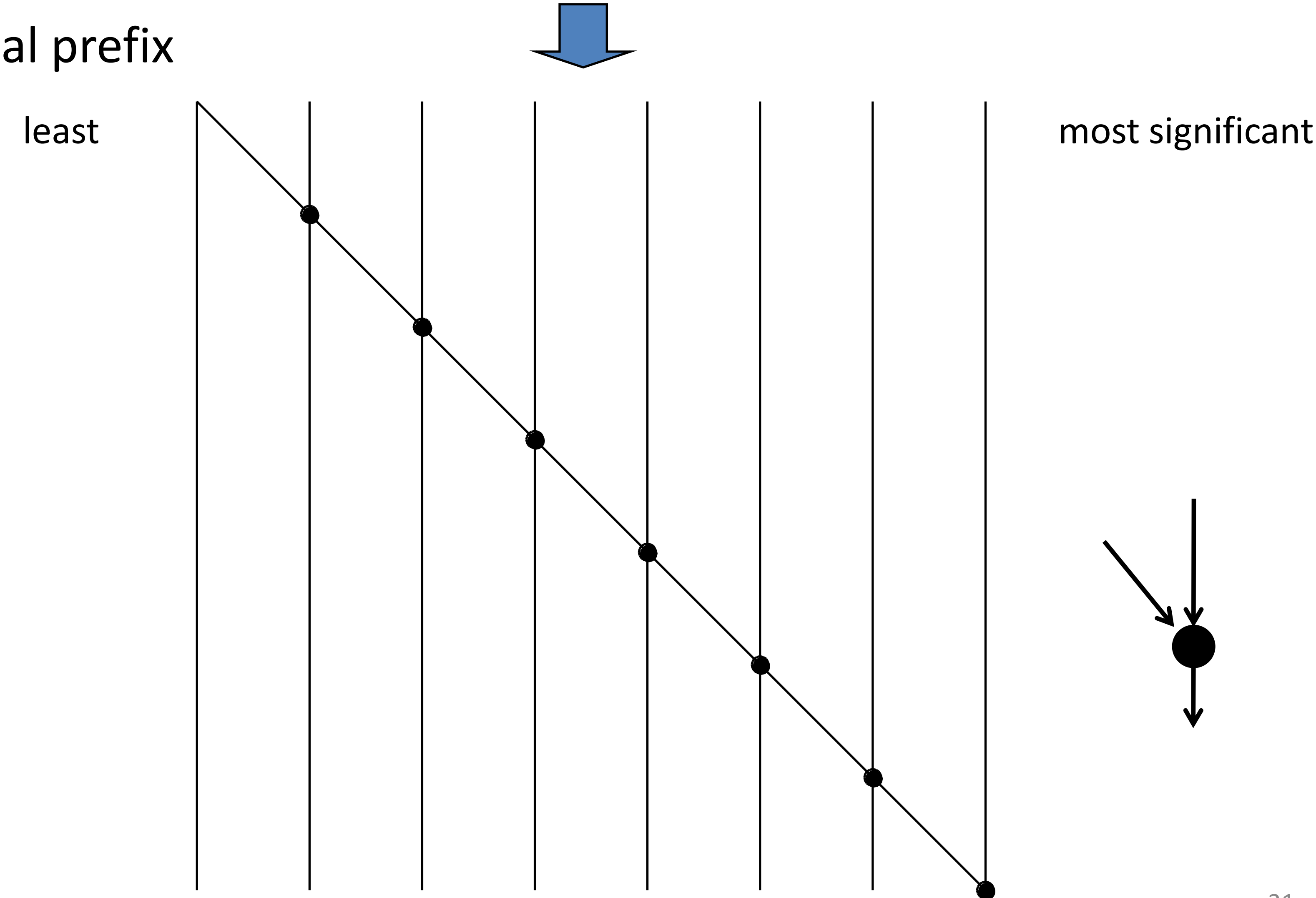
Overall performance depends on making them fast

But they should also have low power consumption...

Parallel prefix is a good example of a connection pattern for which it is interesting to do better synthesis

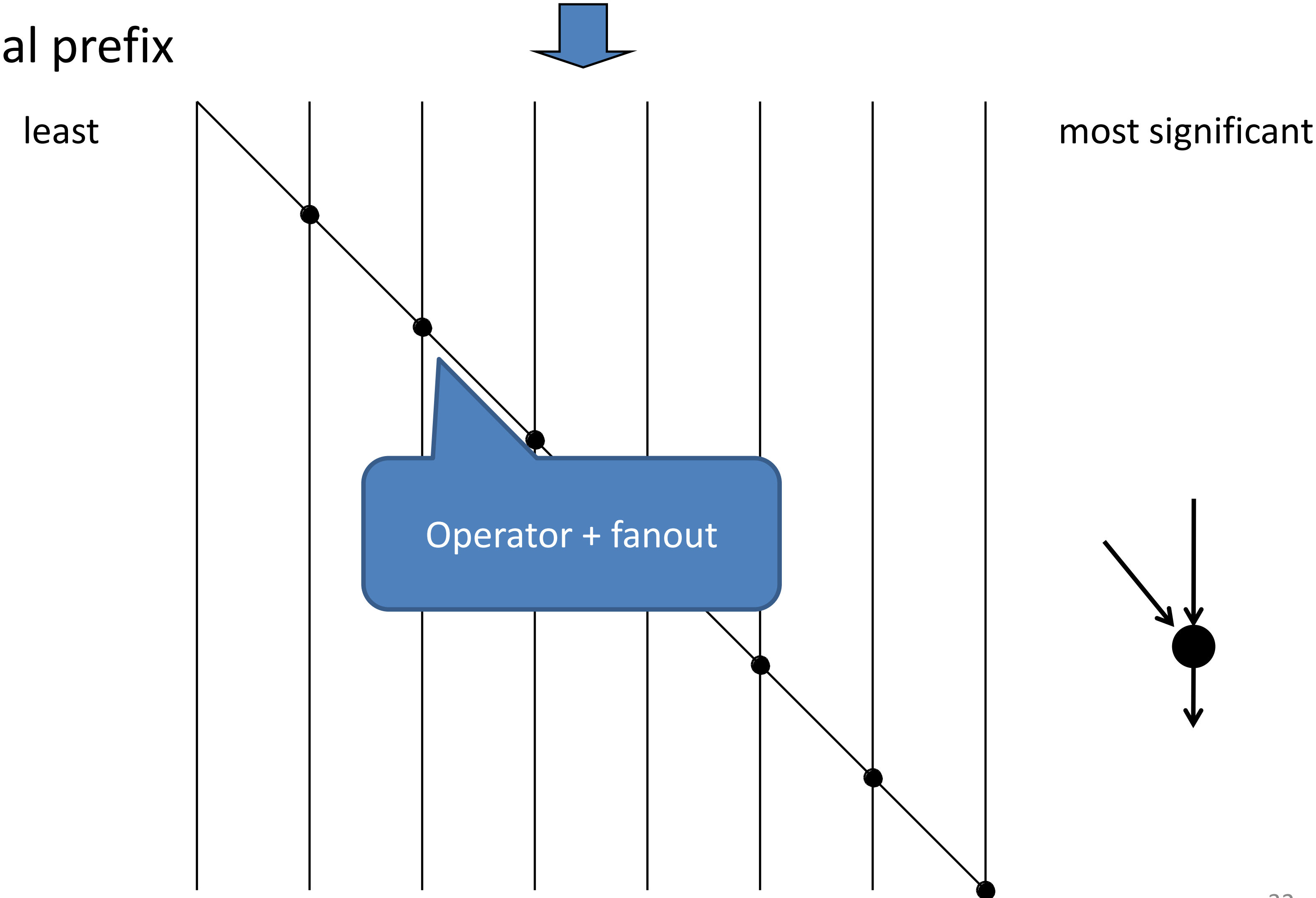
Visualizing prefix networks

Serial prefix

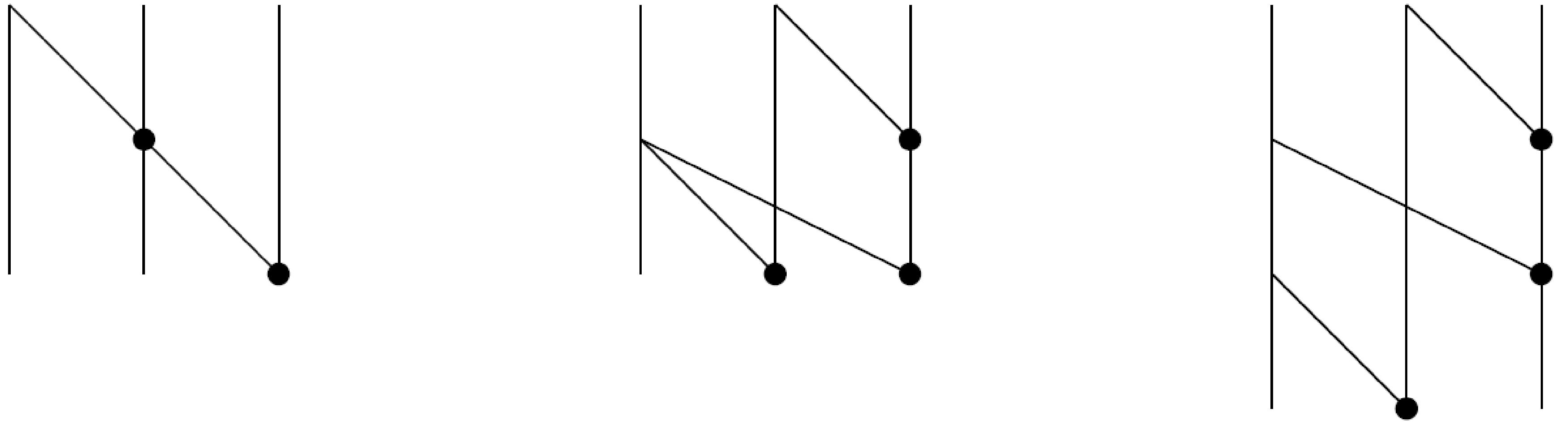


Visualizing prefix networks

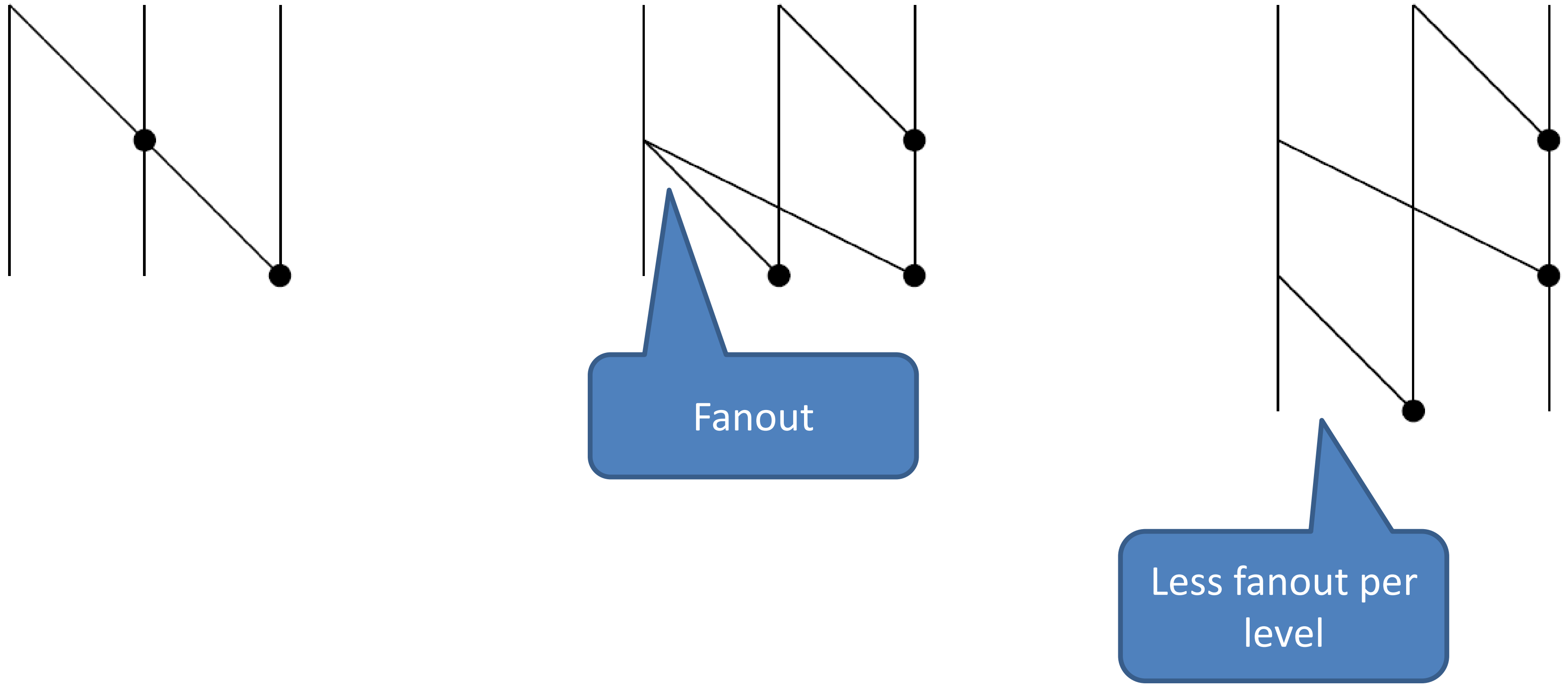
Serial prefix



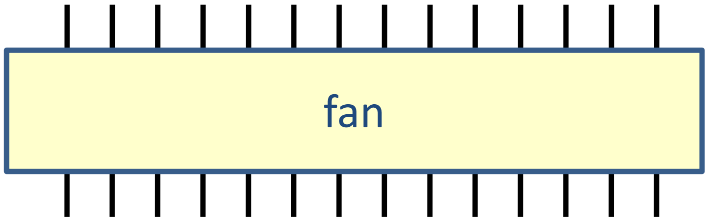
3 more

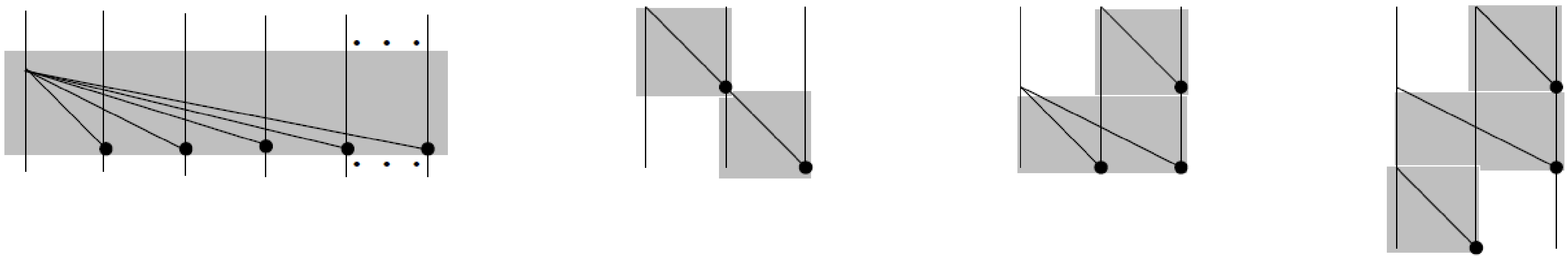


3 more



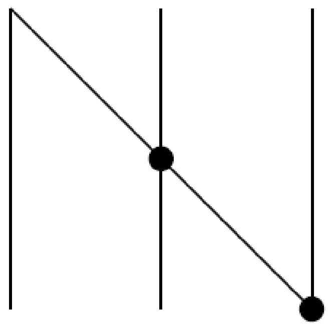
Basic building block: fan



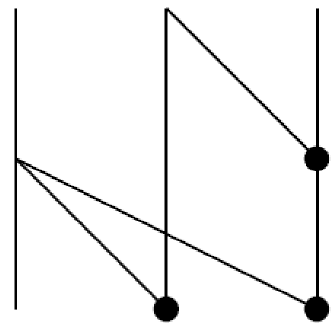


type Fan a = [a] -> [a]

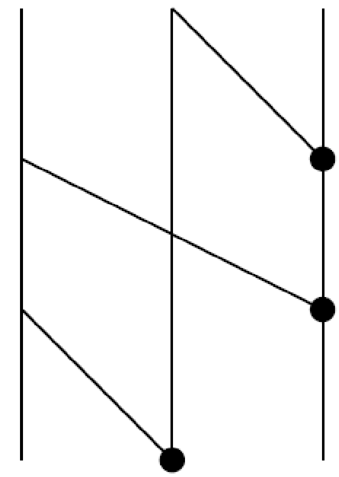
type PP a = Fan a -> [a] -> [a] -- Prefix network



```
ser3 :: PP a
ser3 f [a,b,c] = [a1,b2,c2]
  where
    [a1,b1] = f [a,b]
    [b2,c2] = f [b1,c]
```

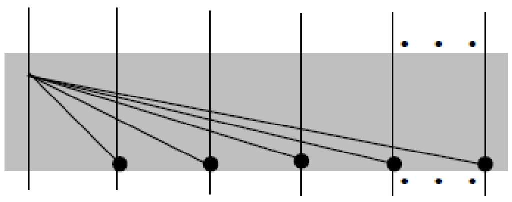


```
f31 :: PP a
f31 f [a,b,c] = [a1,b2,c2]
  where
    [b1,c1]      = f [b,c]
    [a1,b2,c2] = f [a,b1,c1]
```



```
f32 :: PP a
f32 f [a,b,c] = [a2,b2,c2]
  where
    [b1,c1] = f [b,c]
    [a1,c2] = f [a,c1]
    [a2,b2] = f [a1,b1]
```

Useful fan parameters



```
mkFan :: ((a,a) -> a) -> Fan a
mkFan op (i:is) = i:[op(i,k) | k <- is]

pplus :: Fan (Signal Int)  -- For making a prefix of additions
pplus = mkFan plus

delFan ::  Fan (Signal Int)  -- For delay estimation
delFan [i] = [i]
delFan is = replicate n (1 + maximum is)
  where
    n = length is
```

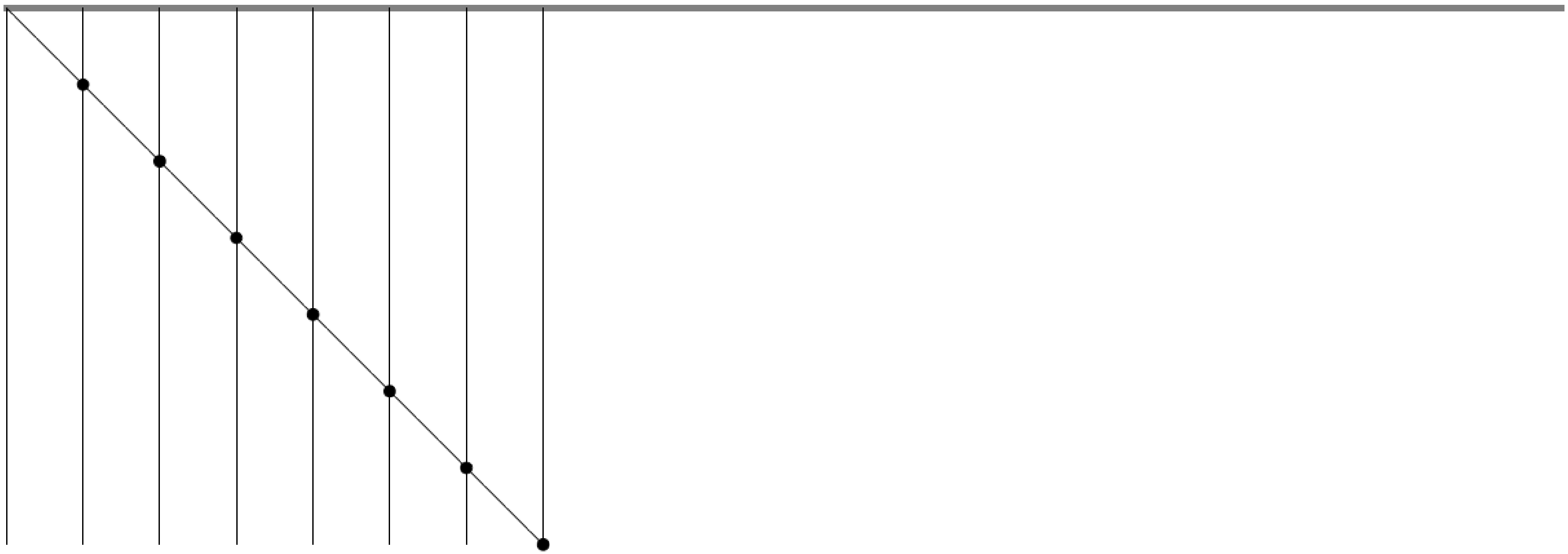
```
t3 = simulate (ser3 pplus) [1,2,3]
> t3
[1,3,6]
```

```
t3d = simulate (ser3 delFan) [0,0,0]
> t3d
[1,2,2]
```

serial prefix

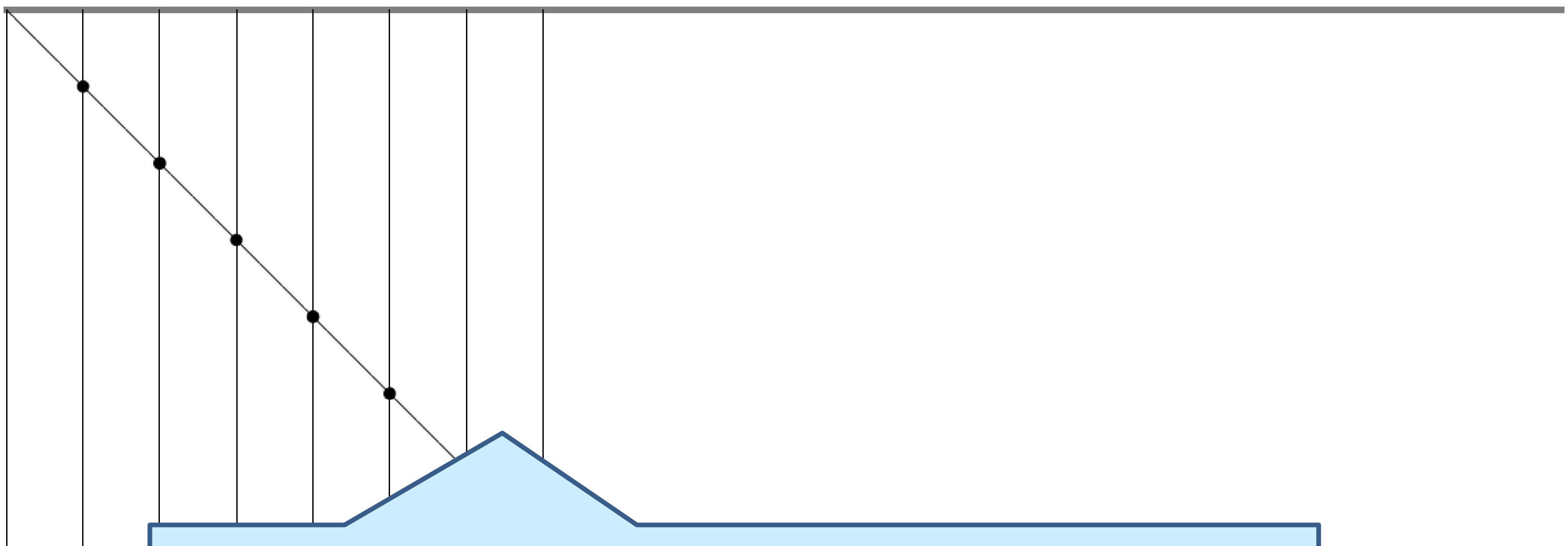
```
ser :: PP a
ser _  [a]        = [a]
ser f  (a:b:bs) = a1:cs
  where
    [a1,a2] = f [a,b]
    cs         = ser f (a2:bs)
```

```
> drawPP "ser" ser 8
```



8 lines, 7 stages, 7 operators, 2 maximum fanout.

```
> drawPP "ser" ser 8
```

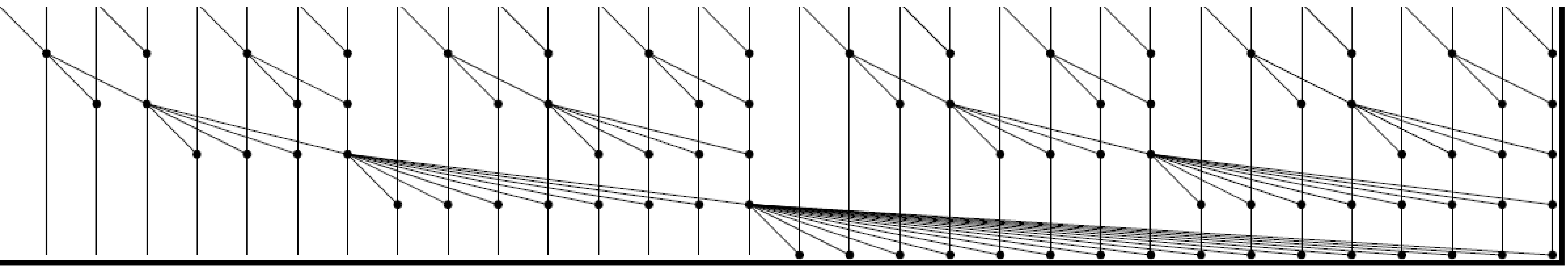


xfig file is produced (by symbolic evaluation) + hack

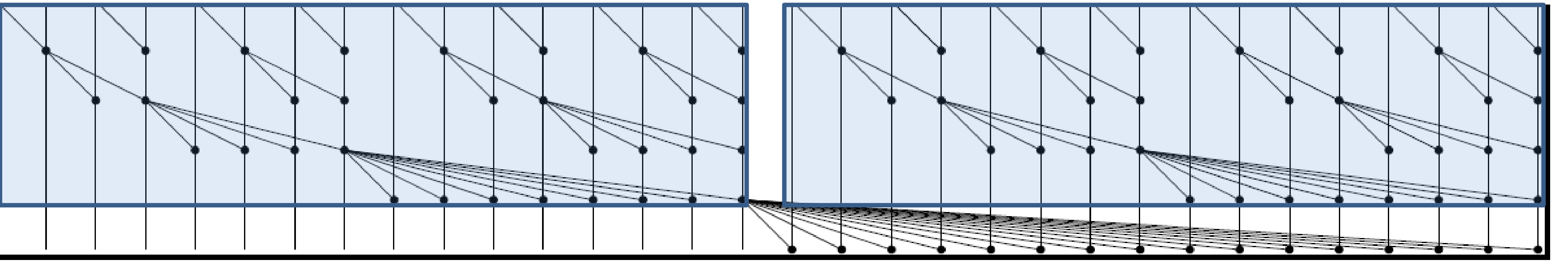
From xfig, pdf and many formats available

8 lin fanout.

Sklansky



Sklansky



32 lines   5 stages (= minimum)   80 operators

```
skl :: PP a
skl _ [a] = [a]
skl f as  = init los ++ ros'
  where
    (los,ros) = (skl f las, skl f ras)
    ros'        = f (last los : ros)
    (las,ras) = splitAt (cnd2 (length as)) as

cnd2 n = n - n `div` 2 -- Ceiling of n/2
```

back to the adder!

```
adder4 :: [(Bit, Bit)] -> ([Bit], Bit)
adder4 abs = (ss,cout)
  where
    gps         = map gpC abs
    (cs,cout) = (row (f1 dotOp) ->- (map fst -|- fst)) ((low,high), gps)
    rs           = zip cs gps
    ss          = map sumC rs
```

```
if  (cs,c)  = row (f1 circ) (e, as)
and e is an identity of circ

then

cs ++ [c]     =   e :  ser (mkFan circ)
```

back to the adder!

```
adder5 :: [(Bit, Bit)] -> ([Bit], Bit)
adder5 abs = (ss,cout)
  where
    gps       = map gpC abs
    (cs,cout) = (ser (mkFan dotOp) ->- unsnoc ->- (map fst -|- fst) ) gps
    rs        = zip (low:cs) gps
    ss        = map sumC rs



unsnoc as = (init as, last as)
```

slight optimisation (remove low)

```
adder6 :: [(Bit, Bit)] -> ([Bit], Bit)
adder6 abs = (ss,cout)
  where
    gps            = map gpC abs
    (cs,cout)      = (ser (mkFan dotOp) ->- unsnoc ->- (map fst -|- fst) ) gps
    ((_,p) : gps') = gps
    rs             = zip cs gps'
    ss             = p : map sumC rs
```

BUT now we can use any prefix network we fancy instead of ser

and there are lots to choose from!

Sklansky

```
adder7 :: [(Bit, Bit)] -> ([Bit], Bit)
adder7 abs = (ss,cout)
  where
    gps             = map gpC abs
    (cs,cout)       = (skl (mkFan dotOp) ->- unsnoc ->- (map fst -|- fst) ) gps
    ((_,p) : gps') = gps
    rs              = zip cs gps'
    ss              = p : map sumC rs
```

Sklansky

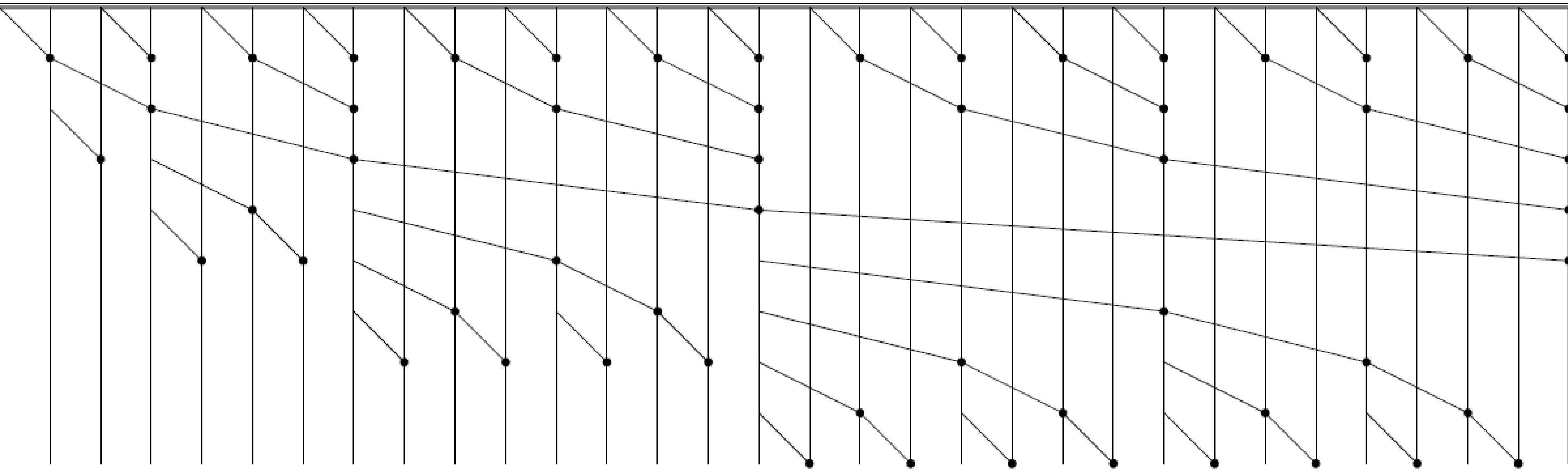
```
adder7 :: [(Bit, Bit)] -> ([Bit], Bit)
adder7 abs = (ss,cout)
  where
    gps            = map gpC abs
    (cs,cout)      = (skl (mkFan dotOp) ->- unsnoc ->- (map fst -|- fst) ) gps
    ((_,p) : gps') = gps
    rs             = zip cs gps'
    ss             = p : map sumC rs
```

Size (= power consumption) and performance completely dominated by the prefix network

Could (and should) parameterise on the pattern

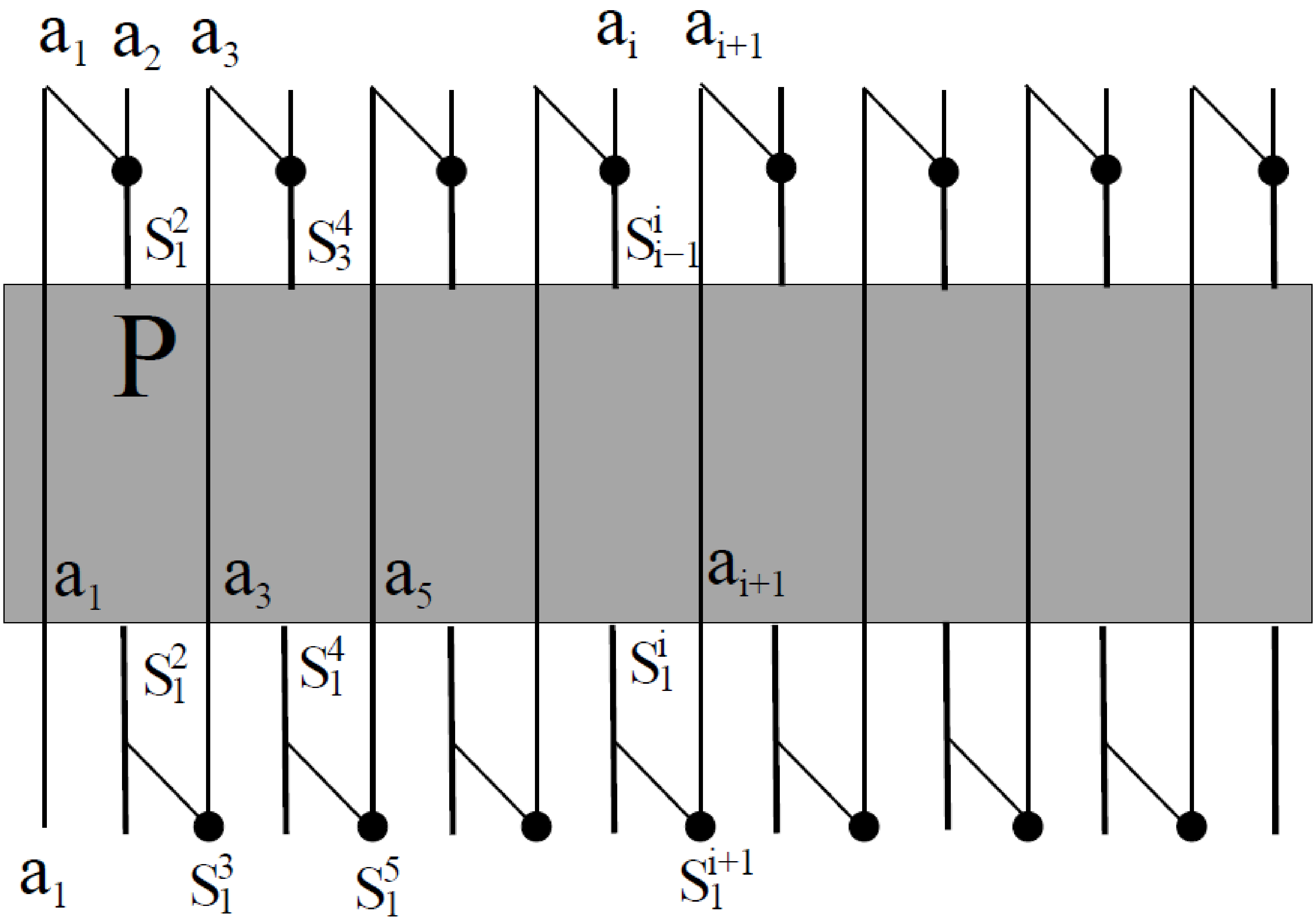
Brent Kung network

> drawPP "bK" bKung 32



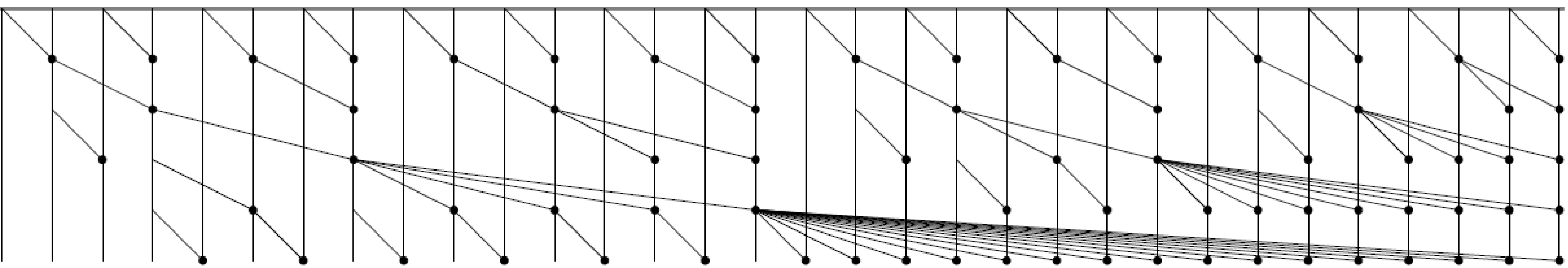
32 lines, 9 stages, 57 operators, 2 maximum fanout.

Brent Kung: Recursive structure



Ladner Fischer min. depth

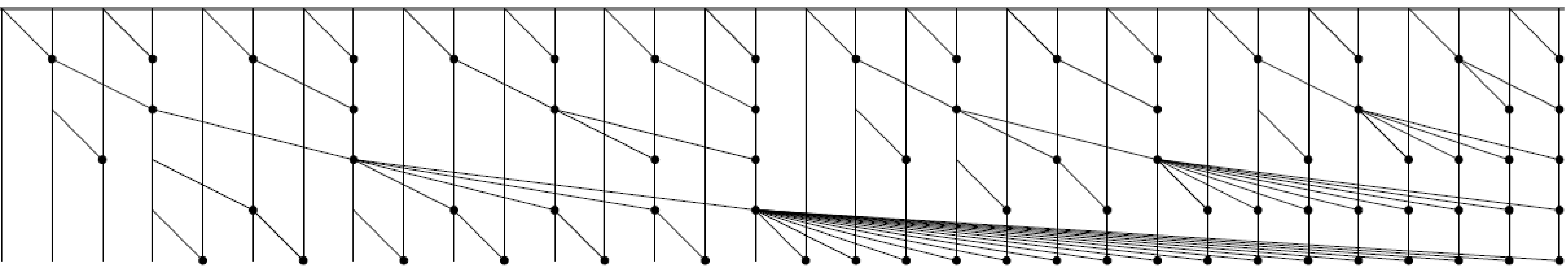
```
> drawPP "LF0" (ladF 0) 32
```



32 lines, 5 stages, 74 operators, 17 maximum fanout.

Ladner Fischer min. depth

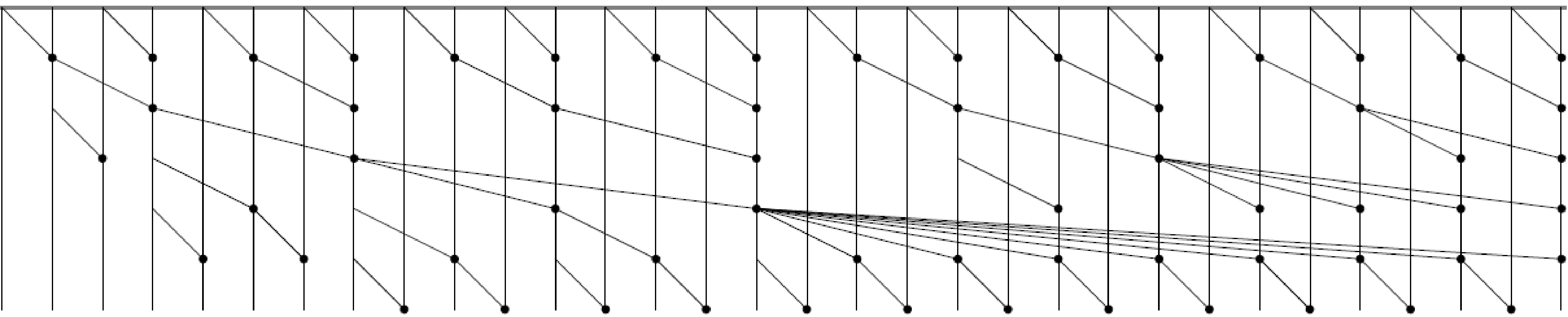
> drawPP "LF0" (ladF 0) 32



32 lines, 5 stages, 74 op 17 maximum fanout.

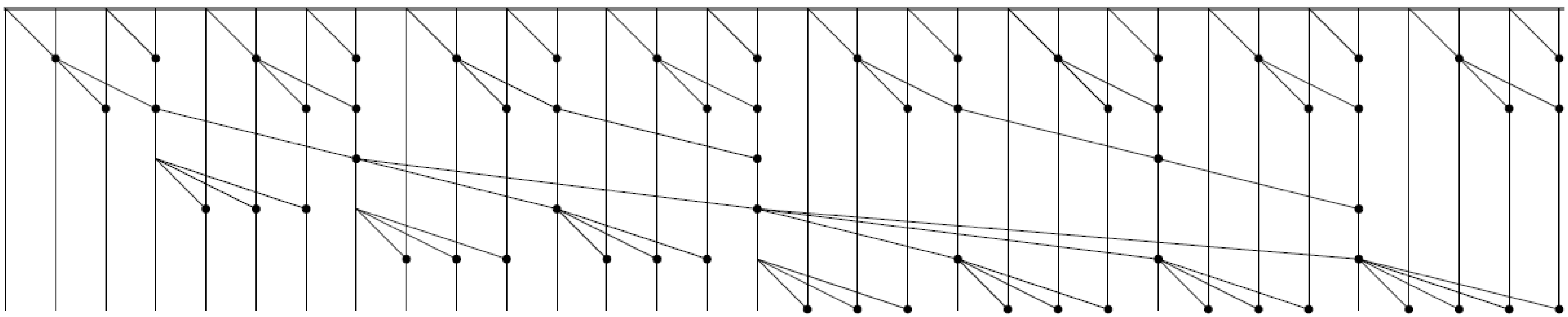
Beware. Many papers and books are wrong about LF (and think it is the same as Sklansky). It is not!

LF min depth + 1



32 lines, 6 stages, 62 operators, 9 maximum fanout.

and more



32 lines, 6 stages, 63 operators, 5 maximum fanout.

Problem

Find a sweet spot LAGOM

Not too big

Not too deep

Not too much fanout

Questions?