Finding good prefix networks using Haskell

Mary Sheeran (Chalmers)

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

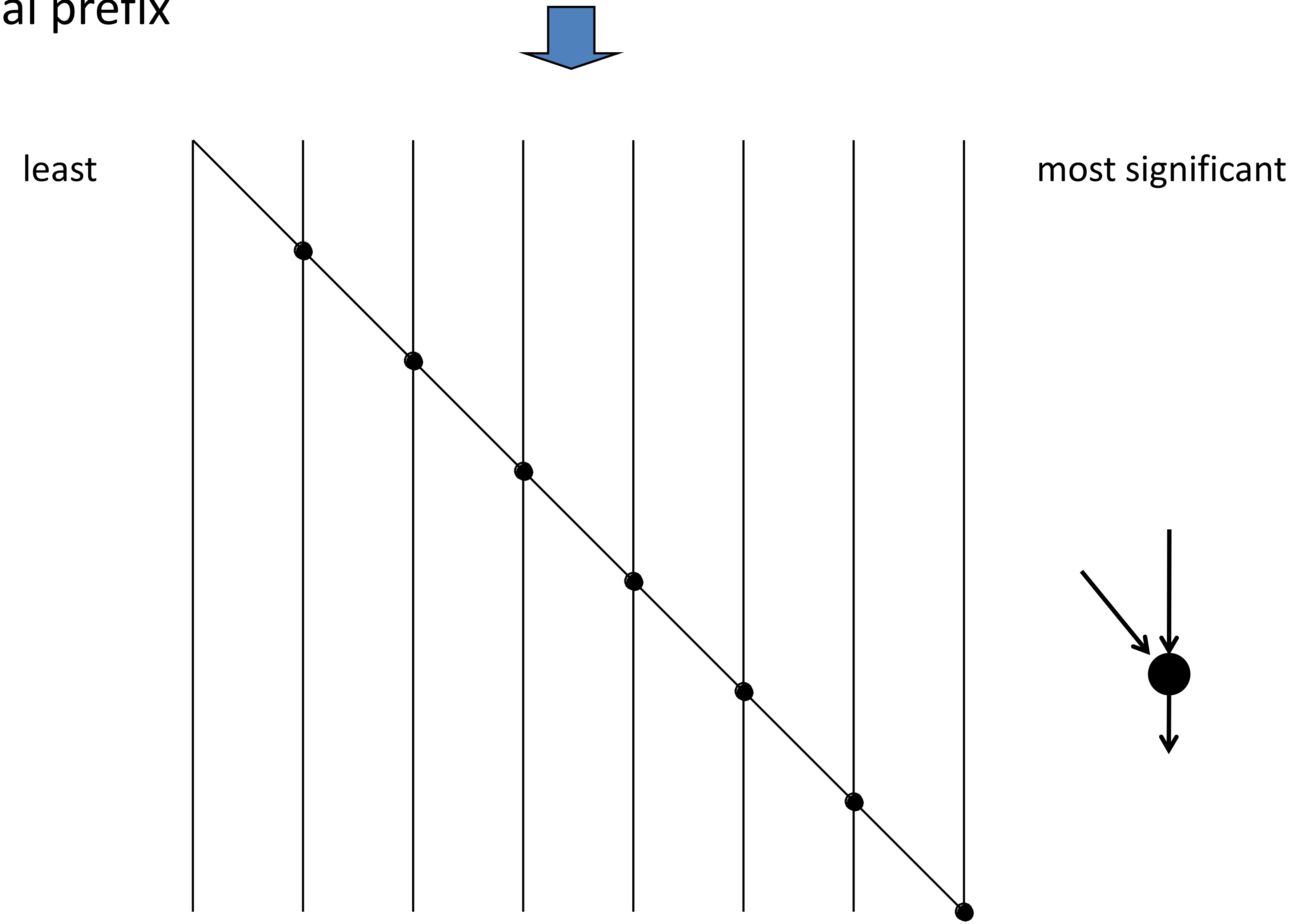
- not only binary and FP adders
- address calculation
- priority encoding etc.

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

Serial prefix



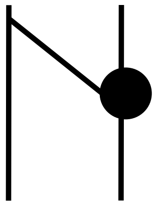
Might expect

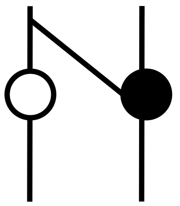
```
serr _  [a]          = [a]
serr op (a:b:bs) = a:cs
  where
    c  = op(a,b)
    cs = serr op (c:bs)
```

```
*Main> simulate (serr plus) [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

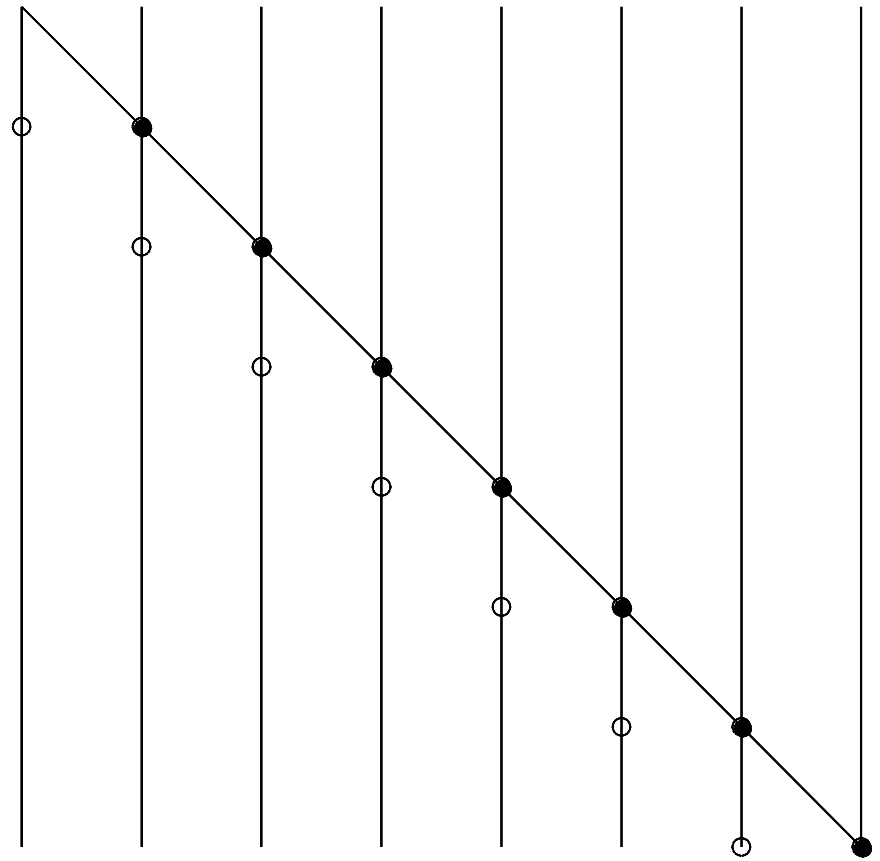
But I am going to prefer building blocks that are themselves pp networks

```
type NW a = [a] -> [a]

type PN    =  forall a. NW a -> NW a



bser _ []     = []
bser _ [a]    = [a]
bser op as    = ser bop as
  where
    bop [a,b] = op[c]++[d]
      where [c,d] = op [a,b]
```

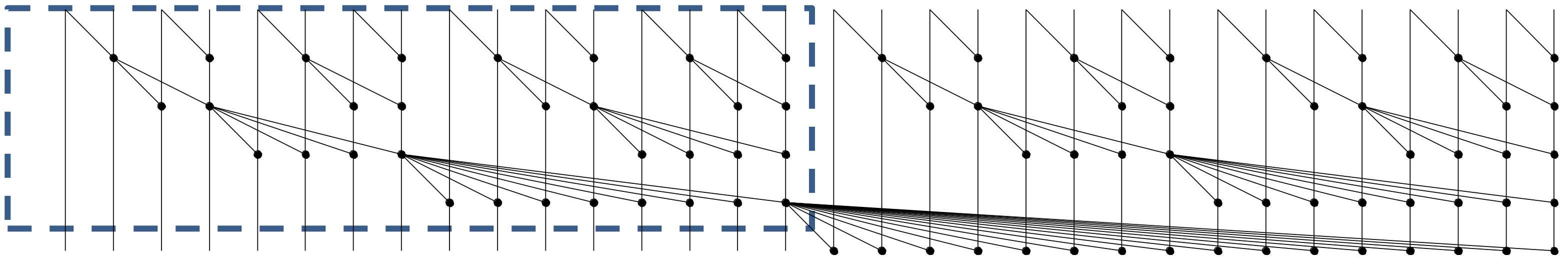




When the operator works on a singleton list, it is a buffer (drawn as a white circle)

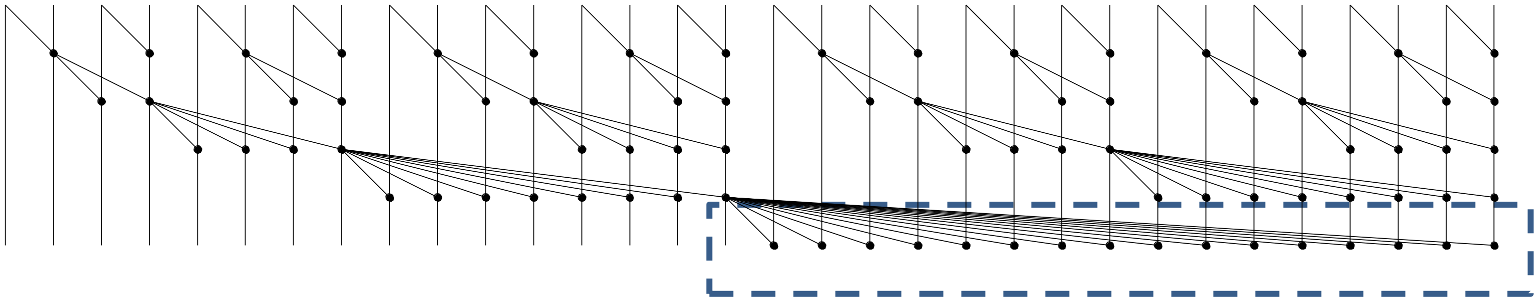


Sklansky



32 inputs, depth 5, 80 operators

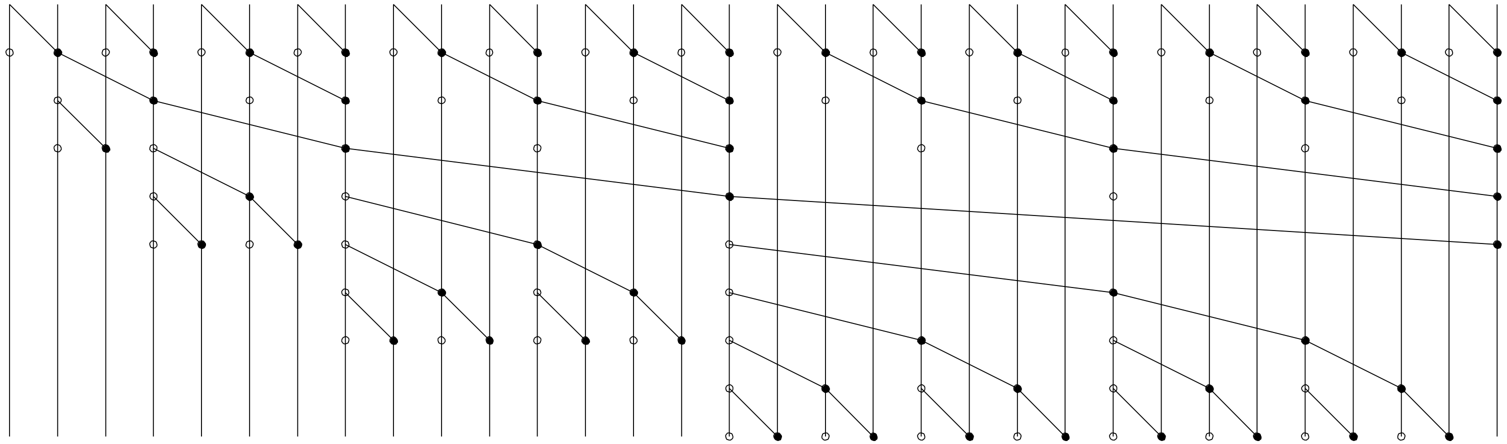
Sklansky



32 inputs, depth 5, 80 operators

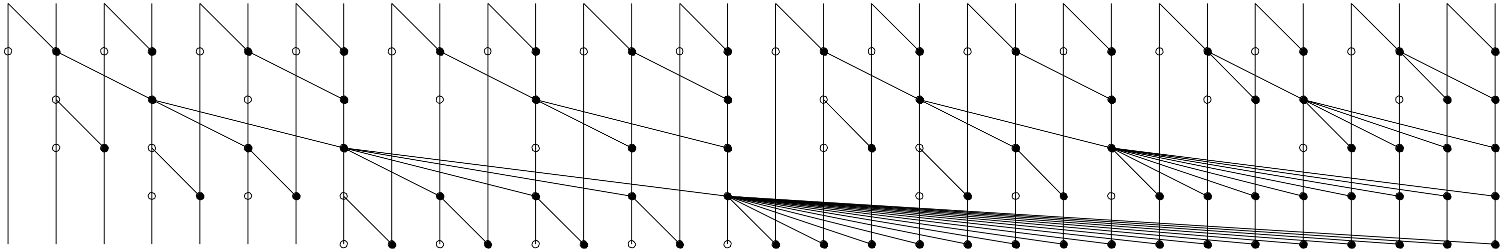
```
skl :: PN
skl _ [a] = [a]
skl op as = init los ++ ros'
  where
    (los,ros) = (skl op las, skl op ras)
    ros'      = fan op (last los : ros)
    (las,ras) = halveList as

plusop[a,b] = [a, a+b]

*Main> (skl plusop) [1..10]
[1,3,6,10,15,21,28,36,45,55]
```

Brent Kung



fewer ops, at cost of being deeper.   Fanout only 2

Ladner Fischer



NOT the same as Sklansky; many books and papers are wrong about this

Question

How do we design fast low power prefix networks?

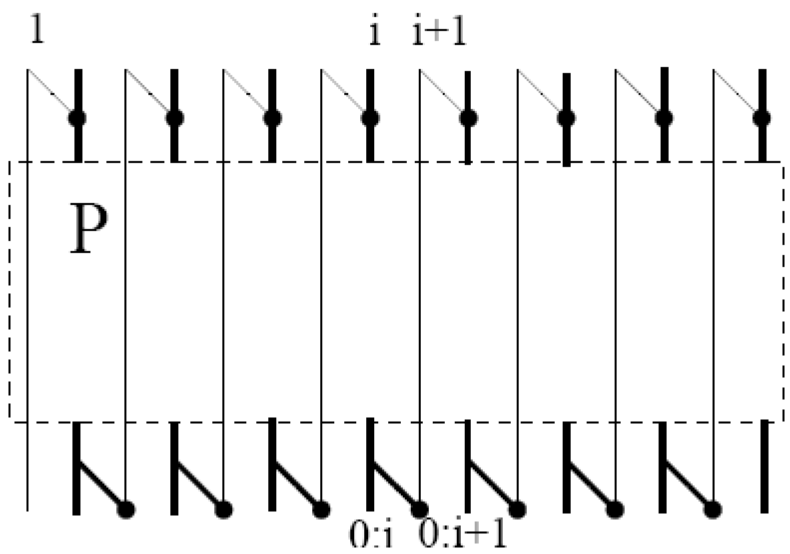
Answer

Generalise the above recursive constructions

Use dynamic programming to search for a good solution

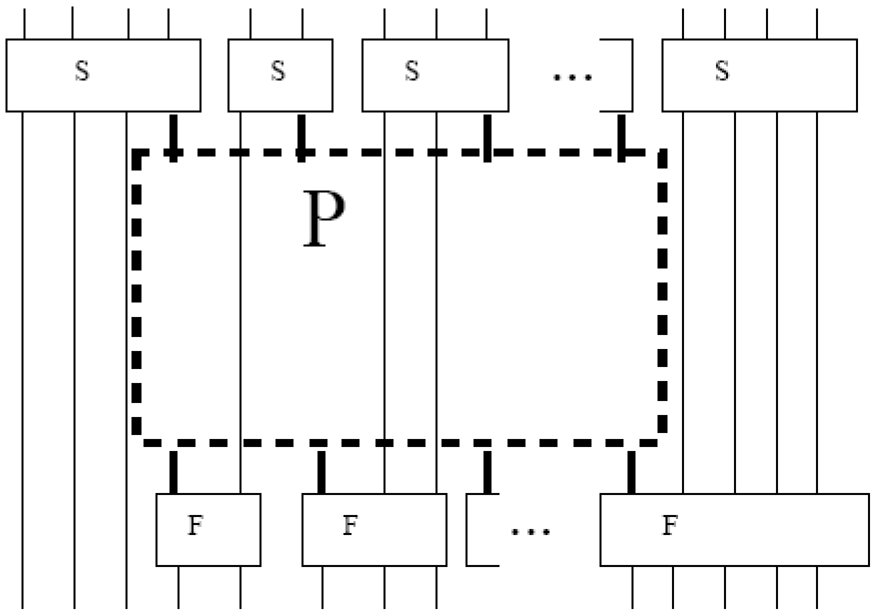
Use Wired to increase accuracy of power and delay estimations

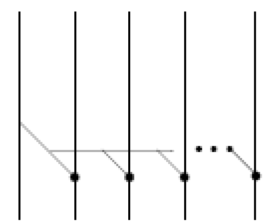
BK recursive pattern



P is another half size network operating on only the thick wires

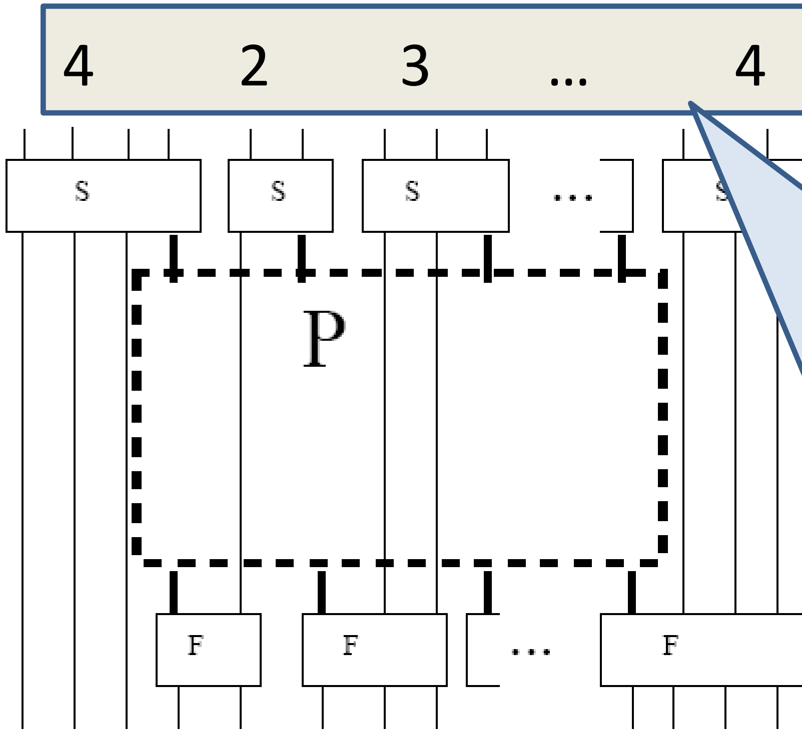
BK recursive pattern generalised





F

Each S is a serial network like that shown earlier

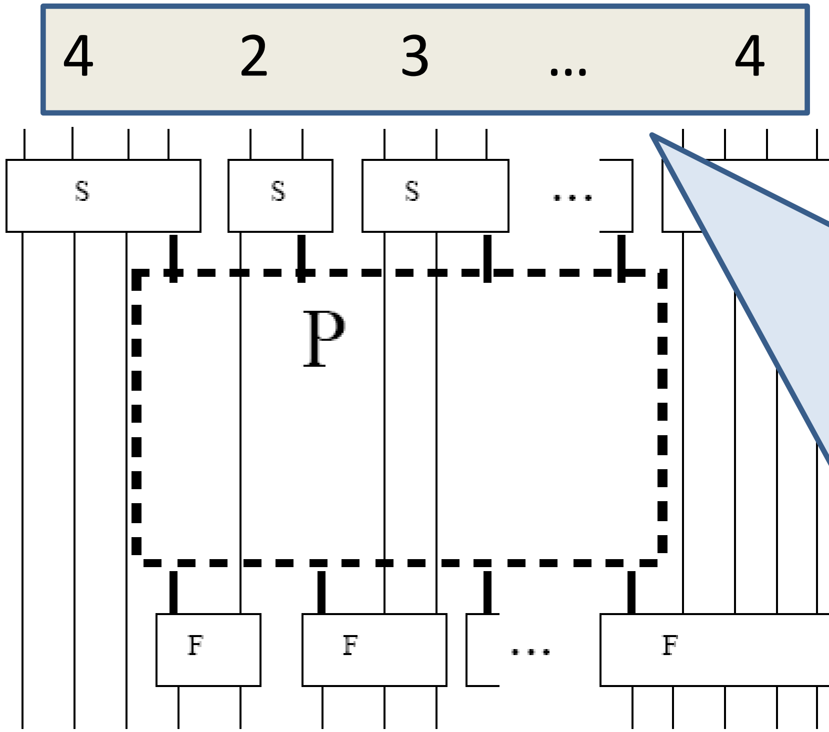


This sequence of numbers determines how the outer ”layer” looks

```
wrp ds p comp as = concat rs
   where
     bs      = [bser comp i | i <- splits ds as]
     ps      = p comp $ map last (init bs)
     (q:qs) = mapInit init bs
     rs      = q:[bfan comp (t:u) | (t,u) <- zip ps qs]
```

```
twos 0 = [0]
twos 1 = [1]
twos n = 2:twos (n-2)
```

```
bk _ [a] = [a]
bk comp as = wrp (twos (length as)) bk comp as
```



So just look at all possibilities for this sequence

and for each one find the best possibility for the smaller P

Then pick best overall!

Dynamic programming

Search!

need a measure function (e.g. number of operators)

Need the idea of a context into which a network (or even just wires) should fit

```
type Context = ([Int],Int)

data PPN = Pat PN | Fail

delF :: NW Int
delF [a] = [a+1]
delF [a,b] = [m,m+1]
  where m = max a b

try :: PN -> Context -> PPN
try  p (ds,w)
  = if and [o <= w | o <-  p delF ds] then Pat p else Fail
```

Need a variant of wrp that can fail, and that makes the ”crossing over” wires explicit (because they might not fit either)

```
wrp2 :: [Int] -> PPN -> PPN -> PPN
wrp2 ds (Pat wires) (Pat p) = Pat r
  where
    r comp as = concat rs
      where
        bs       = [bser comp i | i <- splits ds as]
        qs       = wires comp $ concat (mapInit init bs)
        ps       = p comp $ map last (init bs)
        (q:qs') = splits (mapInit sub1 ds) qs
        rs       = q:[bfan comp (t:u) | (t,u) <- zip ps qs']
wrp2 _ _ _       = Fail
```

```
parpre f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                        [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    pref
      wh
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                     [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

f1 is the measure function being optimised for

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix
      where
        pm (
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                      [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

g is max width of small F networks. Controls fanout.

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w)
        pm (is,w) |                                    Fail
        pm (is,w) =
                                                    - topds g h lis]
          where
            h = maxc
            lis = len
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

use memoisation to avoid expensive recomputation

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) |                                   ail
        pm (is,w) =
                                                        topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

base case: single wire

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn i
                      [wrpC d                          h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2                          s,w-1))
              where
                bs = [bser de
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

Fail if it is simply impossible to fit a prefix network in the available depth

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                         [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp                              (ns,w-1))
              where
                bs = [bser
                ns = map la
                ts = concat
```

Generate candidate sequences

Here is where the cleverness is

I keep them almost sorted

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                         [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2 d                            ,w-1))
              where
                bs = [bser de
                ns = map last
                ts = concat (r
```

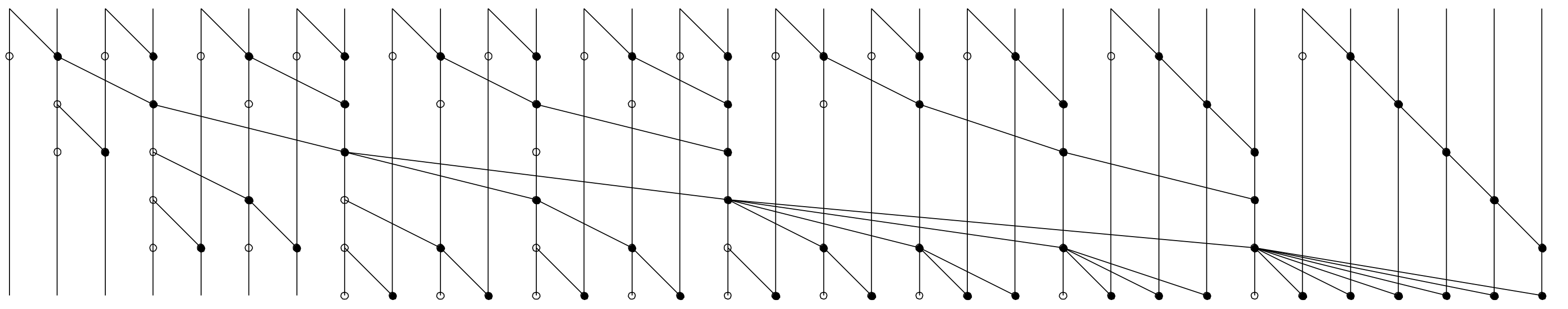
For each candidate sequence:
Build the resulting network
(where call of (prefix f) gives the
best network for the recursive call
inside)

Figures out the contexts for the wires and the call of p in a call of wrp2

```
wso f1 g ctx =                             prefix f1 ctx)
  where
    prefix f =
      where
        pm ([i
        pm (is                                 is = Fail
        pm (is                                l1)
                                             | ds <- topds g h lis]
          where
            h =       d(is,w)
            lis = length is
            wrpC ds p = wrp2 ds (trywire (ts,w-1)) (p (ns,w-1))
              where
                bs = [bser delF i | i <- splits ds is]
                ns = map last (init bs)
                ts = concat (mapInit init bs)
```

```
wso f1 g ctx = getans (error "no fit")  (prefix f1 ctx)
  where
    prefix f = memo pm
      where
        pm ([i],w) = trywire ([i],w)
        pm (is,w) | 2^maxd(is,w) < length is = Fail
        pm (is,w) = ((bestOn is f).dropFail)
                         [wrpC ds (prefix f) | ds <- topds g h lis]
          where
            h = maxd(is,w)
            lis = length is
            wrpC ds p = wrp2                            ns,w-1))
              where
                bs = [bser c
                ns = map las
                ts = concat (
```

Finally, pick the best among all these candidates



Result when minimising number of ops, depth 6, 33 inputs, fanout 7

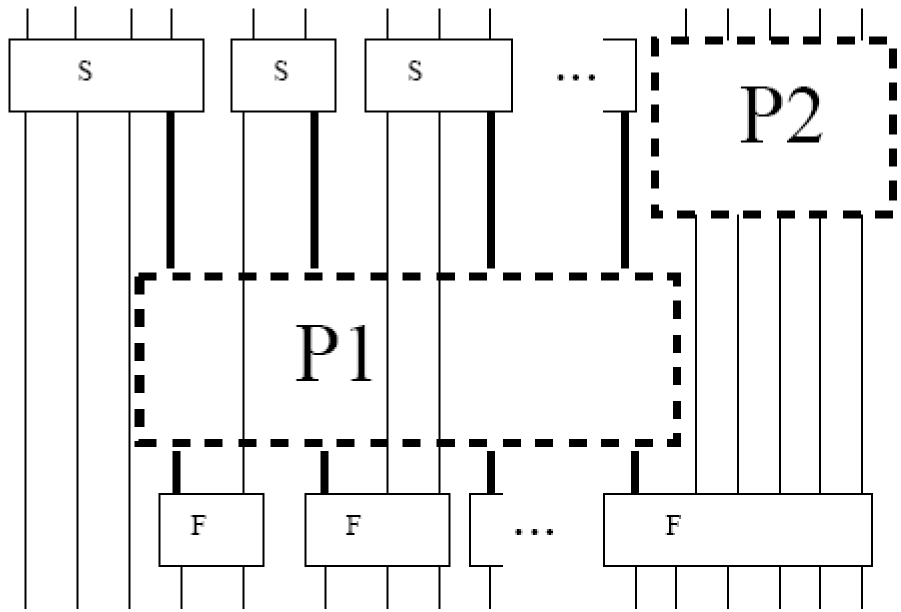
This network is Depth Size Optimal (DSO)

depth + number of ops = 2(number of inputs)-2
(known to be smallest possible no. ops for given depth, inputs)

6 + 58 = 2*33 – 2

BUT we need to move away from DSO networks to get shallow networks with more than 33 inputs

A further generalisation



Result

When minimising no. of ops: gives same as Ladner Fischer for 2^n inputs, depth n,

considerably fewer ops and lower fanout elsewhere (non power of 2 or not min. depth)

Promising power and speed when netlists given to Design Compiler

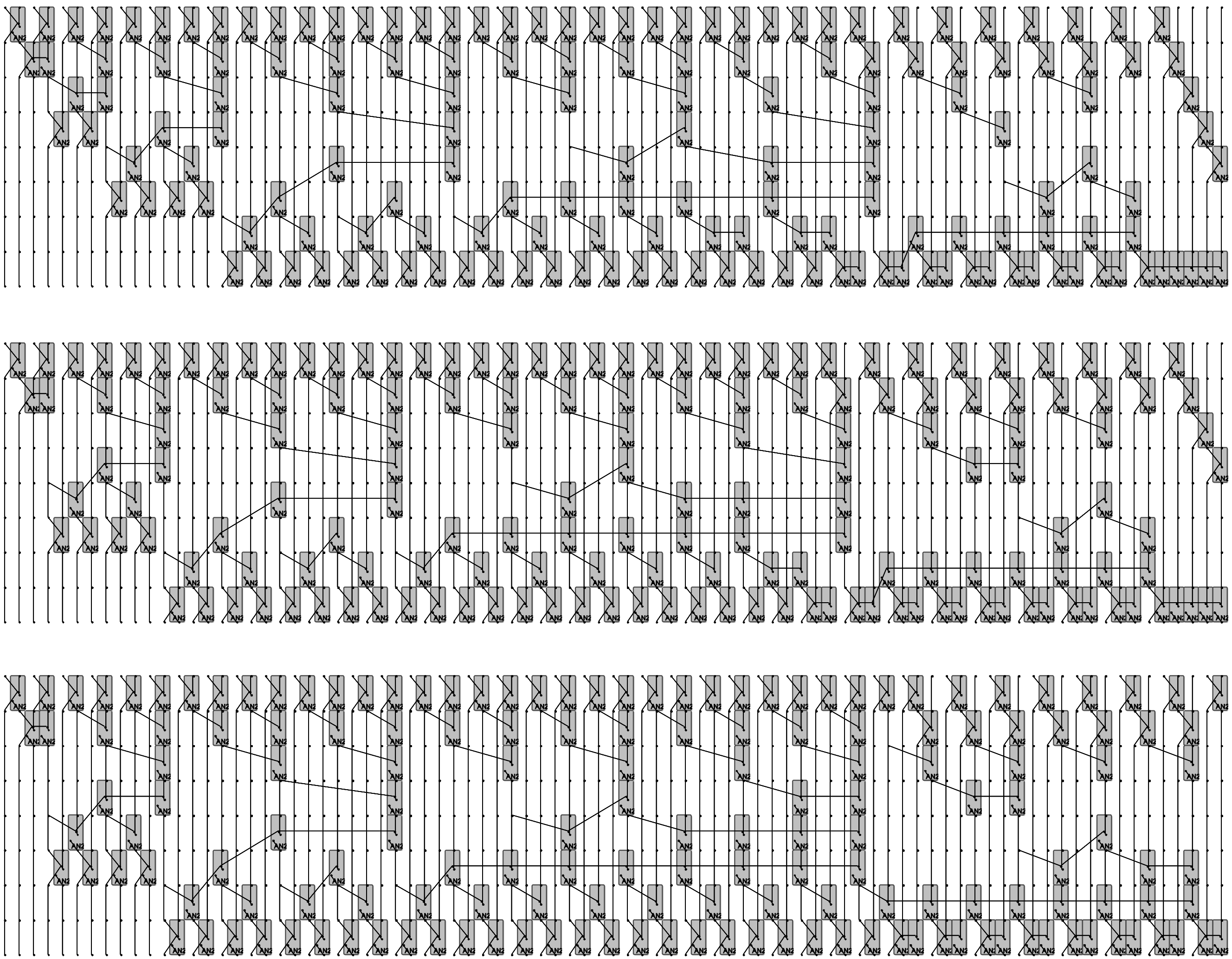
Result (more real)

Use Wired, a system for low level wire-aware hardware design developed by Emil Axelsson at Chalmers

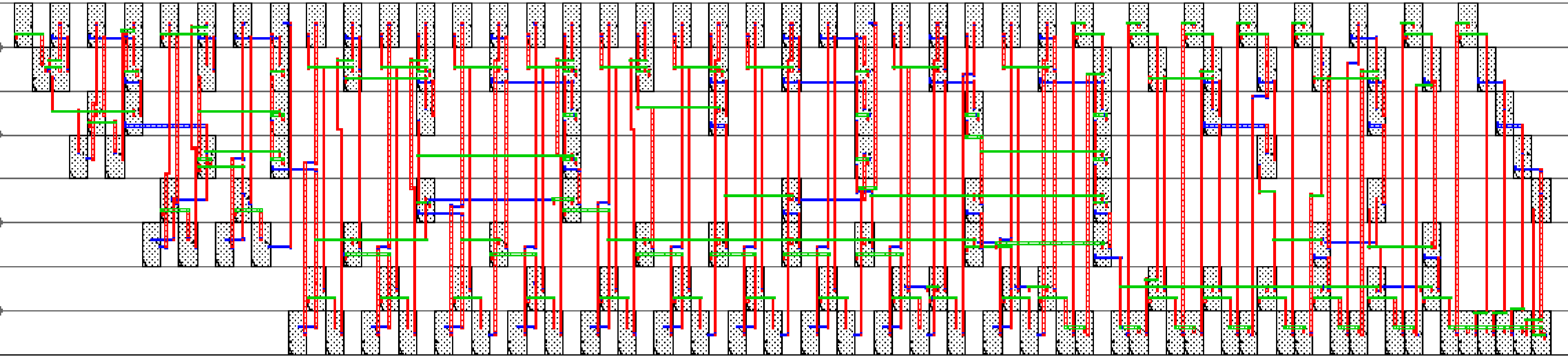
To link to Wired, need slightly fancier context
since physical position is important

Can minimise for (accurately estimated) speed
in P1 and for power in P2   (two measure functions)

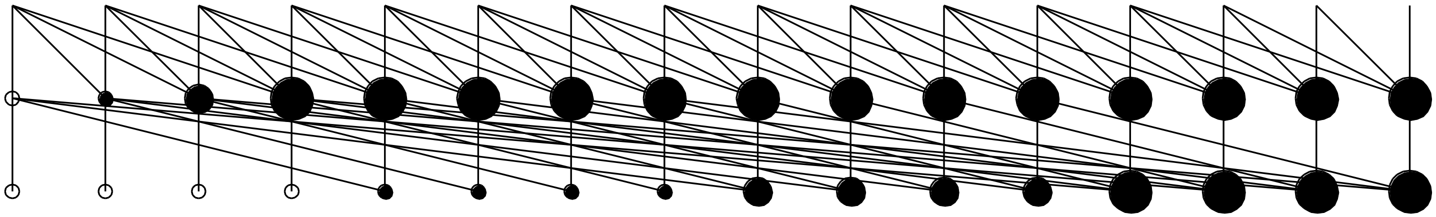
Link to Wired allows more accurate estimates. Can then explore design space



Can also export to Cadence SoC Encounter



Need to do more to make realistic circuits (buffering of long wires, sizing of cells)



And the search space gets even larger if one allows operators with more than 2 inputs.

So there is more fun to be had .

Conclusion

Search based on recursive decomposition gives promising results

Need to look at lazy dynamic programming

Need to do some theory about optimality (taking into account fanout)

Will try to apply similar ideas in data parallel programming on GPU (where scan is also important)