Functional domain-specific languages

Emil Axelsson

Guest lecture, IntroFP, 2011

My history

- Lava
  - Functional hardware description language
  - Embedded in Haskell
  - My first encounter with Haskell (last course before graduation)
  - My masters thesis: multiplier circuits in Lava
- Wired
  - Extending Lava with layout- and wire-awareness
  - … to be able to predict and control performance
  - My Ph.D.
- Feldspar
  - Functional language for digital signal processing
  - Motivating application: mobile radio base stations
  - Funded by Ericsson
  - My current job

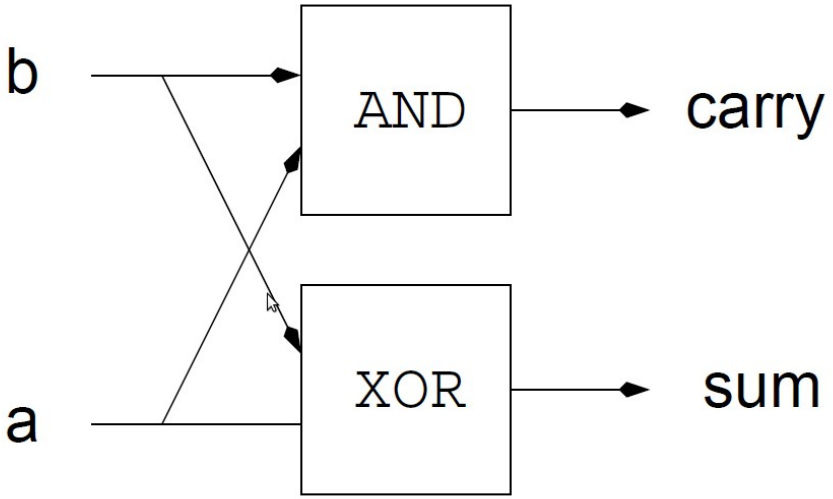
Links

All three languages are available as public Haskell packages:

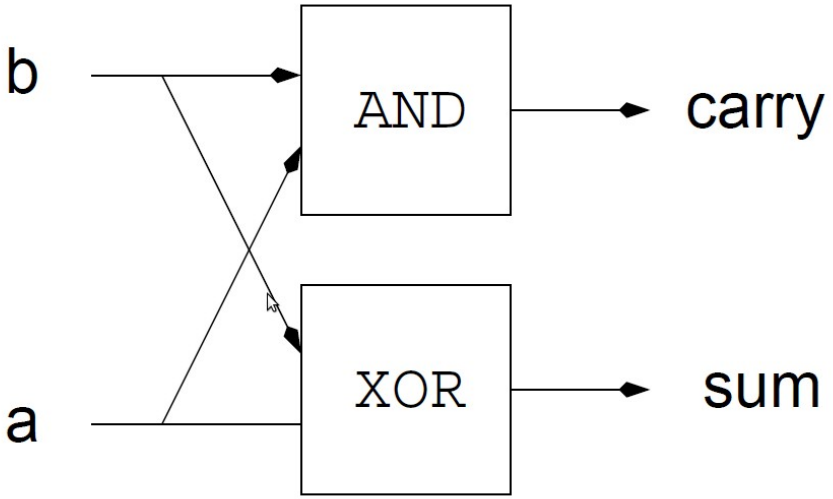
- http://hackage.haskell.org/package/chalmers-lava2000
- http://hackage.haskell.org/package/Wired
- http://hackage.haskell.org/package/feldspar-language
  http://hackage.haskell.org/package/feldspar-compiler

- Feldspar page: http://feldspar.inf.elte.hu/feldspar/

Lava – half adder

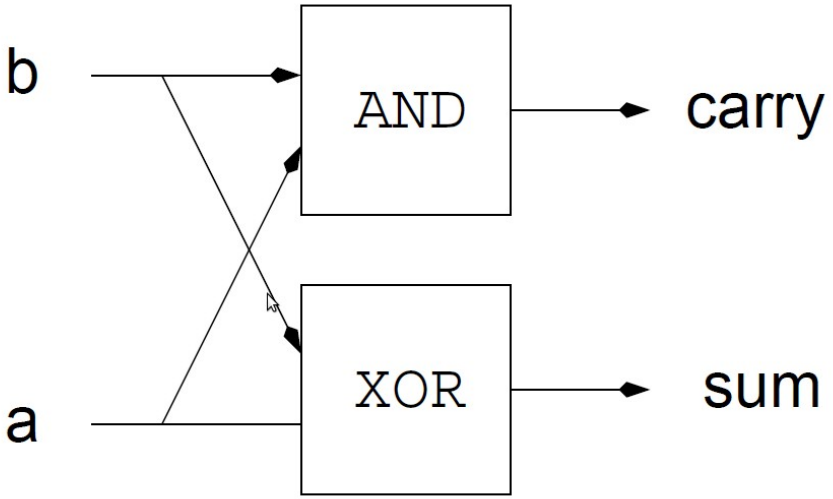


Lava – half adder



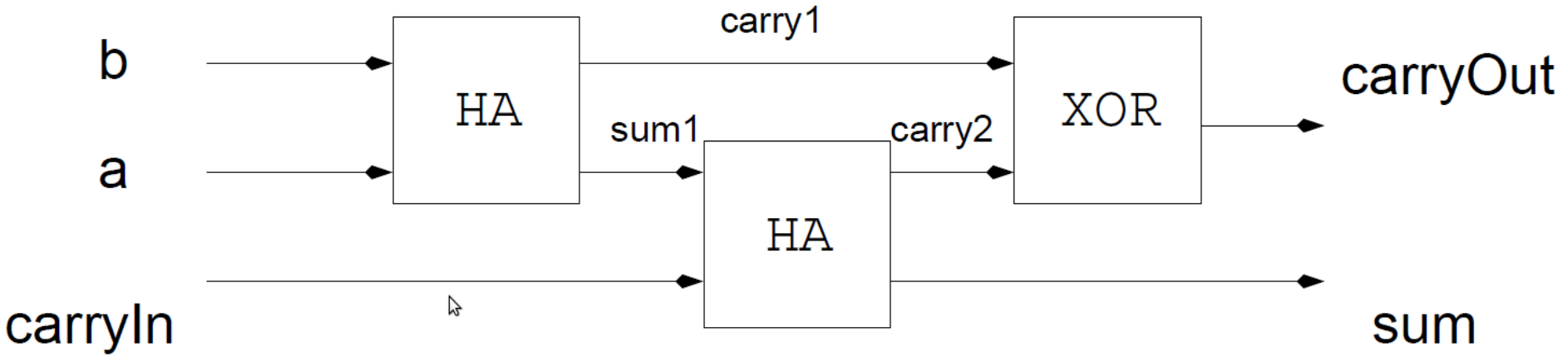

```
halfAdd (a,b) = (sum,carry)
  where
    sum   = ...
    carry = ...
```

Lava – half adder

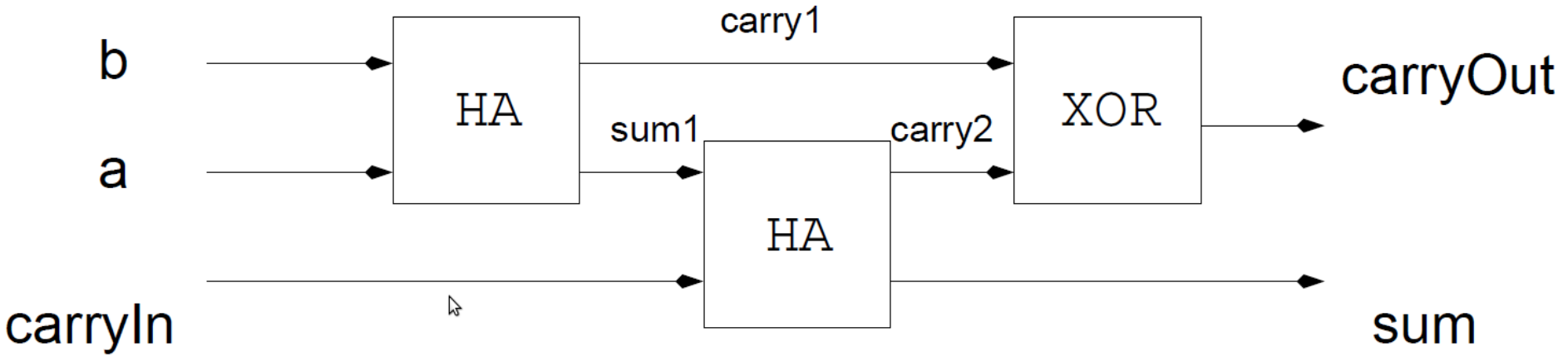



```
halfAdd (a,b) = (sum,carry)
  where
    sum   = xor2 (a,b)
    carry = and2 (a,b)
```

Lava – full adder

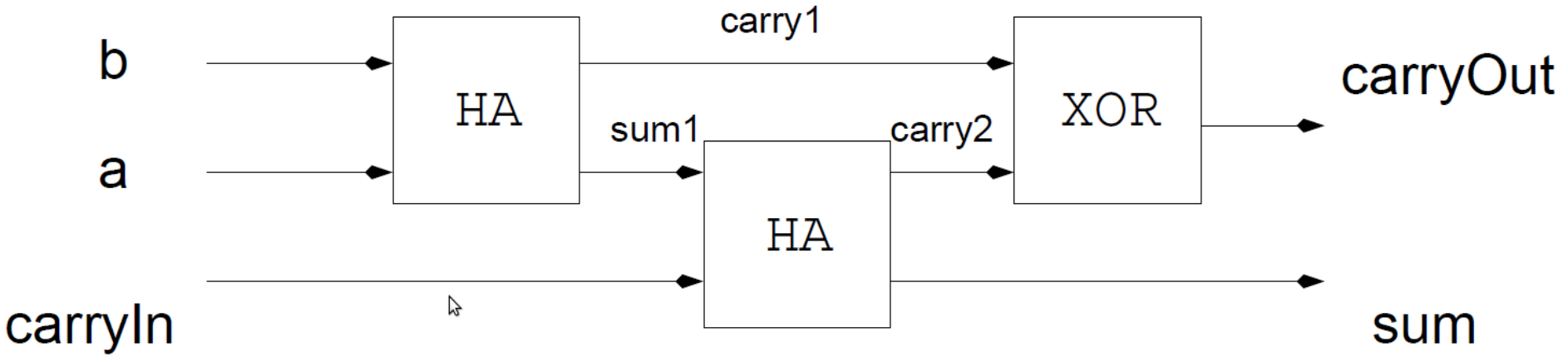


Lava – full adder



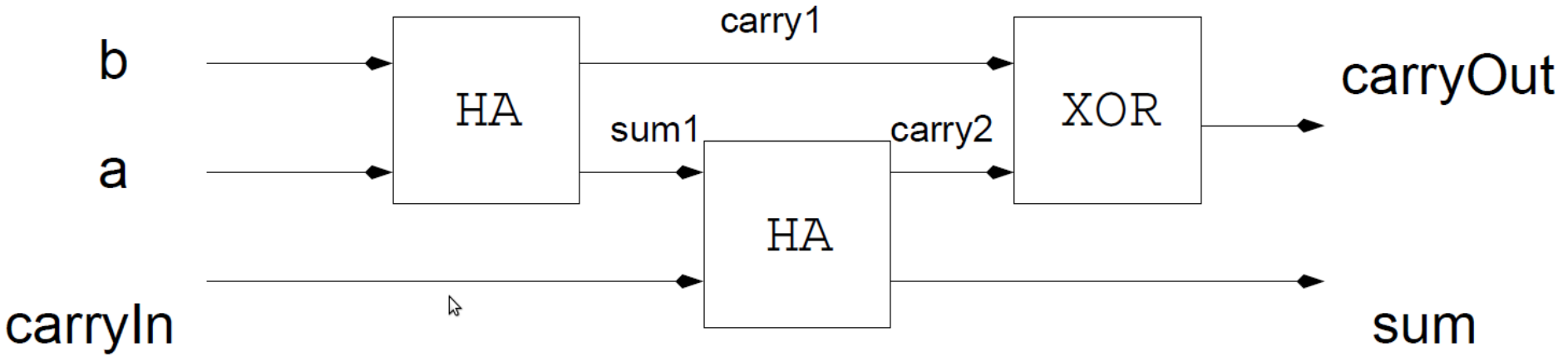

```
fullAdd (carryIn,(a,b)) = (sum,carryOut)
  where
    ...
```

Lava – full adder



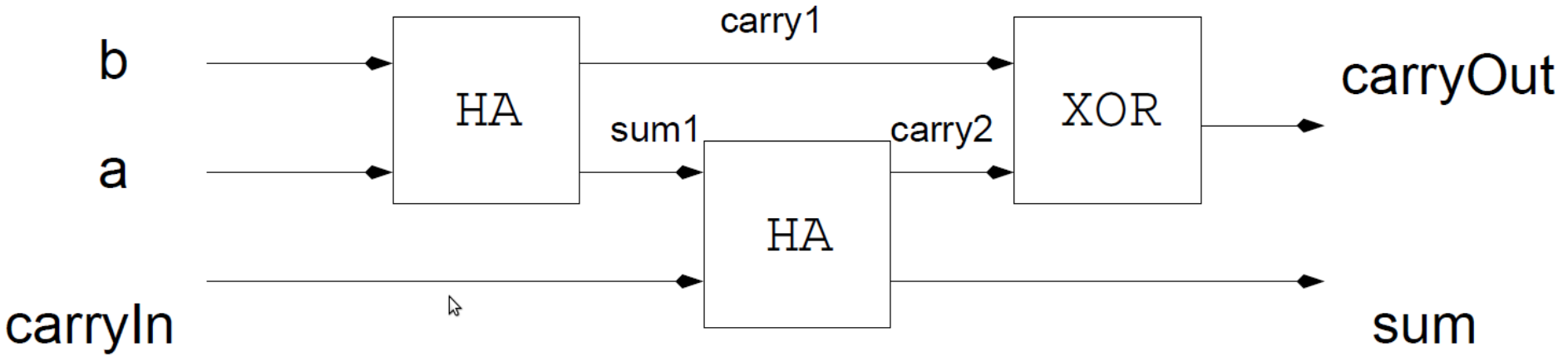

```
fullAdd (carryIn,(a,b)) = (sum,carryOut)
  where
    (sum1,carry1) = ...
    (sum,carry2)  = ...
    carryOut      = ...
```

Lava – full adder



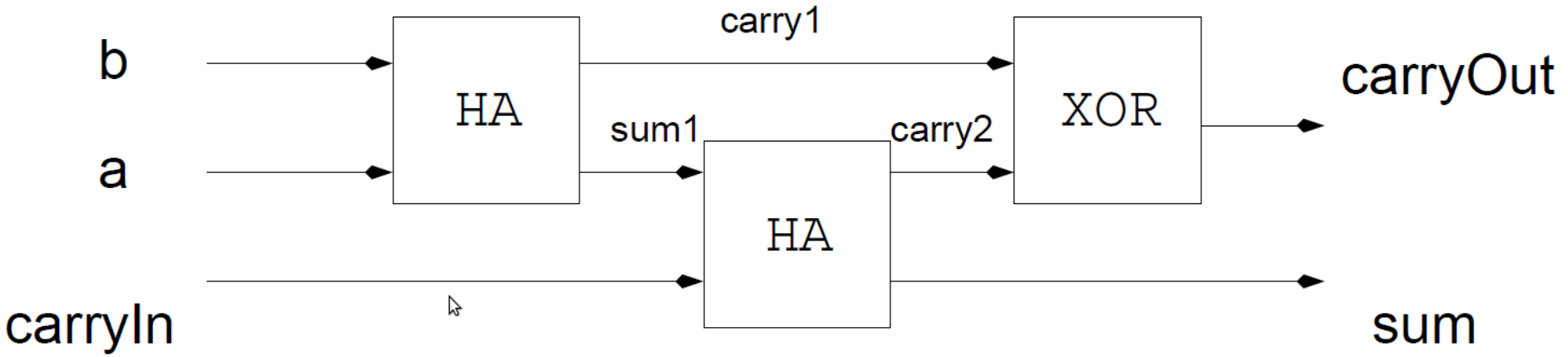

```
fullAdd (carryIn,(a,b)) = (sum,carryOut)
  where
    (sum1,carry1) = halfAdd (a,b)
    (sum,carry2)  = ...
    carryOut      = ...
```

Lava – full adder




```
fullAdd (carryIn,(a,b)) = (sum,carryOut)
  where
    (sum1,carry1) = halfAdd (a,b)
    (sum,carry2)  = halfAdd (carryIn,sum1)
    carryOut      = ...
```

Lava – full adder




```
fullAdd (carryIn,(a,b)) = (sum,carryOut)
  where
    (sum1,carry1) = halfAdd (a,b)
    (sum,carry2)  = halfAdd (carryIn,sum1)
    carryOut      = xor2 (carry2,carry1)
```

Lava

- Describe circuits by “connecting” Haskell functions
- Much simpler than standard hardware description languages
- Use higher-order functions to capture common “patterns”
  - Example:

```
bitMult b as = map (\a → and2 (b,a)) as
```

- Based on an expression data type, similar to lab 4
  Important difference:
  - In lab 4, the user enters a String that is parsed
  - In Lava, the user writes plain Haskell
    (parsing, type checking, etc. for free!)

Wired

- Based on Lava
- Uses monadic notation
- Annotation functions to control layout

Wired example: parallel prefix

```
sklansky op [a] = return [a]

sklansky op as = do

    let k        = length as `div` 2
        (ls,rs) = splitAt k as'

    ls' <- sklansky op ls
    rs' <- sklansky op rs

    rs'' <- sequence [op (last ls', r) | r <- rs']
    return (ls' ++ rs'')
```

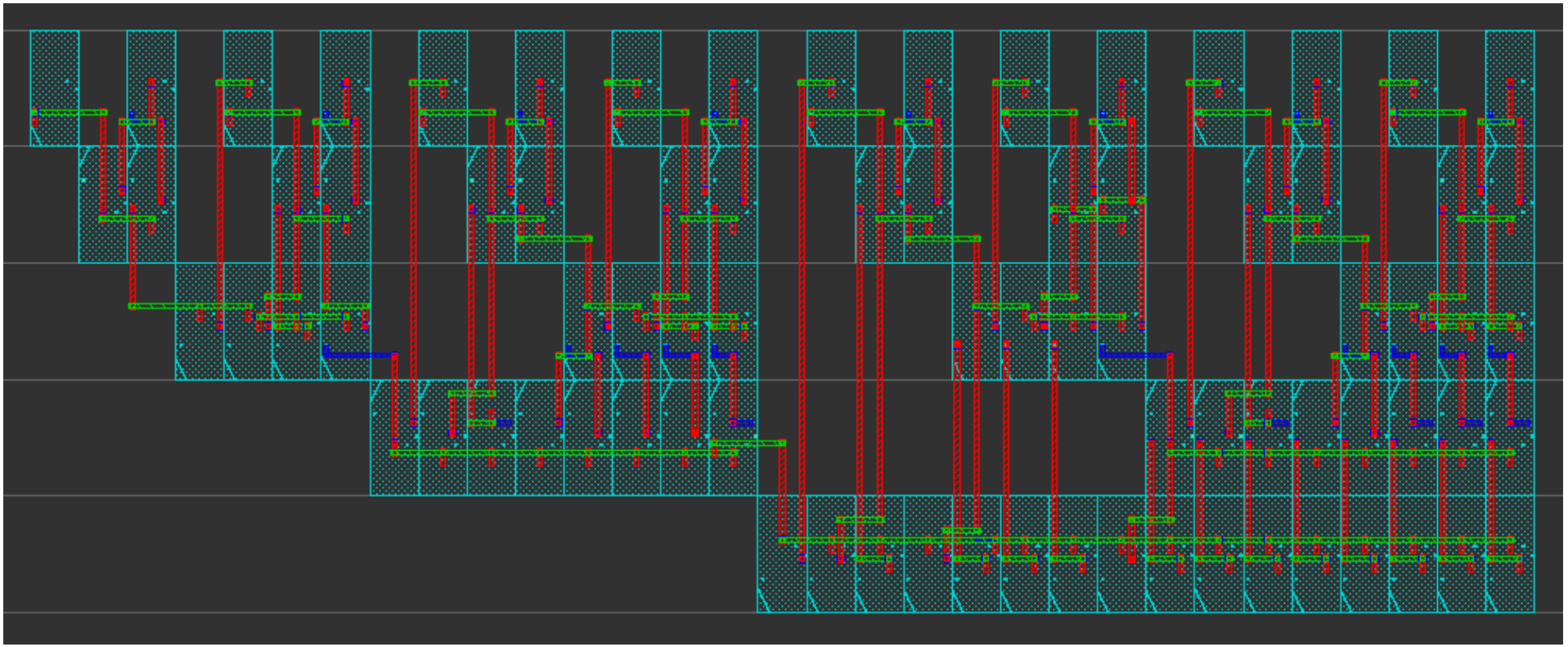
Purely structural
(no geometry)

Wired example: parallel prefix

```
sklansky op [a] = space cellWidth [a]

sklansky op as = downwards 1 $ do

    let k        = length as `div` 2
        (ls,rs) = splitAt k as'

    (ls',rs') <- rightwards 0 $ liftM2 (,)
        (sklansky op ls)
        (sklansky op rs)

    rs'' <- rightwards 0 $
              sequence [op (last ls', r) | r <- rs']
    return (ls' ++ rs'')
```

Insert simple placement annotations

Parallel prefix: result

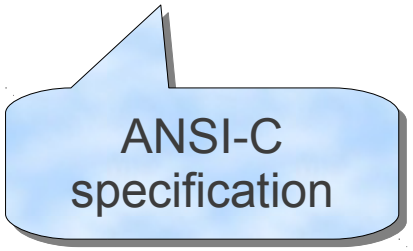


Feldspar

- Cooperation between Ericsson, Chalmers (FP group) and ELTE University (Budapest)
- Goal: Improve productivity of developing and maintaining digital signal processing (DSP) software
- Motivation: Ericsson needs to produce base stations that can handle tomorrow's mobile communications
- Current DSP software written in C
  - Hard to develop and maintain
  - Highly optimized code not portable between hardware platforms
  - Shifting to modern hardware ⇒ major software rewrite

C example: part of AMR voice codec

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```



C example: part of AMR voice codec

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

ANSI-C specification

Equivalent loop optimized for specific processor

- Pragmas
- Intrinsic instructions
- Loop unrolling
- Etc.

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
  pj_pj = _pack2 (p[j], p[j]);

  p0_p1 = _mem4_const(&p0[j+0]);
  prod0_prod1 = _smpy2 (pj_pj, p0_p1);
  t0 = _sadd (t0, _hi (prod0_prod1));
  t1 = _sadd (t1, _lo (prod0_prod1));

  p2_p3 = _mem4_const(&p0[j+2]);
  prod0_prod1 = _smpy2 (pj_pj, p2_p3);
  t2 = _sadd (t2, _hi (prod0_prod1));
  t3 = _sadd (t3, _lo (prod0_prod1));

  p4_p5 = _mem4_const(&p0[j+4]);
  prod0_prod1 = _smpy2 (pj_pj, p4_p5);
  t4 = _sadd (t4, _hi (prod0_prod1));
  t5 = _sadd (t5, _lo (prod0_prod1));

  p6_p7 = _mem4_const(&p0[j+6]);
  prod0_prod1 = _smpy2 (pj_pj, p6_p7);
  t6 = _sadd (t6, _hi (prod0_prod1));
  t7 = _sadd (t7, _lo (prod0_prod1));
}

corr[-i] = t0; corr[-i+1] = t1;
corr[-i+2] = t2; corr[-i+3] = t3;
corr[-i+4] = t4; corr[-i+5] = t5;
corr[-i+6] = t6; corr[-i+7] = t7;
```

C example: part of AMR voice codec

```
for (j = 0; j < L_frame; j++, p++, p1++)
{
   t0 = L_mac (t0, *p, *p1);
}
corr[-i] = t0;
```

ANSI-C specification

Equivalent loop optimized for specific processor

- Pragmas
- Intrinsic instructions
- Loop unrolling
- Etc.

Probably not suited for different processor.
Non-portable!

```
#pragma MUST_ITERATE(80,160,80);
for (j = 0; j < L_frame; j++)
{
  pj_pj = _pack2 (p[j], p[j]);

  p0_p1 = _mem4_const(&p0[j+0]);
  prod0_prod1 = _smpy2 (pj_pj, p0_p1);
  t0 = _sadd (t0, _hi (prod0_prod1));
  t1 = _sadd (t1, _lo (prod0_prod1));

  p2_p3 = _mem4_const(&p0[j+2]);
  prod0_prod1 = _smpy2 (pj_pj, p2_p3);
  t2 = _sadd (t2, _hi (prod0_prod1));
  t3 = _sadd (t3, _lo (prod0_prod1));

  p4_p5 = _mem4_const(&p0[j+4]);
  prod0_prod1 = _smpy2 (pj_pj, p4_p5);
  t4 = _sadd (t4, _hi (prod0_prod1));
  t5 = _sadd (t5, _lo (prod0_prod1));

  p6_p7 = _mem4_const(&p0[j+6]);
  prod0_prod1 = _smpy2 (pj_pj, p6_p7);
  t6 = _sadd (t6, _hi (prod0_prod1));
  t7 = _sadd (t7, _lo (prod0_prod1));
}

corr[-i] = t0; corr[-i+1] = t1;
corr[-i+2] = t2; corr[-i+3] = t3;
corr[-i+4] = t4; corr[-i+5] = t5;
corr[-i+6] = t6; corr[-i+7] = t7;
```

Functional DSP

Typical DSP operations (math. notation)

- Element-wise multiplication $x_i = a_i \cdot b_i$
- Moving average $x_i = \frac{a_{i+1} + a_i}{2}$
- Scalar product $x = \sum a_i \cdot b_i$

Functional DSP

Try to express these operations as Haskell functions:

$$x_i = a_i \cdot b_i \qquad x_i = \frac{a_{i+1} + a_i}{2} \qquad x = \sum a_i \cdot b_i$$

```
elemMult  :: [Float] -> [Float] -> [Float]

movingAvg :: [Float] -> [Float]

sProd     :: [Float] -> [Float] -> Float
```

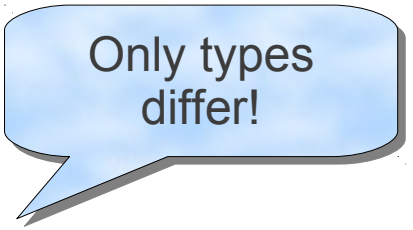
Haskell and Feldspar

- Haskell:

```
elemMult  :: [Float] -> [Float] -> [Float]
movingAvg :: [Float] -> [Float]
sProd     :: [Float] -> [Float] -> Float

elemMult as bs = zipWith (*) as bs
movingAvg as   = zipWith (\a a' -> (a+a')/2) (tail as) as
sProd as bs    = sum $ zipWith (*) as bs
```

- Feldspar:

```
elemMult  :: DVector Float -> DVector Float -> DVector Float
movingAvg :: DVector Float -> DVector Float
sProd     :: DVector Float -> DVector Float -> Data Float

elemMult as bs = zipWith (*) as bs
movingAvg as   = zipWith (\a a' -> (a+a')/2) (tail as) as
sProd as bs    = sum $ zipWith (*) as bs
```



Haskell and Feldspar

- Haskell code not suitable for running in a base station
  - Very limited memory
  - Top speed needed

- However, Feldspar produces efficient C code:

```
*Main> icompile movingAvg

...
void test(struct array mem, struct array in0, struct array * out1)
{
    ...
        for(i5 = 0; i5 < w4; i5 += 1)
        {
            at(float,(* out1),i5) =
                    ((at(float,in0,(i5 + 1)) + at(float,in0,i5)) / 2.0f);
        }
}
```

Feldspar summary

- Efficient, functional description of DSP algorithms
- Programming style close to ordinary Haskell
- Has been used to generate small parts of real baseband software at Ericsson
  - Very positive reactions from (two) programmers
  - Performance results: both positive and negative
- Ongoing research:
  - How to better control execution and memory use without losing the functional feel of the programs?
  - How to build larger applications with many concurrent tasks running?