Homework solution

Nan Yang   Student number: 0963123
`n.yang@student.tue.nl`

January 13, 2017

Contents

1 Autoregressive Filter

This section is going to present my solution for exercise 1. The implementation code can be found in the attached folder.
The formula of an autoregressive filter is as follows:

$$y_z = x_z + \sum_{i=0}^{N-1} a_i \times y_{z-i-1}$$

Here two assumptions as following are made:

1. Let a be a bounded sequence of length $N = 5$; i.e., $a \in \{j \to V\}$ and $a_j = 0$ for $j < 0$ and $j \geq N$.
2. Let h be a bounded sequence of length $z_{end} = 9$; i.e., $y \in \{j \to V\}$ and $y_j = 0$ for $j < 0$ and $j \geq z_{end}$.

Then we can derive all boundaries from the conditions and assumptions:
$a_i y_{z-i-1} = 0$ for $i < 0, z - i - 1 < 0, i > N - 1$ and $z - i - 1 > z_{end} - 1$.
With these limitations, the dependency graph can be drawn.

1.1 Dependency graph

A basic cell of the dependency graph is showed as below, where $S_{z,i} = S_{z,i-1} + a_i \times y_{z-i-1}$ and $S_{z,-1} = x_z$. The addition and multiplication are the operations performed in the basic cell. They form to a processing element.

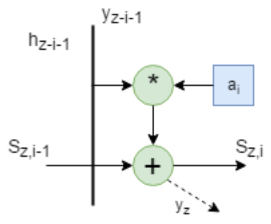


Figure 1: basic cell

Based on the basic cell, we can have a dependency graph as below(figure 2).

1.2 Scheduling vectors range

There are some restrictions on choosing $\vec{s}$ and $\vec{d}$. The equation $\vec{s}.\vec{d} \neq 0$ should be satisfied which is easy to understand as it is impossible to project equi-time zone operations on a single processing element. Additionally, the scheduling vector expresses the order of the operations will be executed in the PE of SFG. Since the operation will not depend on future's operation, the angle between scheduling vector $\vec{s}$ and edges $\vec{e}$ should be less than 90 degrees which can be expressed as $\vec{s}^{\mathrm{T}}.\vec{e} > 0$. And for the streaming environment, the condition $\vec{s}^{\mathrm{T}}.\vec{d} > 0$ is also

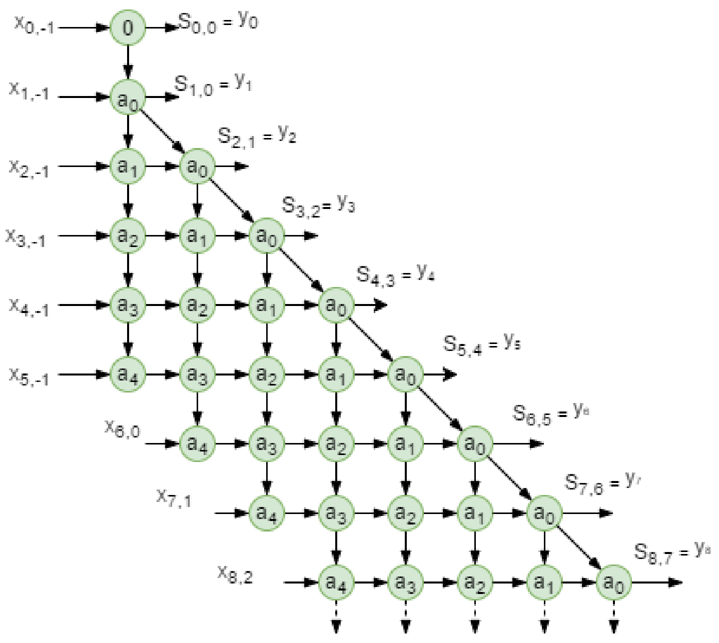


Figure 2: Dependency graph

required. However, with our initial assumption, the number of inputs and outputs is finite. They can be mapped in any order on various inputs in the SFG. Therefore, only $\vec{s}^{\mathrm{T}}.\vec{e} > 0$ is necessary to observe. In this dependency graph, only vector $\vec{e_1}$, vector $\vec{e_2}$ and vector $\vec{e_3}$ are existed, so the scheduling vectors can be picked up within the range showed in figure 3 where the blue part is the valid range for scheduling vectors.

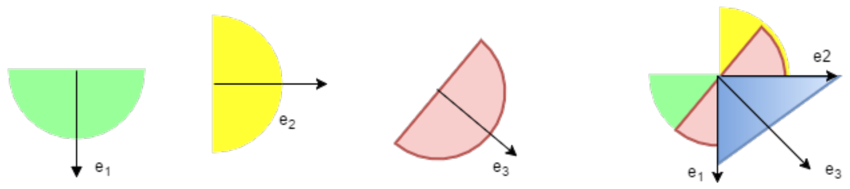


Figure 3: Scheduling vectors range

1.3 Equi-time zones

Theoretically, we can randomly pick up a scheduling vector within the range derived above. While here two vectors $(1, 1)$ and $(1, 0)$ are chosen for the easier calculation. The dependency graph with different scheduling vectors are showed in figure 4. The red dash line separates equi-time zones for scheduling vector $(1, 0)$. While blue dash line separates equi-time zones for scheduling vector $(1, 1)$.

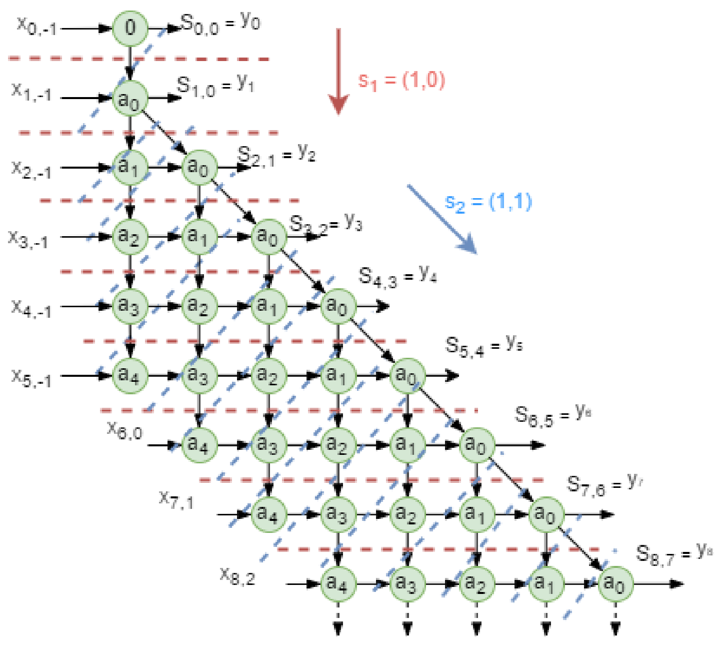


Figure 4: Dependency graph with scheduling vectors

1.4 Signal flow graphs

The projection vector $\vec{d} = (1, 1)$ is adopted. Then we can project all processing elements on the hyper-plane which is perpendicular to $\vec{d}$. And with different scheduling vectors, different delay units and annotations can be obtained by formulas below:

$$\vec{W}_{source} = P.\vec{v}_{source} \qquad n = \vec{s}.\vec{v}_{dest} - \vec{v}_{source} \qquad t = \vec{s}.\vec{v}_{source}$$

The right side part of figure 5 shows the signal flow graphs with scheduling vector $\vec{s} = (1, 1)$. While the left side shows the signal flow graphs with scheduling vector $\vec{s} = (1, 0)$.

1.5 Haskell implementation

Figure 6 shows the code of implementation in Haskell for scheduling vector $\vec{s} = (1, 0)$. A **dotProduct** function is defined and a **sim** function is used to simulate the **dotProduct** function. The ys is the initial state, the inp is the input value, and $coefs$ is a list of coefficients for the filter.

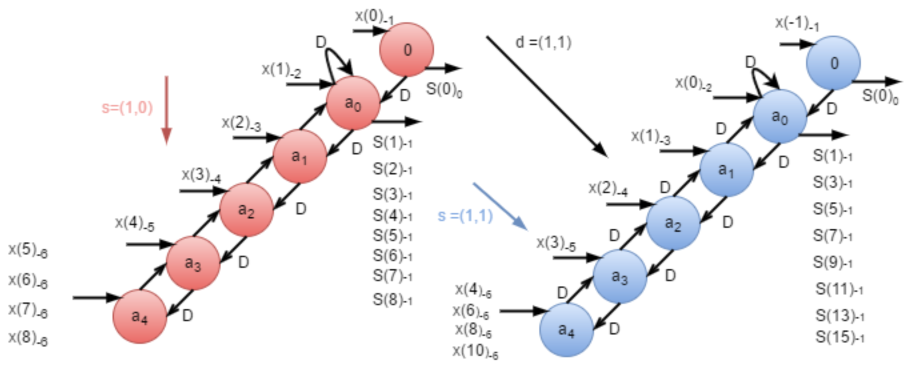
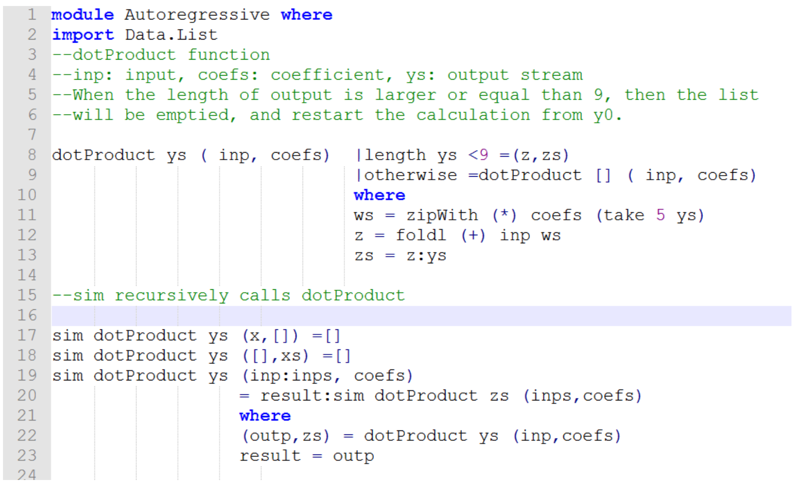


Figure 5: Signal flow graphs with different scheduling vector

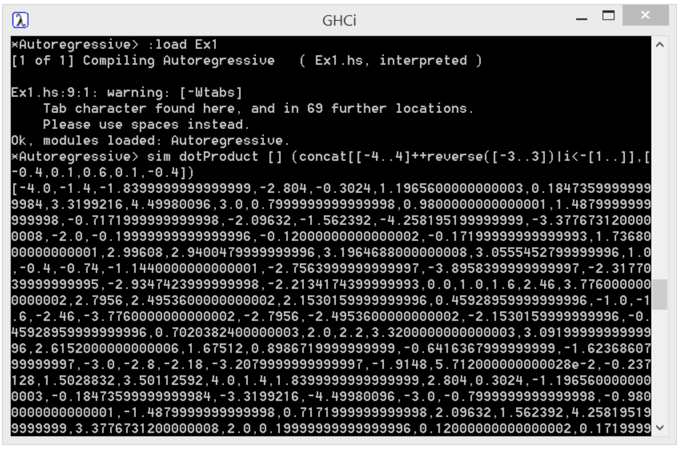


```haskell
module Autoregressive where
import Data.List
--dotProduct function
--inp: input, coefs: coefficient, ys: output stream
--When the length of output is larger or equal than 9, then the list
--will be emptied, and restart the calculation from y0.

dotProduct ys ( inp, coefs)  |length ys <9 =(z,zs)
                             |otherwise =dotProduct [] ( inp, coefs)
                             where
                             ws = zipWith (*) coefs (take 5 ys)
                             z = foldl (+) inp ws
                             zs = z:ys

--sim recursively calls dotProduct

sim dotProduct ys (x,[]) =[]
sim dotProduct ys ([],xs) =[]
sim dotProduct ys (inp:inps, coefs)
                  = result:sim dotProduct zs (inps,coefs)
                  where
                  (outp,zs) = dotProduct ys (inp,coefs)
                  result = outp
```

Figure 6: Haskell code screenshot

1.6 Simulation result

During the simulation, the initial state is assigned an empty list , the *inps* is assigned a input stream $concat[[-4..4] + +reverse([-3..3])|i < -[1..]]$ and the $coefs$ is assigned a coefficient list $[-0.4, 0.1, 0.6, 0.1, -0.4]$. Figure 7 shows the simulation results.



```
*Autoregressive> :load Ex1
[1 of 1] Compiling Autoregressive   ( Ex1.hs, interpreted )

Ex1.hs:9:1: warning: [-Wtabs]
    Tab character found here, and in 69 further locations.
    Please use spaces instead.
Ok, modules loaded: Autoregressive.
*Autoregressive> sim dotProduct [] (concat[[-4..4]++reverse([-3..3])|i<-[1..]],[
-0.4,0.1,0.6,0.1,-0.4])
[-4.0,-1.4,-1.8399999999999999,-2.804,-0.3024,1.1965600000000003,0.184735999999
9984,3.3199216,4.49980096,3.0,0.7999999999999998,0.9800000000000001,1.487999999
999998,-0.7171999999999998,-2.09632,-1.562392,-4.258195199999999,-3.37767312000
0008,-2.0,-0.19999999999999996,-0.12000000000000002,-0.17199999999999993,1.73680
0000000001,2.99608,2.9400479999999996,3.1964688000000008,3.0555452799999996,1.0
,-0.4,-0.74,-1.1440000000000001,-2.7563999999999997,-3.8958399999999997,-2.31770
39999999995,-2.9347423999999998,-2.2134174399999993,0.0,1.0,1.6,2.46,3.776000000
0000002,2.7956,2.4953600000000002,2.1530159999999996,0.4592895999999996,-1.0,-1
.6,-2.46,-3.7760000000000002,-2.7956,-2.4953600000000002,-2.1530159999999996,-0.
4592895999999996,0.7020382400000003,2.0,2.2,3.3200000000000003,3.091999999999999
96,2.6152000000000006,1.67512,0.8986719999999999,-0.6416367999999999,-1.62368607
99999997,-3.0,-2.8,-2.18,-3.2079999999999997,-1.9148,5.712000000000028e-2,-0.237
128,1.5028832,3.50112592,4.0,1.4,1.8399999999999999,2.804,0.3024,-1.19656000000
0003,-0.18473599999999984,-3.3199216,-4.49980096,-3.0,-0.7999999999999998,-0.980
0000000000001,-1.4879999999999998,0.7171999999999998,2.09632,1.562392,4.25819519
9999999,3.377673120000008,2.0,0.19999999999999996,0.12000000000000002,0.1719999
```

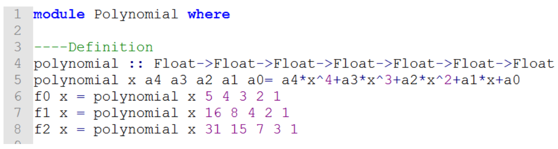
Figure 7: Simulation results

2 Polynomial

This section is going to present my solutions for the exercise 2. The implementation and RTL-schematics can be found in the attached folder.

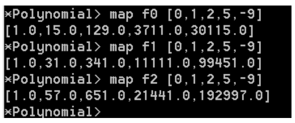
2.1 Haskell syntax

The definition of functions is written in Haskell syntax. Figure 9 below shows the code and Figure 10 shows the test result with some values in GHCI



```haskell
module Polynomial where

----Definition
polynomial :: Float->Float->Float->Float->Float->Float->Float
polynomial x a4 a3 a2 a1 a0= a4*x^4+a3*x^3+a2*x^2+a1*x+a0
f0 x = polynomial x 5 4 3 2 1
f1 x = polynomial x 16 8 4 2 1
f2 x = polynomial x 31 15 7 3 1
```

Figure 9: Haskell syntax



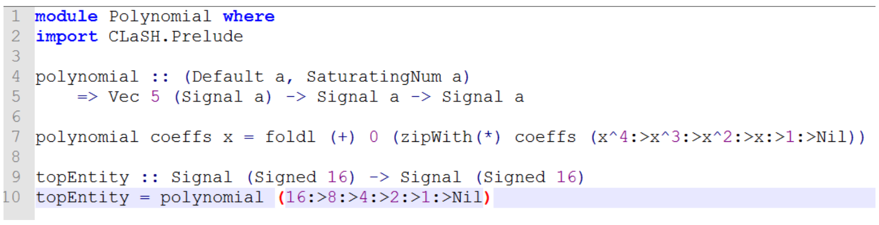
```
*Polynomial> map f0 [0,1,2,5,-9]
[1.0,15.0,129.0,3711.0,30115.0]
*Polynomial> map f1 [0,1,2,5,-9]
[1.0,31.0,341.0,11111.0,99451.0]
*Polynomial> map f2 [0,1,2,5,-9]
[1.0,57.0,651.0,21441.0,192997.0]
*Polynomial>
```

Figure 10: Values test

2.2 $C\Lambda aSH$ implementation

In last section, the functions are defined in GHCI with *Float* as input data type. However, $C\Lambda aSH$ has limitation on using floating point number. The reason given by $C\Lambda aSH$ document is that implementing purely combinational (not pipelined) arithmetic circuits for floating point data type can unreasonably and extremely slow down the circuit. Therefore, the data type for input and co-efficients are defined as *signed 16* here.
The quartus generated RTL-schematics show the different structures among different variants. The one with co-efficients $(31, 15, 7, 3, 1)$ has the most complex structure, while the one with co-efficients $(16, 8, 4, 2, 1)$ has the simplest structure. Figure 11 shows the optimization part that is done by quartus in the structure of the variant $(16, 8, 4, 2, 1)$. In this figure, the *mul0* generates x^4. The multiplication result is 32 bits. While only the signed bit(31st) and

the valid data bits are signaled to adder. For example, the *mul0* only signals 31st bit and previous 11 bits. While the left 4 bits are connected to ground which makes the original number $2^4 = 16$ times larger. By doing in this way, four multipliers are reduced. So the RTL-schematic of this variant is simpler than the others.



```haskell
module Polynomial where
import CLaSH.Prelude

polynomial :: (Default a, SaturatingNum a)
    => Vec 5 (Signal a) -> Signal a -> Signal a

polynomial coeffs x = foldl (+) 0 (zipWith(*) coeffs (x^4:>x^3:>x^2:>x:>1:>Nil))

topEntity :: Signal (Signed 16) -> Signal (Signed 16)
topEntity = polynomial (16:>8:>4:>2:>1:>Nil)
```

Figure 10: Clash code with variant $(16, 8, 4, 2, 1)$

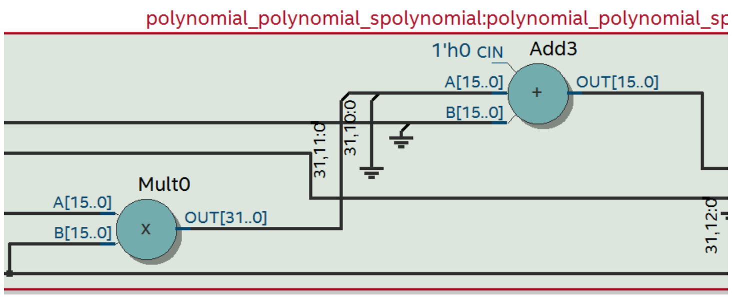
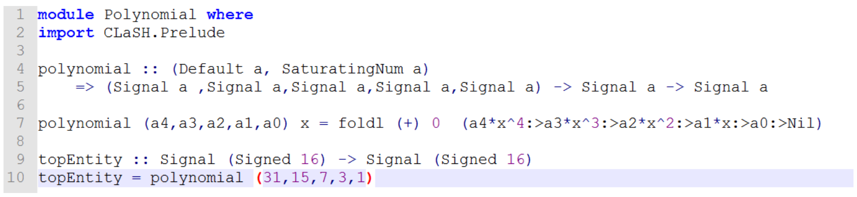


Figure 11: Quartus optimization

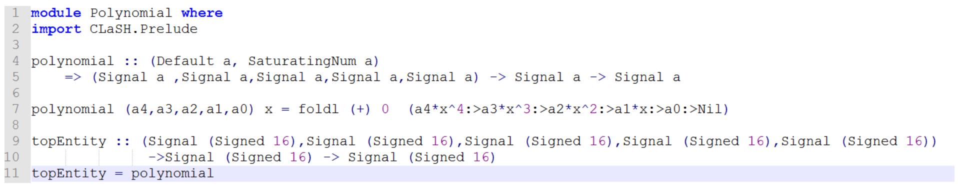
2.3 Use of tuple

The co-efficients are defined as a five elements' tuple. Figure 12 shows the definition of *topEntity* variable with concrete instantiation for the co-efficients while figure 13 shows the definition of *topEntity* variable without the concrete instantiation.



```haskell
module Polynomial where
import CLaSH.Prelude

polynomial :: (Default a, SaturatingNum a)
    => (Signal a ,Signal a,Signal a,Signal a,Signal a) -> Signal a -> Signal a

polynomial (a4,a3,a2,a1,a0) x = foldl (+) 0  (a4*x^4:>a3*x^3:>a2*x^2:>a1*x:>a0:>Nil)

topEntity :: Signal (Signed 16) -> Signal (Signed 16)
topEntity = polynomial (31,15,7,3,1)
```

Figure 12: Use of tuple with instantiation



```haskell
module Polynomial where
import CLaSH.Prelude

polynomial :: (Default a, SaturatingNum a)
    => (Signal a ,Signal a,Signal a,Signal a,Signal a) -> Signal a -> Signal a

polynomial (a4,a3,a2,a1,a0) x = foldl (+) 0  (a4*x^4:>a3*x^3:>a2*x^2:>a1*x:>a0:>Nil)

topEntity :: (Signal (Signed 16),Signal (Signed 16),Signal (Signed 16),Signal (Signed 16),Signal (Signed 16))
           ->Signal (Signed 16) -> Signal (Signed 16)
topEntity = polynomial
```

Figure 13: Use of tuple without instantiation

2.4 Mathematical equivalence

Table1 shows three other equivalences that are discussed and compared with the original one.

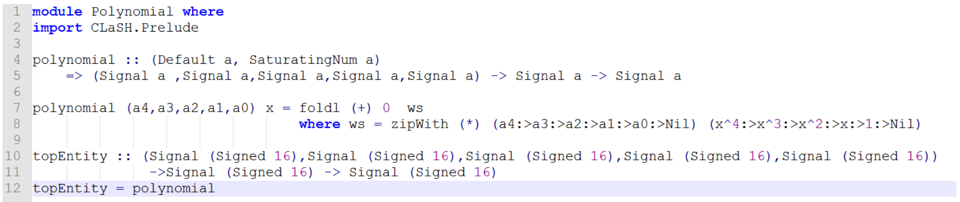
Equivalence	DSP block usage
$a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$	14 blocks
$(a_4x^3 + a_3x^2 + a_2x^1 + a_1)x + a_0$	12 blocks
$(a_4x^2 + a_3x + a_2)x^2 + a_1x + a_0$	10 blocks
$(a_4x + a_3)x^3 + a_2x^2 + a_1x + a_0$	12 blocks

Table 1: equivalences and their resource usage

Figure 1 shows the three equivalence and their FPGA resource usage. The original equivalence make use of 14 blocks while other equivalences use less resource. The third equivalence $(a_4x^3 + a_3x^2 + a_2x^1 + a_1)x + a_0$ only uses 10 DSP blocks. The RTL-schematics shows the reason why these variants are structurally different. The fact is that, by factorizing equivalence, the $16b*16b$ multiplication is reduced. For instance, the implementation of equivalence $a_4x^4+a_3x^3+a_2x^2+a_1x+a_0$ requires 7 multiplications namely $x*x, x^2*x, x^3*x, a_4*x^4, a_3*x^3, a_2*x^2$ and a_1*x. While that of equivalence $(a_4x^2+a_3x+a_2)x^2+a_1x+a_0$ only requires 5 multiplications namely $x*x, a_4*x^2, a_3*x, (a_4x^2+a_3x+a_2)*x^2$ and a_1*x. While the other two equivalences both require 6 multiplications respectively. The fewer multiplication operations occupies fewer DSP blocks. So we can observe that the generated structures can be different even the variants are equivalent mathematically. A more efficient mathematical description of a architecture sometimes leads to a structure with better efficiency.

2.5 Implementation by Higher order function

The *foldl* and *zipWith* are used in this version of code. Figure 14 shows the implementation. The RTL-schematic is in the attached folder.



```haskell
module Polynomial where
import CLaSH.Prelude

polynomial :: (Default a, SaturatingNum a)
    => (Signal a ,Signal a,Signal a,Signal a,Signal a) -> Signal a -> Signal a

polynomial (a4,a3,a2,a1,a0) x = foldl (+) 0  ws
                                where ws = zipWith (*) (a4:>a3:>a2:>a1:>a0:>Nil) (x^4:>x^3:>x^2:>x:>1:>Nil)

topEntity :: (Signal (Signed 16),Signal (Signed 16),Signal (Signed 16),Signal (Signed 16),Signal (Signed 16))
           ->Signal (Signed 16) -> Signal (Signed 16)
topEntity = polynomial
```

Figure 14: Implementation by Higher order function