Capítulo 1

Circuitos eletrônicos

Para construir um dispositivo eletrônico, é preciso descrever o referido dispositivo com um programa em Haskell. O compilador Haskell encarrega-se de gerar as especificações de baixo nível do circuito.

A maneira mais fácil e barata de fazer um circuito integrado é alimentar uma placa FPGA com as especificações geradas pela Haskell ou por qualquer outra linguagem adequada para a tarefa. Com isso, o circuito é produzido pela ativação de portas lógicas pré-existentes na pastilha.

Não vamos perder tempo explicando o que é FPGA. Em vez disso, vamos dar um exemplo de circuito especificado em Haskell. Mas aqui temos um problema. O compilador ghc não serve para gerar circuitos. Na próxima seção, mostramos como instalar o compilador CLaSH, que gera circuitos.

1.1 Instalando CLaSH

Como a UFU tem um proxy, talvez fosse mais fácil você instalar o CLaSH em sua casa. Em todo caso, eis como fazer a instalação da UFU:

```
~$ sudo http_proxy=proxy.ufu.br:3128 cabal update
~$ sudo http_proxy=proxy.ufu.br:3128 \
      cabal install clash-ghc --enable-documentation
```

Vai demorar uma eternidade, mas no fim de muita compilação você estará com o CLaSH em sua máquina, pronto para projetar circuitos eltrônicos.

Se você preferir instalar o CLaSH no aconchego de seu lar, não precisará de usar proxy. Eis como fazê-lo:

```
~$ sudo cabal update
~$ sudo cabal install clash-ghc --enable-documentation
```

1.2 Construindo circuitos

Construir hardware não tem nada de diferente de construir software. Basta você usar o intérprete *clash* em vez de *ghci*. Simples assim.

O intérprete *clash* é idêntico ao ghci, mas com dois comandos a mais, a saber, `:vhdl` e `:systemverilog`

Como em todo curso de eltrônica digital, o primeiro circuito que você vai construir é o clássico Multiply Accumulate Circuit (MAC). O circuito é tão simples quanto seu nome diz: Ele multiplica as *inputs* e as acumula.

```
~/hs/hwr/mac$ clash --interactive
CLaSHi, version 0.6.17 (using clash-lib, version 0.6.15):
http://www.clash-lang.org/  :? for help
CLaSH.Prelude> :set editor emacs -nw
CLaSH.Prelude> :e MAC.hs
```

```
module MAC where

import CLaSH.Prelude

ma acc (x,y) = acc + x * y

macT acc (x,y) = (acc' , output) where
    acc' = ma acc (x,y)
    output = acc

mac = mealy macT 0

topEntity :: Signal (Signed 9, Signed 9) -> Signal (Signed 9)
topEntity = mac
```

```
CLaSH.Prelude> :load MAC.hs
[1 of 1] Compiling MAC              ( MAC.hs, interpreted )
Ok, modules loaded: MAC.
*MAC> :vhdl
[1 of 1] Compiling MAC              ( MAC.hs, MAC.o )
Loading dependencies took 1.492306s
Applied 67 transformations
Normalisation took 0.282895s
Netlist generation took 0.007263s
Testbench generation took 0.000483s
Total compilation took 1.864143s
```

Antes de explicar como o circuito acima funciona, vamos ver se conseguimos fazer a compilação para verilog, systemverilog ou vhdl. Se você quiser a especificação em vhdl, digite o comando `:vhdl` – como mostrado no exemplo. Se preferir systemverilog, tecle `:systemverilog` na linha de comando do CLaSH. Finalmente, digite `:verilog` para obter a especificação nessa linguagem. Além do que foi dito, você não precisa nem saber o que é `vhdl` ou `verilog`.

1.3 Verilog e VHDL

Depois de compilar o programa, você pode verificar que surgiu uma pasta nova no diretório do projeto `mac`, a saber, `~/hs/hwr/mac/vhdl/`

Agora, você deve compilar o conteúdo da pasta `~/hs/hwr/vhdl/` e gerar um circuito de testes em uma placa de desenvolvimento xilinx ou altera.

Gerar circuitos eletrônicos a partir da especificação é extremamente fácil. As placas de desenvolvimento têm aplicações que permitem montar o circuito por meio de uma tecnologia chamada Field-Programmable Gate Array (FPGA) em um circuito integrado da xilinx ou da altera. A pastilha de teste já vem montada na placa de desenvolvimento, com fonte, input/output e tudo mais que você pode precisar para testar o circuito. Quando você estiver satisfeito com o resultado dos testes, pode gravar a montagem no circuito integrado do sistema que está construindo.

Os circuitos eletrônicos devem ser amplamente testados no intérprete Haskell antes da geração do hardware. Vamos mostrar como efetuar esses testes.

```
~/hs/hwr/mac$ clash --interactive -v0
CLaSH.Prelude> :load MAC.hs
[1 of 1] Compiling MAC              ( MAC.hs, interpreted )
Ok, modules loaded: MAC.
*MAC> ma 4 (8,9)
76
*MAC> ma 2 (3,4)
14
*MAC> :t ma
ma :: Num a => a -> (a, a) -> a
*MAC> :type ma
ma :: Num a => a -> (a, a) -> a
*MAC> _
```

A função `ma` é do tipo `ma :: Num a => a -> (a, a) -> a`, conforme podemos ver acima.

Enquanto trabalhávamos com software, não enfatizamos muito o type da função. Quando o assunto é hardware, entretanto, não podemos deixar de observar que um dos mais importantes aspectos do projeto de circuitos é que o engenheiro sempre consegue determinar, pelo *type*, se o circuito utiliza lógica combinatória ou lógica sequencial síncrona. Para enxergar isso, vamos examinar o *type* de um componente de circuito sequencial: o *register*. O *type* de um *register* tem sempre a forma abaixo.

```
register:: a -> Signal a -> Signal a
register i s = ...
```

Podemos ver no *template* acima que o segundo argumento e o resultado de um *register* têm o *type* `Signal a`

Todos os circuitos sequenciais síncronos trabalham com valores de *type* `Signal a`. Por outro lado, circuitos combinatórios trabalham com valores que não são de *type* `Signal a`

Um `Signal a` é uma lista infinita de *samples* (amostras, em francês), onde as *samples* correspondem aos valores do `Signal a` nos momentos discretos e consecutivos dos *ticks* do *clock*. Todos os componentes sequenciais são sincronizados com esse *global clock*.

Um *register* é um *latch* (dispositivo de dois estados) que somente muda de estado quando o *global clock* dá um *tick*. Vamos examinar as 4 primeiras *samples* produzidas pela função `register` do sinal constante `(signal 8)`

```
*MAC> sampleN 4 (register 0 (signal 8))
[0,8,8,8]
*MAC> _
```

Podemos ver aqui que o valor inicial do sinal é 0 e que esse valor é seguido por fileira de oitos.

Sequencial circuit. A função `register` é o principal dispositivo sequencial que é capaz de capturar um *state*. A maneira de descrever um circuito sequencial é pela utilização de um modelo de máquina.

A máquina Mealy, em contraste com a máquina de Moore, é um *finite-state automaton* cujos valores de saída (*output values*) são determinados tanto pelo estado atual quanto pelas *inputs* (= entradas).

No exemplo do circuito MAC, podemos combinar a função de transição e a função de saída em uma única definição da linguagem Haskell. Com isso, obtemos a seguinte especificação:

```
macT acc (x,y) = (acc' , output) where
    acc' = ma acc (x,y)
    output = acc
```

Quando examinamos o *type* de `macT`, descobrimos que ainda estamos lidande com um circuito completamente combinatório.

```
*MAC> :t macT
macT :: Num t => t -> (t, t) -> (t, t)
```

A versão sequencial do circuito pode ser obtida assim:

```
mac= mealy macT 0
```

Vamos testar o *type* no intérprete:

```
*MAC> :t mac
mac :: Num o => Signal (o, o) -> Signal o
```

O primeiro argumento da função `mealy` é a nossa função `macT`, e o segundo argumento é o estado inicial, nesse caso, 0. No intérprete, podemos ver que a função `mac` está funcionando corretamente.

```
~/hs/hwr/mac$ clash --interactive -v0
CLaSH.Prelude> :load MAC.hs
[1 of 1] Compiling MAC ( MAC.hs, interpreted )
Ok, modules loaded: MAC.
*MAC> import qualified Data.List
*MAC Data.List> let xs= [(1::Int,1),(2,2),(3,3),(4,4)]
*MAC Data.List> Data.List.take 4 $ simulate mac xs::[Int]
[0,1,5,14]
```

Decoder. Mostramos abaixo um decoder de 4 para 16 bits.

```
module CODE where

import CLaSH.Prelude

decoderShift :: Bool -> BitVector 4 -> BitVector 16
decoderShift enable binaryIn=
  if enable
    then shiftL 1 (fromIntegral binaryIn)
    else 0
```

```
CLaSH.Prelude> :load CODE.hs
*CODE> decoderShift True 7
0000_0000_1000_0000
```

No exemplo acima, queremos decodificar um número de 4 bits, de modo a obter um número de 8 bits.

Encoder. `\en i-> decoderShift en (encoder en i)=== i`

```
~/hs/hwr/decoder$ clash --interactive -v0
CLaSH.Prelude> :set editor emacs -nw
CLaSH.Prelude> :edit ENC.hs
```

```
module ENC where

import CLaSH.Prelude

encoderCase :: Bool -> BitVector 16 -> BitVector 4
encoderCase enable binaryIn | enable =
  case binaryIn of
    0x0001 -> 0x0
    0x0002 -> 0x1
    0x0004 -> 0x2
    0x0008 -> 0x3
    0x0010 -> 0x4
    0x0020 -> 0x5
    0x0040 -> 0x6
    0x0080 -> 0x7
    0x0100 -> 0x8
    0x0200 -> 0x9
    0x0400 -> 0xA
    0x0800 -> 0xB
    0x1000 -> 0xC
    0x2000 -> 0xD
    0x4000 -> 0xE
    0x8000 -> 0xF
    _ -> 0           -- otherwise
```

Contador. Na eletrônica, para um contador funcionar, é preciso ter um sinal gerado pelo *clock*. No programa abaixo, esse sinal é gerado por `oscillate`. A função primitiva

```
regEn :: a -> Signal Bool -> Signal a -> Signal a
```

é uma versão de `register` que somente muda seu conteúdo quando o segundo argumento torna-se `True`. No exemplo que segue, o contador começa com 0 e encontra `oscillate` em `False`. O resultado é que `upCounter` não muda

de estado, continua com 0. Então, os dois primeiros elementos do sinal `[0,0,1,1,2,2,3,3]` valem 0.

Examine a lista `oscillate=[False, True, False, True, False]`. Seguindo `oscillate`, `upCounter` encontra um `True` e um segundo `False` depois do primeiro `False`. Então `upCounter` muda para 1 pela ação do `True` e permanece em 1 ao encontrar o `False`. E assim por diante.

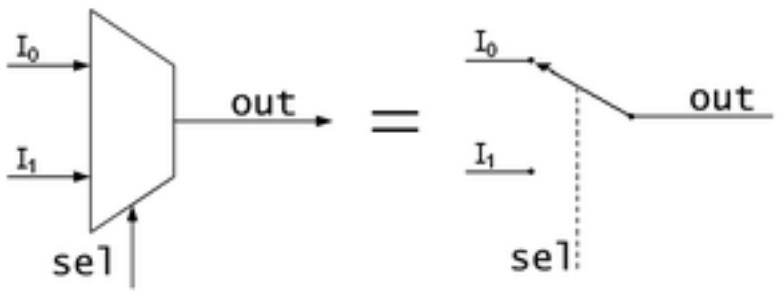
```
~/hs/hwr/decoder$ clash --interactive -v0
CLaSH.Prelude> :set editor emacs -nw
CLaSH.Prelude> :edit SimpleUpCounter.hs
```

```
module SimpleUpCounter where

import CLaSH.Prelude

upCounter :: Signal Bool -> Signal (Unsigned 8)
upCounter enable = s
  where
    s = regEn 0 enable (s + 1)

oscillate= register False (not1 oscillate)
```

```
CLaSH.Prelude> :load SimpleUpCounter.hs
*SimpleUpCounter> sampleN 8 $ upCounter oscillate
[0,0,1,1,2,2,3,3]
```

Contador de descida subida. O próximo contador deteta as passagens de `True` para `False` e de `False` para `True`.

Em eletrônica, um multiplexer, ou mux, é um dispositivo que seleciona um de vários sinais e produz o sinal selecionado em uma única linha. Na figura abaixo, podemos ver como um multiplexer de duas entradas pode ser representado diagramaticamente.



A propósito, representar circuitos em diagramas não é prático. Um computador, por exemplo, pode ter milhões de portas lógicas. Como representar isso em um diagrama? É por isso que engenheiros de computação utilizam linguagens como Haskel para representar circuitos eletrônicos.

```
~/hs/hwr/decoder$ clash --interactive -v0
CLaSH.Prelude> :set editor emacs -nw
CLaSH.Prelude> :edit UDC.hs
```

```
module UDC where

import CLaSH.Prelude

upDownCounter :: Signal Bool -> Signal (BitVector 8)
upDownCounter upDown= s
  where
    s= register 0 (mux upDown (s+1) (s+1))

oscilador= register False (not1 oscilador)
```

```
CLaSH.Prelude> :load UDC.hs
*UDC> sampleN 6 $ upDownCounter oscilador
[0000_0000,0000_0001,0000_0010,0000_0011,0000_0100,0000_0101]
```

Capítulo 2

Elementos de circuitos

Quando utilizadas para gerar circuitos, as linguagens funcionais têm dois tipos de funções primitivas:

1. Funções sintetizáveis – aparecem nos circuitos que o engenheiro está construindo. Como já vimos, os circuitos sintetizáveis podem ser:

   - Sequenciais, quando tem argumentos e resultados que são sinais.
   - Combinatórios, quando produzem um resultado que é a combinação dos argumentos de entrada.

2. Funções não sintetizáveis – não aparecem no circuito final. Essas funções servem apenas para testar os circuitos no intérprete Haskell, antes de colocá-los em pastilhas.

Isso dito, vamos examinar algumas funções para hardware, indicando aquelas que não são sintetizáveis.

mux :: `Signal Bool -> Signal a -> Signal a -> Signal a`
`mux b t f` produz `t` quando `b` é `True` e `f` quando `b` é `False`

signal :: `a -> Signal a` cria um sinal constante

```
*UDC> sampleN 5 (signal 4 :: Signal Int)
[4,4,4,4,4]
```

register :: `a -> Signal a -> Signal a`

`register i s` atrasa os valores em `Signal s` de um ciclo do relógio, e coloca `i` no tempo 0.

regEn :: `a -> Signal Bool -> Signal a -> Signal a`

Versão de `register` que só faz *update* do conteúdo quando o segundo argumento é elevado para `True`. Então, dados:

```
oscillate = register False (not1 oscillate)
count     = regEn 0 oscillate (count + 1)
```

obtemos o seguinte comportamento:

```
CLaSH.Prelude> sampleN 8 oscillate
[False,True,False,True,False,True,False,True]
CLaSH.Prelude> sampleN 8 count
[0,0,1,1,2,2,3,3]
```

.&&. :: `Signal Bool -> Signal Bool -> Signal Bool`

Trata-se da versão de (`&&`) que retorna um `Signal Bool`

.||. :: `Signal Bool -> Signal Bool -> Signal Bool`

Essa é a versão de (`||`) que retorna um `Signal Bool`

not1 :: `Signal Bool -> Signal Bool`

Versão de (`not`) que opera com `Signal Bool`

bundle :: `(Signal a, Signal b) -> Signal (a,b)`

unbundle :: `Signal (a,b) -> (Signal a, Signal b)`

unbundle :: `Signal Bit -> Signal Bit`

2.1 Funções de simulação

Como o nome indica, as funções de simulação servem apenas para isto: fazer simulações. Portanto, não são sintetizáveis.

simulate :: `(Signal' clk1 a -> Signal' clk2 b) -> [a] -> [b]`
Essa função faz simulações de uma função do tipo `Signal a -> Signal b)` aplicada a uma lista de *samples*. Exemplo:

```
CLaSH.Prelude> sampleN 8 $ simulate (register 8) [1..]
[8,1,2,3,4,5,6,7]
```

simulateB:: `(Bundle a,Bundle b)=>(Unbundled a->Unbundled b)->[a]->[b]` faz a simulação de uma função do tipo `(Unbundle a -> Unbundle b)` aplicada a uma lista de *type* `a`

```
CLaSH.Prelude> :set +m
CLaSH.Prelude> :{
CLaSH.Prelude| sampleN 6 $
CLaSH.Prelude|    simulateB (unbundle . register (8,8) . bundle)
CLaSH.Prelude|                [(i,i) | i <- [1..]]::[(Int,Int)]
CLaSH.Prelude| :}
[(8,8),(1,1),(2,2),(3,3),(4,4),(5,5)]
```

2.2 Conversão de lista para sinal

Funções que convertem de listas para sinais não são sintetizáveis.

sample :: `Signal a -> [a]`

sampleN :: `Int -> Signal a -> [a]` pega `n` *samples* de `Signal a`. Os elementos da lista `[a]` correspondem aos valores de `Signal a` em ciclos consecutivos do *clock*.

fromList :: `[a] -> Signal' clk a` gera um sinal a partir de uma lista infinita. Exemplo:

```
CLaSH.Prelude> sampleN 2 (fromList [1,2,3,4,5])
[1,2]
```

2.3 Bits class

A classe Bits define *bitwise operations* sobre *integral types*. Bits são indexados a partir de 0. O bit 0 é o menos significativo.

.&. :: `a -> a -> a` – bitwise and

.|. :: `a -> a -> a` – bitwise or

xor :: `a -> a -> a` – bitwise xor

complement :: `a -> a` – reverse all the bits in the argument

shift :: `a -> Int -> a` – `shift x i` desliza `x` de `i` bits para a esquerda se `i` é positivo. Desliza para a direita se `i` é negativo.

rotate :: `a -> Int -> a` – `rotate x i` gira `x` de `i` bits para a esquerda, se `i` é positivo, ou para a direita, se `i` é negativo.

zerobits :: `a` – é uma constante com todos os bits *unset*.

bit :: `Int -> a` – `bit i` é um valor com o bit `i` *set* e todos os outros *unset*

setBit :: `a -> Int -> a` – `setBit x i` acende o bit `i` de `x`

clearBit :: `a -> Int -> a` – `clearBit x i` apaga o bit `i` de `x`

isSigned :: `a -> Bool` – retorna `True` se o argumento é negativo

shiftL :: `a -> Int -> a` – desloca o argumento para a esquerda

shiftR :: `a -> Int -> a` – desloca o argumento para a direita

rotateL :: `a -> Int -> a` – gira argumento para a esquerda:

```
oneHotCounter :: Signal Bool -> Signal (BitVector 8)
oneHotCounter enable = s
  where
    s = regEn 1 enable (rotateL s 1)
```

rotateR :: `a -> Int -> a` – gira argumento para a direita

Capítulo 3

Classes

Haskell têm classes de funções de vários argumentos que os programadores Lisp chamam de métodos. O programador vê um conjunto desses métodos como uma única função. Como o método é escolhido pelo *type* dos argumentos, esse sistema é chamado de *parametric polymorphism*.

Para aprender mais sobre o assunto, vamos estudar a classe `Eq a` da Haskell. No `Prelude` da Haskell, a classe `Eq a` é definida assim:

```
class  Eq a  where
   (==), (/=)     :: a -> a -> Bool
   x /= y         =  not (x == y)
```

A *constraint* `Eq a` indica que um *type* `a` deve ser uma *instance* da classe `Eq`. Assim, `Eq a` não é uma expressão de *type*, mas enforça uma *constraint* em um *type*. Por isso, `Eq a` é denominada *contexto*. A consequência da declaração de classe acima é estabelecer que a função `==` tem o seguinte *type*:

```
(==)   ::   (Eq a) => a -> a -> Bool
```

É possível definir a classe `Ord`, que herda todas as operações de `Eq a`, mas adiciona comparações, além das funções de encontrar o mínimo e o máximo de dois objetos.

```
class  (Eq a) => Ord a  where
   (<), (<=), (>=), (>)  :: a -> a -> Bool
   max, min              :: a -> a -> a
```

Com a declaração acima, dizemos que `Eq` é uma superclasse de `Ord` (ou `Ord` é uma subclasse de `Eq`) e que qualquer *type* de `Ord` deve ser também uma *instance* de `Eq`

Haskell permite heranças múltiplas. Assim, o esquema abaixo define uma classe `C a` que herda operações tanto de `Eq a` quanto de `Show a`

```
class (Eq a, Show a) => YesNo a where
   yesno :: a -> Bool
```

```
module POS where

data TrafficLight = Red | Yellow | Green
  deriving  Eq

instance Show TrafficLight where
   show Red= "Red Light"
   show Yellow = "Yellow light"
   show Green = "Green light"

data Tree a = EmptyTree | Node (a, String) (Tree a) (Tree a)
   deriving (Show, Read, Eq)

class (Eq a, Show a) => YesNo a where
   yesno :: a -> Bool

instance YesNo TrafficLight where
   yesno Green= True
   yesno _ = False

instance YesNo Int where
   yesno 0= False
   yesno x= True

treeInsert x EmptyTree= Node x EmptyTree EmptyTree
treeInsert (i, x) (Node (j,y) left right)
  | i ==j = Node (i,x) left right
  | i < j= Node (j,y) (treeInsert (i,x) left) right
  | i > j= Node (j,y) left (treeInsert (i,x) right)

treeElem x EmptyTree= Nothing
treeElem x (Node (n,a) left right)
  | x== n= Just (n,a)
  | x < n= treeElem x left
  | x > n= treeElem x right

treeFromList xs = foldr treeInsert EmptyTree xs
```

Capítulo 4

FIR filter

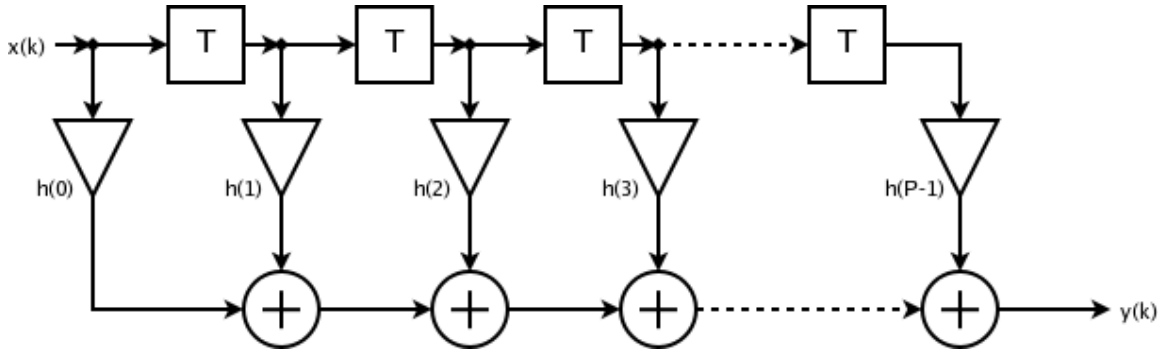
Em processamento de sinais, um Finite Response Filter (FIR) é um filtro digital caracterizado por uma resposta ao impulso que se torna nula após um tempo finito. Um filtro FIR digital genérico tem uma saída dada pela seguinte fórmula:

$$y(n) = h_0 x(n) + h_1 x(n-1) + \ldots + h_P x(n-P)$$

Na expressão acima, P é a ordem do filtro, $x(n)$ é o sinal de saída e h_i são os coeficientes do filtro. A equação anterior pode ser expressa, de maneira compacta, por:

$$y(n) = \sum_{i=0}^{P} h_i x(n-i)$$

No diagrama abaixo, os termos **h** são os coeficientes e os termos **T** são os elementos de atraso do filtro.



Discutimos, nesta seção, o programa Haskell que gera o hardware para a construção de um filtro FIR.

Programaticamente, um filtro FIR é definido como o produto escalar de um conjunto de coeficientes e uma janela sobre o sinal de *input* (= entrada), onde o tamanho da janela cobre perfeitamente o número de coeficientes.

Um dos métodos de construção de Finite Impulse Filters (FIR) consiste em projetar um *Infinite Impulse Response Filter* e, então, truncá-lo com o produto escalar dos coeficientes do filtro por uma janela de largura finita.

Para implementar o produto escalar da janela pelos coeficientes do filtro, precisaremos estudar duas funções, `zipWith` e `fold`

```
Prelude> zipWith (*) [3,4,5,6] [1..8]
[3,8,15,24]
Prelude> foldl (+) 0 [3, 8, 15, 24]
50
Prelude> _
```

No exemplo acima, `zipWith` zipou as listas `[3,4,5,6]` e `[1,2,3,4,5,6,7,9]` com a operação `(*)`. O resultado foi `[3*1,4*2,5*3,6*4]`

A função `foldl (+) 0 [3,8,15,24]` introduz um operador de dois argumentos entre os elementos de uma lista. Assim, `foldl (+) 0 [3,8,15,24]` calcula `[3+8+15+24+0]`

Na definição de `dotp`, em vez de `foldl`, utilizamos `fold` que dispensa a constante do segundo argumento. Constante essa que se introduz depois do último elemento da lista. Em suma, `foldl (+) 0 [4,5,6]` produz `[4+5+6+0]`, enquanto `fold (+) [4,5,6]` suspende a somatória ao atingir o último elemento da lista e retorna `[4+5+6]` como resultado.

```
module FIR where

import CLaSH.Prelude

dotp :: SaturatingNum a
     => Vec (n + 1) a
     -> Vec (n + 1) a
     -> a
dotp as bs = fold boundedPlus (zipWith boundedMult as bs)

fir :: (Default a, KnownNat n, SaturatingNum a)
    => Vec (n + 1) (Signal a) -> Signal a -> Signal a
fir coeffs x_t = y_t
  where
    y_t = dotp coeffs xs
    xs  = window x_t

topEntity :: Signal (Signed 16) -> Signal (Signed 16)
topEntity = fir (2:>3:>(-2):>8:>Nil)
```