基于 CLaSH 的优先队列及相关算法实现

Author：吕文龙(lvwenlong_lambda@qq.com)

摘要

对函数式硬件描述语言(Functional Hardware Description Language)的概念做了调研，并尝试使用 CLaSH[1] 这一语言完成一些电路设计工作，主要完成的工作有：

- 函数式硬件描述语言调研
- CLaSH 语言文档与相关论文阅读
- 使用 CLaSH 设计优先队列(priority queue)数据结构，并生成 verilog 代码
- 使用优先队列实现了堆排序(heap sort)，并生成 verilog 代码
- 使用优先队列实现了求图上最短路径的 A*与 Dikjstra 算法
- 使用 CLaSH 对堆排序与最短路径算法进行了仿真测试

1. 背景介绍与研究目标

背景介绍

相比传统的结构化、过程式编程语言，函数式语言的语法具有更高的抽象性与表达能力。目前，新兴的编程语言，如 Rust，Swift，大多都带有函数式的特性，而传统的编程语言如 C++，Java 等，在语言自身演进的过程中也加入了函数式的特性，如 c++11 标准加入的的函数对象，Java8 新加入的的 lambda 等。

最有名的两门函数式语言是 Lisp 与 Haskell。其中 Lisp 的特点在于独特的 S-Expression 语法，以及强大的宏(Macro)系统。而 Haskell 则是因为其激进的函数式特征，如强大的类型系统，以及不可变数据结构。

函数式语言的基本特征有：

1. 高阶函数，像操作数据那样操作函数，函数可以作为参数传入另一个函数，函数也可以作为另一个函数的返回值动态的生成。
2. 不可变数据结构，在纯函数式语言中，变量一旦声明，值就不可以再改变。例如 x=x+1 这样的语句，就不是函数式的风格。不过，一门不允许可变状态的语言注定是无法与真实世界交互的，函数式语言往往提供独特的机制来处理可变状态(即函数的“副作用”)。函数式语言并非是要禁止可变的状态，而是要隔离可变的状态与不可变的状态。
3. 因为没有可变的状态，函数式语言往往通过递归而非迭代来解决问题，而递归带来的栈溢出问题，可用通过尾递归优化技术(tail recursion optimization)，在编译时将递归优化为占用常量空间的迭代。

函数式语言的以上特征中，高阶函数提供了强大抽象性，一种高阶函数往往代表一种设计模式(Design Pattern)。谷歌公司研究总监(Director of Search Quality) Peter Norvig 曾经给过

一次报告指出，大多数在 java 中面向对象的设计模式，在具有函数式特性的语言中都可以轻易的实现。

不可变的数据结构，一来为使得函数具有引用透明(reference transparency)的特性，即对一个函数，如果给定相同的输入，必定会给出相同的输出。这使得对程序的静态分析以及程序优化更加方便。另一方面，这也带来了隐式的并行性(implicit parallism)。例如，对一个函数 `f(x,y)`，在一门具有副作用的语言中，`x` 与 `y` 的求值顺序必须是事先明确的，否则，对于 `f(x++,x++)` 这样的语句，就无法给出确定的值，而对于一门函数式语言，最终的返回值，是与 x 和 y 的求值顺序无关的，因而在理论上，`x` 与 `y` 是可以并行求值的。
事实上，对函数式语言来说，限制其在计算机上并行性的，并不是并行性难以发掘，而是其中可自动并行的部分太多，但是粒度太细。如果对 `f(x,y)` 这样的函数调用都开多个线程并行对参数求值的话，则并行带来的额外负担就会抵消掉其加速比。不过，如果要将一个函数综合为电路，则这种细粒度的并行并不会成为负担。因为一个电路的输入，本来就是并行的。 从这个角度来讲，函数式语言在语义层面更适合对电路的抽象，至少更适合对电路中可并行部分的抽象。

目前，大多数的函数式语言都不能称作是“纯函数式语言”，即他们大多提供高阶函数，以及不可变数据类型。但同时也允许可变的状态。可以称得上“纯函数式语言”而又具有工业级实用强度的，大抵只有 Haskell 了。即使在通用编程领域，Haskell 也不能算作一门流行的语言。一来是因为它强制要求使用者分离可变与不可变状态。这种独特的编程范式，对于习惯了过程式或者面向对象编程的程序员来说，是思维上不小的挑战。二来，因为它更强调对问题的抽象，它并不“贴近底层”。这会在一定程度上牺牲运行效率，例如，大量的使用函数与高阶函数来解决问题，而函数的调用，编译成机器码，就会是大量的跳转语句，又比如，相比对 cache 更加友好的数组，函数式的语言中更加倾向与使用单链表这一递归的数据结构。

而数字电路的设计，恰恰就是要跟底层打交道，要跟(寄存器的)状态打交道。因此，“使用 Haskell 作为硬件描述语言”这个概念，听上去既吸引人，又让人怀疑。首先，它本身的一些特性，如函数的引用透明性，函数参数的隐式可并行性，似乎都非常适合于电路设计。而对它的怀疑则主要是来自时序电路方面。如果不使用可变状态，如何描述时序电路中状态的转换？

目前，基于 Haskell 的硬件描述语言主要有：

- BlueSpec，这门语言最初是基于 Haskell 的，而在演进的过程中，放弃了许多 Haskell 的语法，目前它正式的名字是 BlueSpec Systemverilog，而且它是收费的商业版本。
- Lava, 这是一门嵌入在 Haskell 中的 DSL (Domain Specific Language), 说它是“嵌入式”，因为它的工作过程并不是直接通过静态分析(Static Analysis) Lava 代码生成底层电路描述，而是它本身作为一个 Haskell 库，通过 Haskell 解释器，执行 lava 代码，生成电路。电路本身被描述为一种独特的数据类型。其好处在于能够综合递归函数，而不足则是对于自定义数据类型，以及一些 Haskell 的特性，如模式匹配的支持不足。Lava 有多种方言，如 Xilinx-Lava，Kansas-Lava 等。

这次研究选用 CLaSH 作为研究语言，这是荷兰屯特大学的一个研究项目。大约开始于 2009 年，最新的文章是今年 3 月份的一篇 PhD thesis。它使用 Haskell 编译器 GHC 作为前端，对 CLaSH 代码进行静态分析，直接生成 verilog/vhcl/systemverilog 代码。选择 CLaSH 作为研究对象，是因为相比前面提到的 BlueSpec 和 Lava。它保留了更多的 Haskell 语义。

研究目的与研究方法

本次研究主要是希望能够回答以下两个问题

1. 函数式语言能否做到对电路的优雅的抽象，即“能不能”的问题。
2. 使用函数式语言进行电路设计，相比传统的硬件描述语言，具有那些优势与不足，即“好不好”的问题。

而本次研究采用的方法，是通过选择一门函数式硬件描述语言，实现一个特定的算法，通过走一遍电路设计-仿真的流程，了解函数式硬件语言的特点，判断其优势与不足。

Haskell 语言介绍

本节并不打算全面的介绍 Haskell 语言，有兴趣读者，可以阅读参考文献[3],[4]

本节将通过一个简单的阶乘函数，来介绍 Haskell 中的函数定义、尾递归优化、函数部分应用，类型系统与高阶函数。

下面是阶乘函数在 Haskell 中的几种实现方式：

```
fac0 n = if n <= 1
            then 1
            else n * fac0 (n-1)

fac1 n | n == 0 = 1
       | n == 1 = 1
       | otherwise = n * fac1 (n-1)

fac2 0 = 1
fac2 1 = 1
fac2 n = n * fac2 (n-1)

fac = fac' 1
  where fac' :: Int->Int->Int
        fac' result 0 = result
        fac' result n = fac' (result * n) (n-1)
```

上面是一个阶乘函数的几种定义方式，其中 `fac0` 采用的是 `if-else-then` 的语法，`fac1` 的语法称为 “guard”，类似于 switch-case 的功能，而 `fac2` 则采用了模式匹配的语法。之后实践中发现，模式匹配的语法非常适合写时序电路的状态机。

上面的几种实现方式中，`fac0`，`fac1`，`fac2` 的实现，都是很直接的阶乘函数的定义，写法上也非常直观。

而最后一个 `fac` 函数需要特别讲解，首先可用看出前面几个阶乘函数都是传统的 `fac n = n * fac (n-1)` 的递归形式，而这种递归方式的一个问题，就是递归带来的栈溢出问题，尤其是当需要将其综合为电路时，这种写法，是无法在编译时确定这样一个函数所需要的硬件资源的。

而最下面的 fac 函数则解决了这个问题，首先看 `fac'` 函数，这个函数的递归部分是 `fac' result n = fac' (result * n) (n-1)`，容易看出，这个函数进入下一轮递归时，并不需要保留上一次递归的信息，因而可用被编译器优化为占用常量空间的迭代。

再看 `fac` 函数，其主体为 `fac = fac' 1`，即 `fac'` 函数只给出第一个参数。haskell 支持函数的部分调用，这一特性称之为"curry"。在 haskell 中，其实所有的函数都是单输入函数。例如，`fac'` 的类型签名为 `fac' :: Int->Int->Int`，这可以理解为“输入为两个整数，输出为整

数”的函数，也可以理解为“输入为一个整数，输出为一个‘输入整数，输出整数的函数’”的函数。因此，`fac' 1` 会返回一个 `Int->Int` 类型的函数。

我们再看上面几个 `fac` 函数，查看上述几个 `fac` 函数的类型，结果如下

```
fac0 :: (Ord a, Num a) => a -> a
fac1 :: (Eq a, Num a) => a -> a
fac2 :: (Eq a, Num a) => a -> a
fac  :: Int -> Int
```

可以看到，`fac0` 的类型签名为 `fac0 :: (Ord a, Num a) => a -> a`，其中 `a` 为类型变量，代表不确定的类型，而 `(Ord a, Num a)` 则为类型约束，`Ord`, `Num` 为类型类，可以理解为接口，分别表示“可以比较大小的”和“是数字的”，对于一个没有显式的指明类型的函数，haskell 会根据函数的定义自动推导其类型。 `fac0` 的定义中出现了 `n<1` 这样的表达式，同时使用了加法，因而 haskell 的编译器认定该函数的输入必须可用执行数字运算，同时必须可用比较大小。

`fac1` 与 `fac2` 的函数签名为 `(Eq a, Num a) => a -> a`，`Eq` 也是类型类，可用理解为“可用比较是否相等的”，因为它们的函数定义中没有比较大小操作，但是有 `n == 0` 这样的判断相等的操作，因而被推导为输入必须可用比较相等。`Ord` 是 `Eq` 的扩展，即如果可以比较大小，必定能够比较相等。

而 `fac` 函数的类型签名则为 `fac :: Int->Int`，即确定的由整数到整数，这是因为 `fac` 调用了函数 `fac'`，而 `fac'` 被显式的声明为 `Int->Int`，则这种类型约束会依着调用链传播。

Haskell 中自动类型推导的基础是 Roger Hindley 与 Robin Miller 各自独立设计的 Hindley-Miller 算法，该类型推导算法已经被证明是完备的。

Haskell 是一门强类型语言，并且禁止隐式类型转换。严格的类型系统可以在编译期检查出许多错误，同时完备的类型推导系统可以使程序员不用显式的声明每个函数的类型。

除了常见的 Char, Double, Float, Int 等数据类型，Haskell 还支持扩展的代数数据类型。下面介绍两个常见的扩展类型 `Maybe` 与 `Either`。

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

其中，`Maybe` 用来表示可能失败的运算。当计算失败时，将值赋为 `Nothing` ，而正常的值则用 `Just` 这个构造器封装。我们用 `Maybe` 来改造上面的阶乘函数，使当输入小于 0 时，运算失败，代码如下图：

```
fac :: Int->Maybe Int
fac n | n <  0 = Nothing
      | n <= 1 = Just 1
      | otherwise = case fac (n-1) of
                         Nothing -> Nothing
                         Just v  -> Just (n * v)
      -- | otherwise = fac (n-1) >>= return . (*n)
```

则当输入为负数时：

```
*Main> fac 3
Just 6
*Main> fac 4
Just 24
*Main> fac (-1)
Nothing
*Main>
```

用 Maybe 类型可以用来抽象会出错的运算，而 Either 则可以用来抽象更强大的错误处理功能。例如，我们可以使用 Either 类型来进行异常处理。用上面的 fac 函数为例，我们希望，当输入为负数时，运算出错，同时，因为输出为整数，我们希望当运算结果会导致整型溢出时，运算同样出错。使用 Either 改造 fac 函数：

```
data FacError = NegInput
              | IntOverflow deriving(Show,Eq)
fac :: Int -> Either FacError Int
fac n
  | n <  0 = Left NegInput
  | n <= 1 = Right 1
  | otherwise = fac (n-1) >>= calcNext
    where calcNext val
            | val > maxBound `div` n = Left IntOverflow
            | otherwise              = Right (n * val)
```

我们用这个函数，对从-5 到 25 的整数应用 fac 函数，结果如下：

```
*Main> mapM_ print $ map fac [(-5),(-3)..25]
Left NegInput
Left NegInput
Left NegInput
Right 1
Right 6
Right 120
Right 5040
Right 362880
Right 39916800
Right 6227020800
Right 1307674368000
Right 355687428096000
Right 121645100408832000
Left IntOverflow
Left IntOverflow
Left IntOverflow
*Main> maxBound :: Int
9223372036854775807
*Main>
```

我们首先定义了一个自定义类型 FacError，可用认为它是一个枚举类型，代表阶乘函数中可能出现的两种问题，输入为负数或者输出整型溢出。然后将函数的输出类型定义为 Either FacError Int，即，若运算正常，则返回 Int 值，否则，返回相关错误信息。

上面的 fac 函数用到了 >>= 这个操作符，这是一个多态函数，其类型为 (>>=) :: Monad m => m a -> (a -> m b) -> m b。我们这里不介绍 Monad 的概念，只给出在 fac 函数中其对 Either 类型的定义：

```
(>>=) :: (Either FacError Int)
      -> (Int -> Either FacError Int)
      -> Either FacError Int
Left err >>= _ = Left err -- 如果第一个参数为出错的值，则整个运算的结果为该出错值
Right v  >>= f = f v -- 如果第一个参数为正常值，则将其值v取出，继续运算，返回f v的结果。
```

(>>=) 可以用来串联多个对 Either 类型的操作，如 val >>= func1 >>= func2 只要其中一个出错，则整个调用链的返回值就是该错误值。

同时，看上面 (>>=) 的类型签名，可以看出它接受两个参数，第一个参数为一个 Either FacError Int 类型的值，而第二参数，则是一个函数。是一个类型为输入为 Int，输出为 Either FacError Int 的函数。也就是说，(>>=) 是一个高阶函数。

上面介绍的两个类型，Maybe 和 Either，在 CLaSH 中都是支持的。在本 project 中，Maybe 常常用来充当使能信号。即当运算尚未完成时输出为 Nothing，运算完成时输出为 Just val，而 Either 用来表示状态机中的出错状态，如优先队列容量溢出，或者对一个空队列执行 Pop 操作。

以上只是一些 Haskell 中基本概念的介绍，一些其他概念，如惰性求值的机制，Functor，Applicative Functor，Monoid，Monad 等抽象机制，都是 Haskell 中十分强有力的工具，例如，Monoid 可用来抽象诸如加法、乘法、字符串拼接、二叉树 merge 这些满足结

合律的操作，Monad 可用用来抽象各种带有副作用的计算。这些工具都能够被接下来要介绍的 CLaSH 所支持。不过限于篇幅，本文中不再介绍。

CLaSH 硬件描述语言

CLaSH 是荷兰屯特大学 CAES(Computer Architecture for Embedded System) group 的一个研究项目。该 project 的主要作者是 Christiaan.Baaij，为该组的 PhD。本次研究最主要的参考文献，就是他于今年三月份发表的 PhD thesis。

CLaSH 语言是 Haskell 的一个可综合的子集，上一节介绍的 Haskell 特性，如高阶函数，自定义类型，函数部分应用，模式匹配的语法等，CLaSH 都能够支持，不曾介绍的特性诸如 Monad 等，也能够支持。

不能够支持的特性有：

- 递归函数，不过在将来的版本中可能会支持。
- 递归数据类型
- 浮点数操作，不过能够支持定点小数以及有理数操作
- 在 CLaSH 中，一个函数会确定性的综合为一个电路。函数调用则对应一个电路的实例化。一个普通的函数，会被综合成一个组合逻辑电路。而时序逻辑，则需要其输入与输出均为 `Signal` 类型，下面分别介绍 CLaSH 中的组合逻辑与时序逻辑。

用 CLaSH 设计组合逻辑

首先，在 CLaSH 中，函数类型推导是能够支持的，不过，最顶层的电路，在 CLaSH 中，用 `topEntity` 来表示，必须有确定的类型。例如，在下面的代码中，我们实现了一个加法器 `add`，并用这个 `add` 分别实现一个整数的加法器和一个 5 位无符号数加法器。然后我们又希望实现一个复数的加法器，然而 CLaSH 中并没有内置的复数类型，因此，我们自己定义一个 `Complex` 类型，并重载其加法操作 `(+)`，然后直接使用 `add` 实现其加法器。最终，将顶层电路设置为实部与虚部均为 4 为无符号整数的加法器。

```
module Adder where
import CLaSH.Prelude

add x y = x + y

addInt       = add :: Int->Int->Int
addUnsigned5 = add :: Unsigned 5 -> Unsigned 5 -> Unsigned 5

data Complex a= Complex a a

-- 如果一个Complex是一个Num，则必须其实部与虚部均为Num
instance (Num a) => Num (Complex a) where
    (Complex x1 y1) + (Complex x2 y2) = Complex (x1+x2) (y1+y2)
    (-)         = undefined  -- 将一个类型实现完整的Num接口，需要实现这里的所有函数。
    (*)         = undefined  -- 因为仅作实例，所以暂时不实现减法，乘法等操作
    abs         = undefined
    signum      = undefined
    fromInteger = undefined

addComplex :: Complex (Unsigned 4) -> Complex (Unsigned 4) -> Complex (Unsigned 4)
addComplex = add
topEntity = addComplex
```

用 CLaSH 设计时序电路

目前，CLaSH 只支持同步电路设计。对于同步电路，其输入与输出必须均为 `Signal a`，例如输入与输出均为 `Int` 的电路，其对应表示函数的函数签名为 `Signal Int -> Signal Int`，Signal 类型，在底层其实是一个 stream，即无穷列表。它表示的是从上电，或者按下复位键开始，每个 clock cycle 上的信号值的序列。 对于一个同步时序电路，CLaSH 提供了几种灵活的表示方式，下面主要介绍两种，至于其他的表示方式，如使用 Applicative 函子，使用 State Monad 等等，因为涉及一些 Haskell 中比较深入的概念，且并没有在这次 project 中用到，不再赘述。

直接的 Num 操作

Signal 类型实现了 Num 接口，因此，当时序电路的输入与输出之间，有明确的数学关系时，可用直接使用数字操作，例如，上面提到的加法器，如果将其实现为数字电路。则可以直接写作：

```
add x y = x + y
addSig = add :: Signal Int -> Signal Int
```

这种表示方式，对于进行数字信号处理，十分方便，例如，一个 FIR 滤波器，可以如下实现：

```
module FIR where

import CLaSH.Prelude

dotp :: SaturatingNum a
     => Vec n a
     -> Vec n a
     -> a
dotp as bs = foldl boundedPlus 0 (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 $(v [2::Signal (Signed 16),3,-2,8])
```

其中， `dotp` 为自定义的点乘操作，`window` 为内置的库函数，可以对信号执行加窗操作。最后，将顶层函数 topEntity 实现为一个输入与输出均为 16 位有符号数的电路。

下一个例子是一个斐波那契数列。 我们直接从它的递归定义入手，给出一个斐波那契数列发生器，注意 CLaSH 虽然不支持递归函数调用，但是对于 `Signal` 类型，对值的递归(recursion on value)仍然是支持的，我们可以用它表示下一个 clock cycle 的信号与上一个 clock cycle 信号的关系。

斐波那契数列的递归定义是：

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

我们假定已经有了一个斐波那契序列 `fib`： {1,1,2,3,5...}，那么，`fib(n-1)` 可以理解为 `fib` 序列延迟一个 clock cycle，而 `fib(n-2)` 则是延迟两个周期的 `fib`。

若 `fib` 为 {1,1,2,3,5}，则:

```
delay1 = {1, fib} = {1} + {1,1,2,3,5..} = {1,1,1,2,3,5...}
delay2 = {0,0,fib} = {0,0} + {1,1,2,3,5...} = {0,0,1,1,2,3,5...}
delay1 + delay2 = {1,1,2,3,5} = {1,1,2,3,5,8,13...} == fib
```

我们可以根据上面的 `fib` 序列的定义， 直接给出一个斐波那契序列发生器：

```
fib :: Signal Int -- 这里不考虑整型溢出的问题
fib = register 1 fib + (register 0 $ register 0 fib)
-- register函数接受一个值和一个Signal信号，用这个值把信号延迟一个周期，生成一个新的信号。
```

我们可用看一下这段代码生成的 verilog 代码，来看下它生成的代码质量

```
// Automatically generated Verilog-2005
module Fib_fib_1(// clock
                 system1000
                 ,// asynchronous reset: active low
                 system1000_rstn
                 ,bodyVar_o);
  input system1000;
  input system1000_rstn;
  output signed [31:0] bodyVar_o;
  wire signed [31:0] bodyVar_0;
  wire signed [31:0] Fibzhfib2_1;
  wire signed [31:0] bodyVar_o_sig;
  wire signed [31:0] Fibzhfib5_2;
  wire signed [31:0] x_3;
  wire signed [31:0] altLet_4;
  wire signed [31:0] y_5;
  wire signed [31:0] altLet_6;
  wire signed [31:0] repANF_7;
  wire signed [31:0] tmp_8;
  wire signed [31:0] tmp_11;
  wire signed [31:0] tmp_14;
```

```
  // register begin
  reg signed [31:0] n_9;
  always @(posedge system1000 or negedge system1000_rstn) begin : register_Fib_fib_1_n_10
    if (~ system1000_rstn) begin
      n_9 <= 32'sd0;
    end else begin
      n_9 <= Fibzhfib2_1;
    end
  end
  assign tmp_8 = n_9;
  // register end
  assign bodyVar_0 = tmp_8;
  // register begin
  reg signed [31:0] n_12;
  always @(posedge system1000 or negedge system1000_rstn) begin : register_Fib_fib_1_n_13
    if (~ system1000_rstn) begin
      n_12 <= 32'sd0;
    end else begin
      n_12 <= bodyVar_o_sig;
    end
  end
  assign tmp_11 = n_12;
  // register end
  assign Fibzhfib2_1 = tmp_11;
  assign bodyVar_o_sig = altLet_4;
  // register begin
  reg signed [31:0] n_15;
  always @(posedge system1000 or negedge system1000_rstn) begin : register_Fib_fib_1_n_16
    if (~ system1000_rstn) begin
      n_15 <= 32'sd1;
    end else begin
      n_15 <= bodyVar_o_sig;
    end
  end
  assign tmp_14      = n_15;
  // register end
  assign Fibzhfib5_2 = tmp_14;
  assign x_3         = Fibzhfib5_2;
  assign altLet_4    = altLet_6;
  assign y_5         = bodyVar_0;
  assign altLet_6    = repANF_7;
  assign repANF_7    = x_3 + y_5;
  assign bodyVar_o   = bodyVar_o_sig;
endmodule
```

可以看到，生成的代码质量还是很高的。 CLaSH 代码中用了一次加法，三个 `register`，而生成的 verilog 代码也基本上与之意义对应，包含三个 `reg signed[31:0]` 变量，和一次加法操作。

一个问题是它似乎会生成大量的冗余 assign 语句，诸如 `y = f(x)` 的逻辑，它会倾向于生成这种风格的代码：

```
assign tmp1 = x;
assign tmp2 = tmp1;
assign tmp3 = f(tmp2);
assign tmp4 = tmp3;
assign y = tmp4;
```

不过，这种代码在生成具体电路时，并不会带来冗余的开销，只是会影响到生成 verilog 代码的可读性。

通过米利模型，摩尔模型的高阶函数来实现时序电路

我们知道，同步时序电路总有三个要素：输入，状态，输出。而同步时序电路其实就是在描述这三者的关系。有两种描述模型，米利模型与摩尔模型，其中，米利模型是说，旧的状态与新的输入决定新的状态，同时也决定新的输出。而摩尔模型是说，旧的状态与新的输入，决定新的状态。而输出则由状态唯一的决定。CLaSH 中，提供两个高阶函数 `mealy` 与 `moore`，对于输入与输出之间关系难以用数学表达式描述的电路，我们可以通过这两个高阶函数来描述电路的状态转换关系。

对这两个函数其实不用多说，它们的类型签名就是它们的文档，我们来看下这两个函数的类型签名：

```
CLaSH.Prelude> :t moore
moore :: (s -> i -> s) -> (s -> o) -> s -> Signal i -> Signal o
CLaSH.Prelude>
CLaSH.Prelude> :t mealy
mealy :: (s -> i -> (s, o)) -> s -> Signal i -> Signal o
```

其中，`s` 代表状态的类型，`i` 代表输入的类型，`o` 代表输出的类型。`moore` 函数的前三个参数的类型分别为 `(s->i->s)`，`(s->o)`, `s`，即要求使用者给出一个根据状态与输入决定新状态的函数，一个由状态决定输出的函数，以及一个初始状态代表按下复位键时系统的状态。然后 `moore` 函数就会根据这三个函数自动的生成状态机。

`mealy` 函数与之类似，要求用户提供一个根据状态与输入决定下一状态与输出的函数，一个初始状态，就会自动生成一个对应的同步时序电路。

基于 CLaSH 的优先队列实现

本节介绍基于 CLaSH 的优先队列实现，首先给出优先队列(Priority Queue)抽象数据类型(Abstract Data Type，ADT)的描述，然后介绍针对优先队列定义的 Haskell 数据结构，然后简介基于 CLaSH 的优先队列的实现方案。

设计指标：

- Push 与 Pop 操作复杂度均在 O(lgN)
- top 操作的复杂度为 O(1)
- 应当具有使能信号，当优先队列在执行 Push 或者 Pop 操作时，top 值置为 invalid
- 当对满队列执行 Push 操作，或者对空队列执行 Pop 操作时，要能够报错
- 只有当队列没有在执行 Push 或 Pop 操作时，输入端的操作指令才有效。
- 同时支持最大堆、最小堆
- 能够支持不同种数据类型，例如，一个元素为整数的优先队列和一个元素为自定义的图节点数据结构的优先队列，应该能够用同一个函数(电路结构)来描述。
- 时间关系，暂时不支持清空队列的 Clear 操作。

我们知道，队列(Queue)是一种常见的线性表，具有先进先出(FIFO)的特点，队列只允许在后端进行插入(Push)操作，而在队列顶端进行访问(Top)与删除(Pop)操作，队列保证在顶端被访问到的一定是最先被插入队列的元素。

优先队列也是只能在尾端进行插入，在前端进行访问与删除，与队列不同的是，优先队列中每个元素有一个附带的、可比较大小的优先级(Priority)。当对优先队列执行 Pop 操作时，一定是优先级最高(或最低)的那个元素，而不是最先或者最后被插入的元素首先被 Pop 出来。

事实上，栈(Stack)和队列都可以用优先队列来表示，如果我们用元素的插入时间来作为元素的优先级，则若优先队列顶部为优先级(插入时间)最小，则为 FIFO 队列，若为优先级最大元素，则为 FILO 的栈。

一个很直观的优先队列实现方案，是维护一个按照优先级排好序的线性表，但这样子的话，Pop 操作固然可以在 O(1) 时间完成(队列顶部指针减一即可)，但是 Push 操作会需要 O(N) 的复杂度。我们使用二叉堆(Binary Heap)来实现优先队列。

二叉堆是这样一种数据结构：

1. 它是一颗完全二叉树(除最后一层外，其余每一层节点数都是最大节点数)
2. 父节点的优先级总是大于(或小于)子节点的优先级。若父节点优先级大于子节点，则为最大堆；否则，则为最小堆。

我们以最小堆为例讲解优先队列的实现。

因为二叉堆是完全二叉树，因此它可以存储在数组中，按照二叉树树的广度优先(Breadth First)遍历将数组存储在数组中，则对于元素 `arr[i]`，父节点必定为 `arr[(i-1)/2]`，两个子节点分别为 `arr[2*i+1]` 与 `arr[2*i+2]` 。在本次设计中，使用长度为 n 的寄存器数组实现一个最大容量为 n 的优先队列。

对于二叉堆上的 Push 操作，首先将元素插到二叉树的尾端，然后比较其与父节点的大小，若大于等于父节点，则算法完成，否则，将该节点与父节点交换，并重复此过程，直到该元素大于等于父节点，或者该元素位于堆顶。

对于 Pop 操作，首先将堆顶元素与堆底元素交换，然后将堆的 size 减一，指针(用一个存储数组下标的寄存器实现)，指向堆顶，接下来从指针的两个子节点中选择优先级较小的与之比较，若新的该子节点小于父节点，则进行交换，并将指针指向该子节点。重复此过程，直到指针指向的节点小于两个子节点，或者指针位于二叉树的叶节点。

对于这样的实现，Push 与 Pop 操作的复杂度均为 O(lgN)。

对于 top 操作，则如果优先队列没有在执行 pop 或者 push 操作，则返回堆顶元素，否则，返回 invalid。

```
data Output a = Out QueueStatus (Maybe a)   deriving(Show)
data Input a  = Push a | Pop | Nop          deriving(Show)
data NormalStatus   = Pushing  | Popping | Ready deriving(Show,Eq)
data ErrorStatus    = Overflow | Empty deriving(Show,Eq)
type QueueStatus    = Either ErrorStatus NormalStatus
data InnerState n a = S {
    status :: QueueStatus,
    size   :: Size,
    idx    :: Size,
    queue  :: Vec n a
} deriving(Show)
```

上面是使用 CLaSH 定义的相关数据结构。

首先我们定义了这个优先队列的输入与输出数据结构。输入为相关指令集，其中 Pop 与 Nop 都不带操作数，而 Push 则带有操作数，在这里我们用类型变量 "a" 来表示 Push 进优先队列的数据类型，而不指定具体类型，从而使该优先队列能够支持任意可比较大小的数据类型，包括自定义数据类型。

优先队列的输出是堆顶元素，类型为 "a"，但是我们希望在队列执行 Push 与 Pop 操作时，top 的值无效，因此我们将输出的 top 的类型设为 Maybe a，即当 top 无效时，返回 Nothing，否则，返回 Just a。除了堆顶元素的值，还输出了当前队列状态，使用者可以通过这个值判断当前队列是否空闲，或者是否出错。

接下来我们定义了三个正常的状态 Pushing, Popping 与 Ready，分别表示优先队列正在执行 Push 或者 Pop 操作，或者目前没有在执行任何操作。

然后定义了两个错误状态，Overflow 表示使用者试图对一个满队列执行 Push 操作，Empty 表示使用者试图对一个空队列执行 Pop 操作。

我们将队列的操作状态定义为 Either ErrorStatus NormalStatus，即它可能出于出错的 Errorstatus 状态，也可能出于正常的 Normalstatus 状态。

`Innerstate` 表示优先队列的内部状态，其中 `status` 为其操作状态，`size` 为其当前大小，`idx` 为执行 `push` 与 `pop` 操作时用到的指针，而 `queue` 是一个容量为 n 的寄存器数组。 用来存储队列元素。这些状态中，只有 `status` 是被导出到输出的，其他状态均是外部不可见。

使用摩尔模型来表示优先队列的状态机，其顶层描述如下

```
priorityQueueS :: (KnownNat (n+1) , Ord a, Show a)
               => Ordering             -- determine max_queue or min_queue
               -> InnerState (n+1) a   -- state
               -> Input a              -- input
               -> InnerState (n+1) a   -- (state, output)
-- 内部为出错状态时，保持出错状态
priorityQueueS _   st@(S (Left _)        _ _ _)  qIn@_          = st
-- 内部Ready，但是输入为Nop，保持当前状态
priorityQueueS _   st@(S (Right Ready)   _ _ _)  qIn@Nop        = st
-- 内部为Ready，收到Pop指令，进入Popping状态
priorityQueueS _   st@(S (Right Ready)   _ _ _)  qIn@Pop        = initPop  st
-- 内部为Ready，收到Push指令，进入Pushing状态
priorityQueueS _   st@(S (Right Ready)   _ _ _)  qIn@(Push val) = initPush st val
-- 内部状态Pushing，执行Push循环操作
priorityQueueS ord st@(S (Right Pushing) _ _ _)  qIn@_          = processPush ord st
-- 内部状态为Popping，执行Pop循环操作
priorityQueueS ord st@(S (Right Popping) _ _ _)  qIn@_          = processPop  ord st
```

上面的顶层代码，描述的是摩尔模型中的状态方程，即根据状态与输入，决定下一个状态。而输出方程则比较简单，判断队列当前状态是否为 `Ready`，如果是，则返回堆顶元素即可。

当描述状态机时，haskell 的模式匹配的语法提供了很大的便利。

注意到，上面的函数是个多态函数，即任何实现了比较大小操作(Ord)类型，都可以作为优先队列的元素。

上面函数的第一个参数，是一个 `Ordering` 类型的值。Ordering 表示元素比较大小的结果，定义为 `data Ordering = LT | EQ | GT`，分别表示小于，大于和等于，通过对 `priorityQueueS` 的部分应用，我们可以得到最大堆与最小堆：

```
minQS = priorityQueueS LT
maxQS = priorityQueueS GT
```

基于 CLaSH 的堆排序算法实现

因为优先队列总是能够在堆顶给出最小(最大)优先级的元素，因此，我们可以用优先队列来进行排序，并且因为优先队列的 Push 与 Pop 操作的复杂度均为 O(lgN)，整个排序算法的复杂度是 O(NlgN)。并且这个算法的最坏复杂度是有保证的，不会像 quick sort 算法那样，在最坏情况下复杂度可能退化为 O(N^2)。

对排序的思路很简单，首先，将待排序的元素一个一个的 Push 进入优先队列，然后再依次 Pop，直到队列为空，则元素 Pop 出的顺序必定是按照排好的顺序。

堆排序电路的实现，采用并行输入，并行输出。对于一个 n 个元素的排序电路，输入与输出均为长度为 n 的向量。整个电路，包括一个之前已经实现的，实例化为元素类型为整数的优先队列，和一个堆排序的控制逻辑，用来根据优先队列的状态决定下一步的操作。

堆排序控制逻辑的基本状态转换关系为：

- 当尚有元素没有插入优先队列时，准备执行 Push 操作。
- 当所有元素都插入优先队列后，对优先队列执行 Pop 操作。
- 当优先队列处在 Pushing 或者 Popping 状态是，进行数据准备工作，而对优先队列的操作为 Nop。
- 当优先队列处在 Ready 状态是，对其进行 Push 或者 Pop 操作。
- 当优先队列处在错误状态时，控制逻辑也进入错误状态。

- 控制逻辑有两个输入，一个是外部输入的待排序向量，一个是优先队列的输出。

- 控制逻辑有两个输出，一个是向优先队列输入的指令集，一个是向外部输出的排好序的向量，仍然设定为 Maybe 类型，即当尚未排好序时，输出 Nothing， 排好序后，输出 Just vector。

下面是优先队列实现时自定义的数据结构。用来表示其控制逻辑的内部状态。

```
data HeapSortState n a = HSS{
    vec       :: Vec n a,
    sortState :: SortInnerState
} deriving(Show)
data SortInnerState = SPush Size -- 表示现在在向优先队列插入元素
                    | SPop  Size -- 表示现在从优先队列Pop元素
                    | SError     -- 表示电路出错
                    | Sorted deriving(Show) -- 表示元素已经排好序，可以输出了
```

堆排序控制逻辑的类型签名如下，可以看出，控制逻辑仍然采用了摩尔模型：

```
-- 堆排序控制逻辑

-- 状态方程，根据状态和输入，决定下一个状态
heapSortS :: (KnownNat (n+1), Ord a, Default a, Show a)
          => HeapSortState (n+1) a       -- inner state
          -> (Output a, Maybe (Vec n a)) -- input vector and the output of priorityQueue
          -> HeapSortState (n+1) a       -- new state
-- 输出方程，根据状态，决定控制逻辑的输出
heapSortO :: (KnownNat (n+1), Ord a, Show a)
           => HeapSortState (n+1) a
           -> (Input a, Maybe (Vec n a))
```

基于 CLaSH 的最短路径算法实现

A*算法描述

问题描述：给定一张图(Graph) {Vertex,Edge}，并且给出图上的两个节点 start 与 end，希望找出从 start 到 end 的最短路径。

具体到本 project，将图简化为一个二维的网格，网格具有 X 和 Y 两个 维度。start 与 end 就是这个二维网格矩阵上的两个网格。每个网格有一个 `isObstacle` 属性，用来表示它是一个可用通过的点，还是一个障碍物。

这样，一张图可以用存储在内存中的一维数组来表示。并且给出图上一个点，就可以知道它上下左右四个邻居点的坐标。图上一个节点的数据结构定义如下：

```
data Node = Node{
    idx :: Addr,
    prev :: Addr,
    g :: Dist,
    h :: Dist,
    isObstacle :: Bool
} deriving(Show)
```

其中 `idx` 表示节点的数组下标，`prev` 表示最短路径上的上一个节点的数组下标。`g` 与 `h` 为用来计算路径长度的启发式函数的值，`g` 表示当前该节点到 `start` 节点的最短路径的值， `h` 表示当前节点到 `end` 节点的估计值， `isobstacle` 用来表示这个点是不是一个可以通过的点。

A* 算法是一个寻找最短路径的启发式算法，首先我们需要提供一个启发式函数 `heuristic`，该函数以图上一个节点为输入，应该给出它到重点 `end` 的最短路径长度的估计值，只要这个估计函数不会超过实际最短路径长度，A* 算法就能够保证找出的路径为最短路径。

对于这个二维网格上的最短路径问题，我们节点到终点的曼哈顿距离作为启发式函数。原因有二：

1. 曼哈顿距离为两个节点二维数组下标的差求和：`heuristic = |x1-x2| + |y1-y2|`，这样，它的结果总是整数，其他启发式函数，比如欧几里德距离，结果可能不是整数，必须用定点小数近似。
2. 如果图上没有障碍物，则最短距离就是曼哈顿距离，因而曼哈顿距离作为最短路径的估计是合理的。

A*算法的描述如下

1. 初始状态，所有节点的 g 均设为无穷大，因为 g 是整数，无法表示无穷大，所以设为 (maxBound / 2)，即能表示的最大整数除以 2， 之所以除以二，是防止对 g 相加时溢出。
2. 将 start 节点的 g 设置为 0(start 到 start 的距离当然是 0)，h 设置为 start 到 end 距离的估计值.
3. 建立优先队列，优先队列以图上节点， `Node` 类型为存储元素，而优先级设定为 `g + h`。
4. 将 start 节点 push 进入优先队列
5. 若队列非空，或者 top 节点不是 end 节点，进入循环
   - Pop 出优先队列的 top 元素 u，并从内存中读入它的所有邻居元素。
   - 对于 u 的每个邻居 v: 如果 v 不是障碍物，并且满足 `g(v)> g(u) + 1`，则将 v 的 `g` 更新为 `g(u) + 1`，将 v 的 prev 更新为 u, 将更新的 v 写回内存，并将更新的 v push 进入优先队列
   - 进入下一轮循环。直到算法终止。

A* 算法已经被证明，只要启发函数的估计值小于等于实际值，则必定能够求出最短路径，当用返回 0 的常数函数作为启发式函数时，A* 算法退化为 Dikjstra 算法。而在网格上的 Dikjstra 算法，其实就是在做广度优先搜索。一般来说，如果只是求两点最短路径，A* 算法的性能要好于 Dikjstra 算法。不过 Dikjstra 算法的优势在于它可用来求出从一个点到图上所有点的最短路径。

具体到电路实现上，我们使用一个 block ram 来存储图的节点，用一个控制逻辑，连接外部输入、blockRam 和优先队列。

A*算法实现

下面给出在控制逻辑中的自定义数据类型，分别表示外部输入，内部状态，以及输出。

```
type Addr = Unsigned 16
type Size = Unsigned 16
type Dist = Unsigned 16
data Info = In{ xMax  :: Size, yMax  :: Size, start :: Addr, end   :: Addr } deriving(Show)
data PathState   = PS {
    memDelayCounter:: Unsigned 4,
    info :: Info,
    innerState :: PathFinderState ,
    uNode :: Node,
    vNodes :: Vec 4 Node,
    upCounter :: Unsigned 2
} deriving(Show)
data PathFinderState = PWaiting
    | PInitMem              -- read start from memory
    | PInitQ                -- push start to priority queue
    | PCheck
    | PProcessNeighbor
    | PPushNeighbor
    | PFinished
    | PError Error deriving(Show,Eq)
data Error     = QOverflow | NoPath | Unknown deriving(Show,Eq)
data PathOut   = POut {
    qIn :: (Q.Input Node), -- this output would be used as input of queue
    rAddr :: Addr,
    wAddr ::Addr,
    wEn ::Bool,
    wVal:: Node,
    pathNode :: Maybe Node
} deriving(Show)
type PathInput = (Q.Output Node, Maybe Info, Node)
```

而控制逻辑采用米利模型实现，其函数类型签名为：

```
aStarCtrl :: (Node -> Info -> Dist)
          -> PathState                      -- input state
          -> PathInput
          -> (PathState, PathOut)

aStarCtrlLogic    = (aStarCtrl heurAStar) `mealy` def
dikjstraCtrlLogic = (aStarCtrl heurDikj) `mealy` def
```

`aStarCtrl` 是一个可配置的高阶函数，第一个参数，是根据输入 Node 确定该 Node 到终点最短路径长度估计值的启发式函数，我们可以自己定义不同的启发式函数，例如，上面的 `aStarCtrlLogic` 与 `dikjstraCtrlLogic`，就是用 `aStarCtrl` 分别用不同的启发式函数配置而成。

仿真结果

我们首先检验 CLaSH 代码的正确性。

对于 heapSort，我们以逆序排列的向量作为输入，期望的输出是升序排列的数组，首先用 (5,4,3,2,1) 作为输入，测试其正确性，在 CLaSH 的 simulator 中，仿真结果如下：

```
*HeapSort> samp 32
(Just <5,4,3,2,1>,Just <0,0,0,0,0>)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Nothing)
(Nothing,Just <1,2,3,4,5>)
(Nothing,Just <1,2,3,4,5>)
*HeapSort>
```

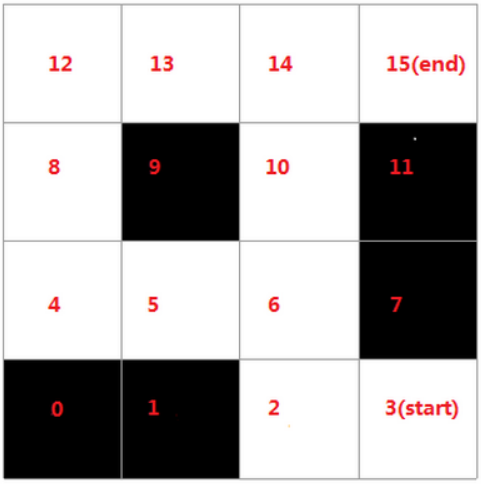
上面的每个序对，表示在每个 clock 上，系统的输入与输出。上面的结果表示，在第一个 clock cycle，输入了向量(5,4,3,2,1)，经过 31 个 clock cycle 之后，输入的向量被正确的排序。 而在向量排好序之前，输出均为 `Nothing`。

接下来，我们检测 heapSort 的算法复杂度，输入不同长度的向量，统计排好序所需要的时钟周期数，结果如下：

input length	clock cycle
100	1264
90	1114
80	964
70	814
60	670
50	540
40	410
30	282
20	172
10	72
5	31
2	12
1	7

可以看到，对于 heapSort 算法，其时间复杂度为 O(lgN)。

对于最短路径算法，我们测试一个 4*4 的网格，共有 16 个节点。 其中 (1，1)，(3，1)，(3，2)，(0，1)，(1，3)节点为障碍，寻找从(3，0)到(3,3)的最短路径， 如下图所示：



CLaSH 输出结果如下：

```
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Nothing
Just (Node {idx = 15, prev = 14, g = 5, h = 0, isObstacle = False})
Just (Node {idx = 14, prev = 10, g = 4, h = 0, isObstacle = False})
Just (Node {idx = 10, prev = 6, g = 3, h = 0, isObstacle = False})
Just (Node {idx = 6, prev = 2, g = 2, h = 0, isObstacle = False})
Just (Node {idx = 2, prev = 3, g = 1, h = 0, isObstacle = False})
Just (Node {idx = 3, prev = 3, g = 0, h = 0, isObstacle = False})
```

上面的结果表示，在第 105 个时钟周期，找到了最短路径(15,14,10,6,2,3)。

生成的 verilog 代码在 ModelSim 上编译通过，不过因为我的 ModelSim License 到期了，未能成功完成仿真。

总结

通过使用 CLaSH 走一遍，对函数式硬件设计的概念有了一些基本的了解。 对于它的优势与不足，总结如下。

优势：

1. 它将函数式语言中的高层次的抽象机制，如高阶函数，类型系统等带入了硬件设计领域，当设计的电路具有一定固有模式(pattern)的时候，这些机制能够极大的避免重复的代码，比如我本次设计的优先队列，堆排序使用的优先队列，是为了进行整数排序，因而其元素为整型，而最短路径算法，则是存储自定义数据结构的优先队列。 而只要我为自己定义的节点数据结构实现了比较操作的接口 `Ord`，之前设计的优先队列就可以直接拿来使用。
2. 强大的类型系统，可以在编译时检查出许多甚至绝大多数错误。
3. 虽然提供了很高层次的抽象，但是并没有降低生成电路的质量，CLaSH 本身并不是一门“算法到电路”的自动综合工具，它仍然是一门“硬件描述语言”，即，CLaSH 的代码与生成的硬件，基本上是一一对应的。

4. 在语义上的不可变状态，带来了隐式的并行性。 相比传统的过程式语言，在语义层面与数字电路更加接近。 函数式语言所谓的“不贴近硬件”，其实只是它不贴近冯诺依曼模型而已。

劣势：

1. 没有在大规模的电路上证明自己
2. 在使用函数式语言时，仍然需要在大脑中有电路的概念。 比如，在用 C 语言在 PC 机上编程，或者用 Haskell 在 PC 机上编程时，即使是多线程环境下，我们对一个优先队列执行 Pop 操作，这个操作对于程序员来说，也是原子性的，我们不会想要在这个执行 Pop 操作的 O(lgN)的时钟周期内做一些事情，下一个动作会发生在 Pop 完成之后，而进行硬件编写时，我们需要考虑的是每个时钟周期上的状态变化，同时还会考虑在 Pop 的执行过程中，其他硬件资源进行的数据准备工作——因为在这里并不会有什么多线程并行粒度太细的问题。 也就是说，它确实能够提升硬件设计者的设计效率，但并不是什么能够让不懂电路设计原理的程序员直接写出高质量电路的工具。
3. 而它同时又要求电路设计者有函数式编程的概念。 函数式编程的学习本身又有一定难度。
4. 对信号延迟的建模能力不够强大。
5. 本身缺乏强大而完整的工具链支持。

而我本人的感觉，上面说的很多缺点，或者说，函数式硬件描述语言，乃至其他非函数式的高层次综合工具不够流行的原因，很大程度上是因为在传统的软硬件开发模式中，程序员就负责开发软件，而电路设计者就负责开发硬件。 即使是贴近底层的嵌入式开发，程序员面对的也是一个被指令集抽象过的冯诺依曼机。 是一个 "ALU + 寄存器 + memory" 的模型。 而函数式语言将来能否受到广泛的应用，很大程度上会取决于近年来兴起的 "CPU + FPGA"的计算模式将来会不会称为主流，不过那是另一个课题了，我对此研究不深，不敢在此妄言。

最后的结论是，函数式硬件描述语言，总的来说，函数式硬件描述语言在语言层面上，可以说对电路提供了一个优雅的抽象。 然而落实到具体的实现上，尚缺乏一个完整而稳定的函数式硬件开发环境。

不过，因为我的研究方向是 CAD，平时主要的开发语言既不是 haskell，也不是 verilog。平时的研究重点也不在数字电路设计上，因此，上述的结论不可避免的会受到自己 Haskell、Verilog、以及对数字电路理解水平的限制，必定带有不少偏见与盲点。

参考文献

1. CLaSH website: www.clash-lang.org
2. CLaSH documentation
3. Lipovaca M. Learn you a haskell for great good!: a beginner's guide[M]. no starch press, 2011.
4. O'Sullivan B, Goerzen J, Stewart D B. Real world haskell: Code you can believe in[M]. " O'Reilly Media, Inc.", 2008.
5. Baaij, C.P.R. (2015) Digital Circuits in CλaSH: Functional Specifications and Type-Directed Synthesis. PhD thesis, University of Twente, Enschede, The Netherlands, January 2015.