MASTER THESIS PROJECT

Implementation of the MUSIC Algorithm in CλaSH

Xiaopeng Jin

Faculty of Electrical Engineering, Mathematics and Computer Science

Supervisors:

Dr.ir. A.B.J. Kokkeler
Dr.ir. J. Kuper
Ir. E. Molenkamp
Dr.ir. J. Broenink

UNIVERSITY OF TWENTE.

Table of Contents

Abstract

CλaSH is a hardware description language based on the functional programming language Haskell. The CλaSH implementation of a hardware design can be translated to synthesizable VHDL code by the CλaSH compiler. The MUSIC algorithm is a classic subspace-based DOA estimation method that performs an eigen-decomposition on the covariance matrix. To achieve real-time performance in practical applications of the MUSIC algorithm, a number of hardware implementations have been developed. In this master project, the MUSIC algorithm is implemented in CλaSH to investigate the advantages and disadvantages of using this language for the hardware implementation of an algorithm. The CλaSH implementation is evaluated in several aspects such as the conciseness of the descriptions, development time and the synthesis result of the generated VHDL code.

1. Introduction

This is the final report of the master thesis project on the implementation of the MUSIC (Multiple Signal Classification) algorithm in CλaSH (CAES Language for Synchronous Hardware).

1.1 Motivation

CλaSH (pronounced as "clash") is a functional hardware description language developed by the CAES (Computer Architecture for Embedded Systems) group at University of Twente. It borrows both the syntax and semantics from the functional programming language Haskell. "Polymorphism and higher-order functions provide a level of abstraction and generality that allow a circuit designer to describe circuits in a more natural way than possible with the language elements found in the traditional hardware description languages."[1] Circuit descriptions can be translated to synthesizable VHDL code by the CλaSH compiler. As CλaSH is a new developed language, it still needs to be evaluated and improved.

DOA (Direction of Arrival) estimation of wireless signals is one of the techniques that is frequently used in smart antenna technology. Smart antennas are used in many fields such as radar, sonar and mobile communications. The MUSIC algorithm estimates the DOA by performing an EVD (eigenvalue decomposition) on the covariance matrix of the signal data. Although MUSIC shows a good performance in DOA estimation, it is achieved at a high cost in computation and storage. To achieve a real-time performance in practical applications, several methods have been proposed to implement MUSIC on hardware.

As MUSIC is a non-trivial algorithm for hardware implementation, it is interesting to use it as a test case of CλaSH. In this project, the MUSIC algorithm is implemented in CλaSH to investigate the advantages and disadvantages of using CλaSH for hardware implementations.

1.2 Methodology

Figure 1 presents the research strategy of this project. First, we have to get familiar with CλaSH language and study the MUSIC algorithm as well as its hardware implementation methods. Then the MUSIC algorithm is implemented in CλaSH according to the hardware designs described in [1]. The CλaSH implementation is evaluated by comparing it with a VHDL implementation in several aspects such as the conciseness of descriptions, namely the amount of code, development time and the synthesis result including maximum clock frequency (Fmax) and the amount of hardware resources. To compare synthesis results, VHDL code was provided by the author of [1]. However, it is likely that the provided VHDL code does not exactly implement the hardware designs described in [1] as its synthesis result turned out to be very different from the results presented in [1], which makes it not comparable with our CλaSH implementation. Therefore, it is decided to make a new VHDL implementation for a small part of the MUSIC algorithm and compare its synthesis result with the result of the corresponding CλaSH implementation. Finally, we can reach a conclusion based on that evaluation.

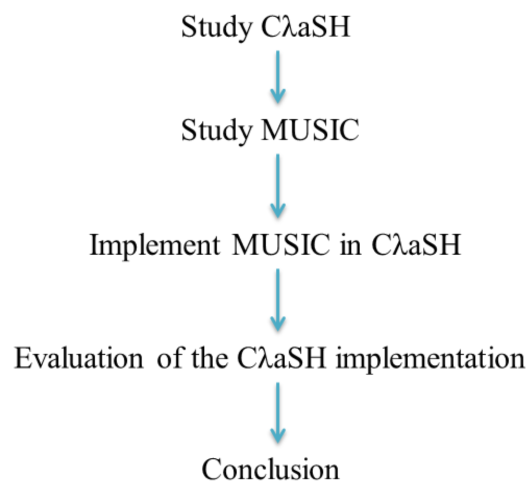

Figure 1. Research strategy

1.3 Report Outline

This report is basically organized according to the research strategy shown in Figure 1. Following the introduction chapter, Chapter 2 is an introduction of the CλaSH language and its compiler. The MUSIC algorithm and its hardware implementation are studied in Chapter 3 and Chapter 4 respectively. Chapter 5 describes how the MUSIC algorithm is implemented in CλaSH and presents the simulation results of the CλaSH implementation. An evaluation of the CλaSH implementation is carried out in Chapter 6. Finally, the conclusions are presented in Chapter 7.

2. CλaSH

2.1 Introduction

Unlike some high-level programming languages, the traditional HDLs (Hardware Description Languages) do not have properties such as function overloading and polymorphism, which makes it cumbersome in expressing higher-level abstractions that are needed for today's large and complex circuit designs. In an attempt to raise the abstraction level, a great number of approaches based on functional languages have been proposed. "Functional languages are especially well suited to describe hardware because combinational circuits can be directly modeled as mathematical functions and functional languages are very good at describing and composing these functions."[2]

CλaSH is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. As a subset of Haskell, CλaSH inherits from Haskell such advanced features as polymorphic typing, user-defined higher-order functions and pattern matching. These features provide great convenience for high-level abstractions and allow circuit specifications to be written in a very concise way. Recursive functions, a crucial aspect of a functional language, are not completely supported by CλaSH yet. CλaSH extends Haskell with some hardware-related elements such as *state* and *vector*. With the support of these elements within the CλaSH compiler, the CλaSH code can be translated to synthesizable VHDL.

2.2 Hardware Description in Haskell

This section introduces the basic language elements of Haskell and describes how they are related to hardware.

2.2.1 Functions

Two basic elements of a functional programming language are functions and function applications. The main reason of using a functional programming language to describe hardware is that a function is conceptually close to a combinational circuit in hardware: both transform input values to output values. The CλaSH compiler translates every function to a component in VHDL, every argument/output to an input/output port, and function applications to component instantiations.

Figure 2 is the block diagram of a half adder which is described as a function called `halfAdd` in Haskell as shown in Listing 1. The `halfAdd` function takes two input arguments `a` and `b` and presents the outputs `sum` and `carry` in a tuple. The `where` clause describes the operations on the input values where `xor` and `and` are predefined functions that perform a bitwise "*exclusive or"* and a bitwise "*and"* operation respectively.

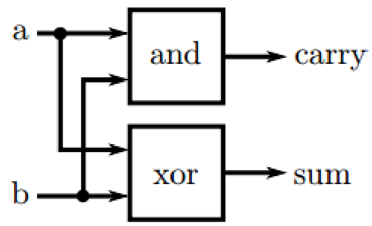

Figure 2. Half adder circuit

```
halfAdd a b = (sum, carry)
     where
        sum = xor a b
        carry = and a b
```

Listing 1. Half adder

A sequential circuit can also be described as a function in Haskell with a basic premise that it is modeled as a Mealy machine to make it a synchronous circuit. There is one implicit global clock affecting all delay components in the circuit. As shown in Figure 3, a Mealy machine consists of combinational logics and memory elements. The output of a Mealy machine in each clock cycle depends on both the input and the content of the memory elements which is also called the current *state*.

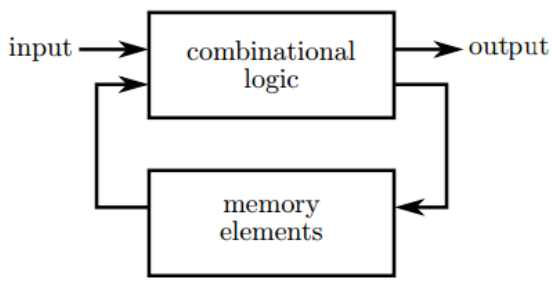

Figure 3. Mealy machine

Figure 4 illustrates the circuit of an accumulator which requires a register to store the intermediate values temporarily. It is described as a function called `acc` in Haskell as shown in Listing 2, where `s` and `s'` denote the old and new state respectively. CλaSH treats the old state as an additional input and the new state as an additional output, while many other functional HDLs model signals as a stream of values over time and state is then modeled as a delay on this stream of values [2]. The synchronous sequential circuits can be simulated by the `simulate` function which will be introduced in Sec. 2.2.6.

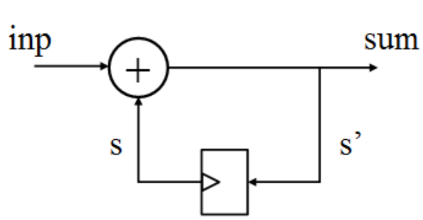

Figure 4. Accumulator circuit

```
acc s inp = (s', sum)
    where
        s' = s + inp
        sum = s'
```

Listing 2. Accumulator in Haskell

2.2.2 Types

"Haskell is a statically-typed language, meaning that the type of a variable or function is determined at compile-time."[2] Not all Haskell constructs have a direct structural counterpart in hardware. For instance, some Haskell types such as `Integer` and `list` cannot be translated into hardware because they do not have a fixed size at compile time. Therefore, CλaSH provides the following built-in types that have a clear correspondence to hardware:

`Bit`: It can be either of the two values: `High` and `Low`, representing the two possible states of a digital device, for instance, a flip-flop.

`Bool`: It is a basic logic type with two possible values: `True` or `False`. It is required in `if-then-else` expressions.

`Signed`, `Unsigned`: They represent the signed and unsigned integers with a static size. For example, `Signed 8` represents an 8-bit signed integer. They will wrap around when an overflow occurs.

`Vec`: It denotes a vector that contains elements of any type. It is defined in CλaSH to replace the `List` type which has a dynamic length. The length of a vector is static and parameterized. For example, `Vec 4 Bit` denotes a vector of 4 bits. The `Vec` type plays an important role in CλaSH as it is used in many built-in higher-order functions which will be discussed in Sec. 2.2.5.

Haskell allows a designer to create a new type with the `data` keyword and type synonyms can be introduced using the `type` keyword. As shown in Listing 3, the `Color` type can be `Red`, `Green` or `Blue`, and the `Pixel` type is a tuple of 3 `Color` elements.

```
data Color = Red | Green | Blue
type Pixel = (Color, Color, Color)
```

Listing 3. User-defined types

2.2.3 Polymorphism

A value is polymorphic if it can have more than one type. Polymorphism is an important and powerful feature of Haskell. Most polymorphism in Haskell falls into one of two broad categories: parametric polymorphism and ad-hoc polymorphism.

Parametric polymorphism allows functions to be defined without specifying the data types and these functions can be used for arbitrary types. The annotation shown in Listing 4 means that the function `first` takes a tuple of an `a`-type element and a `b`- type element as input and the output is of type `a`, where `a` and `b` are not concrete types but parameterized ones that can be any type. As we know, VHDL is a strongly typed language, meaning that the type of every variable has to be explicitly declared. Haskell is also strongly typed but the compiler can infer the variables' types from the functions' types. For example, if the `first` function is applied with an input `(arg1, arg2)`, `arg1` and `arg2` will automatically have the `a` and `b` types. This somewhat reduces the verbosity of the source code. With parametric polymorphism, a list operation can be used for lists that have different lengths and different element types. It is the fundamental of the built-in higher-order functions which will be introduced in Sec. 2.2.5.

```
first :: (a, b) -> a
```

Listing 4. Parametric polymorphism

Another type of polymorphism is ad-hoc polymorphism. It refers to functions that work with types in the same type class. Listing 5 indicates that the type of the `add` function is `a->a->a` and `a` must be a member of `Num` which is the class of numeric types including all real numbers.

```
add :: Num a => a -> a -> a
add a b = a + b
```

Listing 5. Ad-hoc polymorphism

CλaSH supports both parametric polymorphism and ad-hoc polymorphism with one constraint: the arguments of the top-level cannot be polymorphic as there is no way to infer their concrete types.

2.2.4 Choices

In Haskell, choices can be described in several forms: `case` expressions, `if-then-else` expressions, pattern matching and guards. All the four forms can be mapped to multiplexers. Pattern matching is a user-friendly and also powerful form of choice that is not found in the traditional HDLs. As shown in Listing 6, a function called `muxPatterns` is defined in multiple clauses with different patterns. When the function is applied with the input values that match one of the patterns, the corresponding clause will be used: if the first argument of `muxPatterns` is `Low`, the output will be the first element of the tuple; otherwise, the output will be the second element of the tuple. Figure 5 illustrates the corresponding circuit.

```
muxPatterns Low (x, y) = x
muxPatterns High (x, y) = y
```

Listing 6. Pattern matching

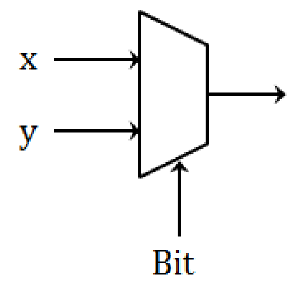

Figure 5. Multiplexer circuit

2.2.5 Higher-order Functions

Higher-order function is a powerful abstraction mechanism in a functional programming language. A higher-order function is a function that takes one or more functions as arguments. A function to be passed to the higher-order function as an argument is called a *first-class* function. Haskell provides a number of built-in higher-order functions such as `map`, `zipWith` and `foldl`.

`map` is a higher-order function that can be found in many functional languages. Listing 7 means that the first-class function `f` is applied to each element of the `xs` list and `ws` is a list of the results, as shown in Figure 6.

```
ws = map f xs
```

Listing 7. map

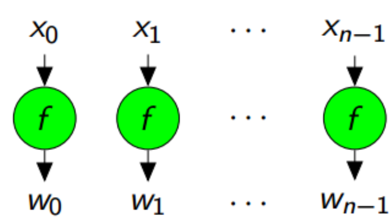

Figure 6. map

In Haskell, the first-class function can be written in another two ways: partial application and lambda expression. Partial application means applying a function with fewer arguments than it needs, which produces a new function. As shown in Listing 8, `(add 1)` is a partial application of the `add` function with the value 1 and it is again a function that takes one input and adds 1 to it. The new function `(add 1)` is applied to every element in the list `xs`, as shown in Figure 7.

```
map (add 1) xs
```

Listing 8. Partial application

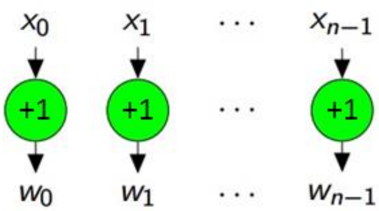

Figure 7. map (add 1)

A lambda expression allows the designer to introduce a function in any expression without first defining that function. Such a function is also called an anonymous function since it does not have a name. The expression `(λx -> x + 1)` in Listing 9 is an example of lambda expression which describes the same function as `(add 1)`.

```
map (λx -> x + 1) xs
```

Listing 9. Lambda expression

`zipWith` is a higher-order function that applies a function pairwise to the elements of two lists. For example, Listing 10 means that the elements of `xs` and `ys` are pairwise multiplied and `ws` is a list of the results, as shown in Figure 8.

```
ws = zipWith (*) xs ys
```

Listing 10. zipWith

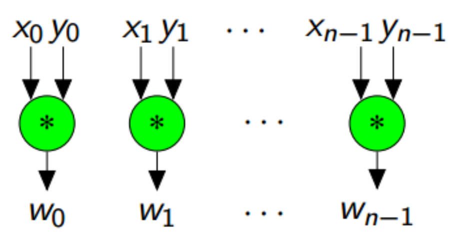

Figure 8. zipWith

Another very useful higher-order function is `foldl`. Listing 11 means that a binary operator `(+)` is iteratively applied to an element of the `ws` list and a value initialized with 0 till the end of the list, as shown in Figure 9.

```
z = foldl (+) 0 ws
```

Listing 11. foldl

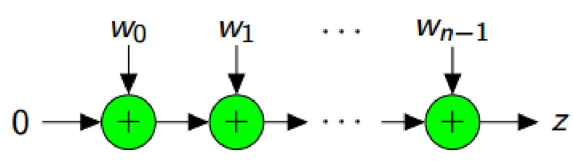

Figure 9. foldl

These higher-order functions are polymorphic as they accept lists with different lengths and different types as long as the first-class function can handle these types. Since lists cannot be translated to hardware, `map`, `zipWith` and `foldl` are replaced by `vmap`, `vzipWith` and `vfoldl` respectively in CλaSH. These functions work with vectors instead of lists.

2.2.6 Recursive Functions

Recursion plays an important role in Haskell. As shown in Listing 12, a typical example of recursion is the factorial function which cannot be translated to hardware by the CλaSH compiler. A translatable function must have a clear correspondence to a static amount of hardware resources at compile time. However, the amount of multipliers `fac` requires depends on the input value, namely `n`, which cannot be known at compile time.

```
fac : :  Int -> Int
fac 0 = 1
fac (n+1) = (n+1) * fac n
```

Listing 12. Factorial in Haskell

On the other hand, many frequently used functions in CλaSH are defined recursively, such as `vmap`, `vzipWith` and `vfoldl`. Listing 13 shows the definition of the `vmap` function, where the `:>` operator is used to add an element to the head of a vector and `Nil` denotes an empty vector. This function is supported by the CλaSH compiler because the amount of hardware resources is determined by the length of the vector `xs`, namely `n`. As we discussed in Sec. 2.2.2, `n` is a static value which is known by the compiler.

```
vmap :: (a -> b) -> Vec n a -> Vec n b
vmap _ Nil = Nil
vmap f (x :> xs) = f x :> vmap f xs
```

Listing 13. Definition of vmap

2.3 The CλaSH Compiler

The CλaSH compiler is basically a front-end of the Glasgow Haskell Compiler (GHC) extended with a Haskell library that can compile circuit descriptions written in Haskell to VHDL. Figure 10 illustrates the compiling mechanism according to [3].

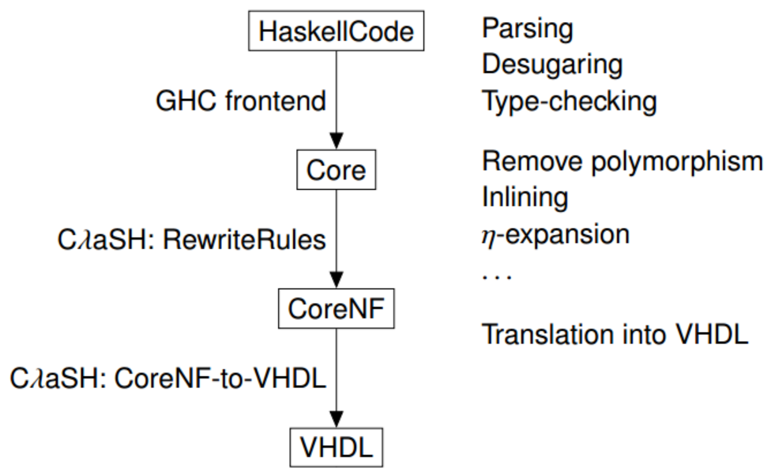

Figure 10. CλaSH pipeline

The GHC front-end performs parsing, type checking and desugaring to the original Haskell code. Haskell is a rather large language, containing many different syntactic constructs. Haskell provides a lot of "syntactic sugar" to be easy for humans to read and write, and the programmer can choose the most appropriate one from a wide range of syntactic constructs. However, the flexibility for the user leads to the complexity for the compiler because there are often several ways to describe the same meaning. For example, an `if-else-then` expression is identical in meaning to a `case` expression with `True` and `False` branches. Therefore the GHC front-end removes all the syntactic sugar and translates the original Haskell code into a much smaller typed language called *Core*.

A description in core can still contain elements which have no direct translation to hardware, such as polymorphic types and function-valued arguments. The second stage of the compiler repeatedly applies a set of rewrite rules on the *Core* description till it is in a *normal form*, which corresponds directly to hardware. This set of transformations includes β-reduction, η-expansion, unfolding higher-order functions to first order function, specifying the polymorphic types with concrete types and function inlining. The final step in the compiler pipeline is to translate the normal form to a VHDL description, which is a straightforward process due to the resemblance of a normalized description and a set of concurrent signal assignments.

Figure 11 shows the circuit of an arithmetic logic unit (ALU) and it is modeled as a function called `alu`, as defined in Listing 14. The `alu` function performs addition (`ADD`), multiplication (`MUL`) or subtraction (`SUB`) according to the `opCode`. Listing 15 presents the normalized description of the `alu` function. It becomes a lambda function with a `let-in` expression. The normalized description has a clear correspondence to the circuit in Figure 11: 1) Every variable indicates a signal (wire). 2) `λ` and `in` denote the input and output signals respectively. 3) The internal logics are described in the `let` clause where every syntactic construct has a direct translation in hardware, for instance, an adder or a multiplexer.

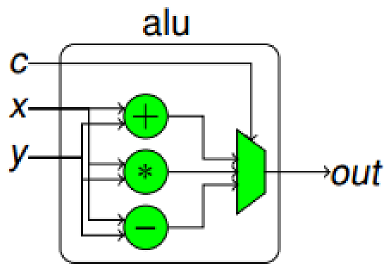

Figure 11. ALU circuit

```
data opCode = ADD | MUL | SUB

alu ADD x y = x + y
alu MUL x y = x * y
alu SUB x y = x - y
```

Listing 14. Haskell definition of alu

```
alu = λc x y. let p = x + y
                  q = x * y
                  r = x - y
                  out = case c of
                         ADD -> p
                         MUL -> q
                         SUB -> r
              in out
```

Listing 15. alu in normal form

3. MUSIC Algorithm

As shown in Figure 12, a far-field narrowband signal with a wavelength of λ arrives at an N-element antenna array. Each element of the array is spaced by d which is equal to $\lambda/2$. The angle of incidence is θ. If the received signal at sensor 1 is $x_1(t) = s(t)$, then it is received earlier at sensor i by $\Delta_i = \frac{(i-1)d\sin\theta}{c}$, where c is the propagation speed, so the received signal at sensor i is $x_i(t) = e^{-j\omega\Delta_i}s(t) = e^{-j\omega\frac{(i-1)d\sin\theta}{c}}s(t)$. The signals received at all N sensors can form a vector as:

$$X(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ \vdots \\ x_N(t) \end{bmatrix} = \begin{bmatrix} 1 \\ e^{-j\omega\frac{d\sin\theta}{c}} \\ e^{-j\omega\frac{2d\sin\theta}{c}} \\ \vdots \\ e^{-j\omega\frac{(N-1)d\sin\theta}{c}} \end{bmatrix} s(t) = a(\theta)s(t) \tag{1}$$

where $\boldsymbol{a}(\boldsymbol{\theta})$ is called a "steering vector".

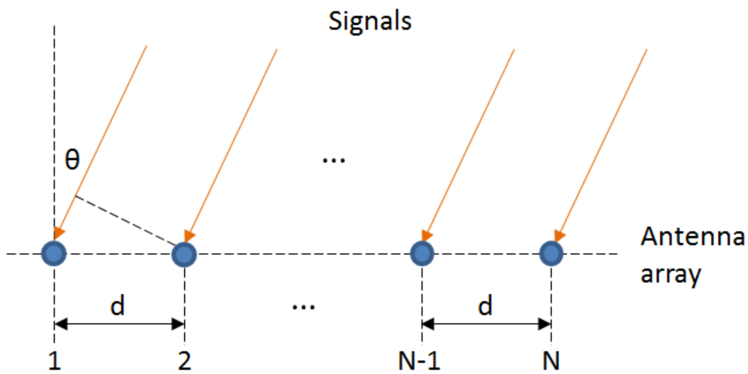

Figure 12. Uniform linear array

If there are M independent source signals and Gaussian white noise is $n(t)$, the signal model can be depicted as:

$$X(t) = AS(t) + N(t) \tag{2}$$

where $X(t) = [x_1(t), x_2(t), \dots, x_N(t)]^T$

$$S(t) = [s_1(t), s_2(t), \dots, s_M(t)]^T$$

$$N(t) = [n_1(t), n_2(t), \dots, n_N(t)]^T$$

$$A = [a(\theta_1), a(\theta_2), \dots, a(\theta_M)]$$

Then the covariance matrix can be calculated as:

$$R_x = E\{X(t)X^H(t)\} = AR_sA^H + \sigma^2 \tag{3}$$

where $R_s = E\{S(t)S^H(t)\}$, σ^2 is the noise variance and I is the $N \times N$ identity matrix. The rank of R_s defines the dimension of the signal subspace.

For $N > M$, the matrix AR_sA^H is singular, so $det[AR_sA^H] = det[R_x - \sigma^2 I]$ = 0, which implies that σ^2 is an eigenvalue of R_x. Since the dimension of the null space of AR_sA^H is $N - M$, R_x has $N - M$ eigenvalues that are equal to σ^2. Since R_x is a positive definite Hermitian matrix, there are M other eigenvalues λ_i and $\lambda_i > \sigma^2 > 0$.

If u_i is the eigenvector of R_x corresponding to λ_i, then $R_x u_i = [AR_sA^H + \sigma^2 I]u_i = \lambda_i u_i$ $(i = 1,2, \dots, N)$, which implies that

$$AR_sA^H u_i = \begin{cases} (\lambda_i - \sigma^2)u_i; & i = 1,2, \dots, M \\ 0; & i = K + 1, \dots, N \end{cases} \tag{4}$$

The *N*-dimensional eigenvector space can be partitioned into the signal subspace U_s and the noise subspace U_n , as shown in Eq. (5) where the eigenvectors are in descending order.

$$[U_s \quad U_n] = [u_1 \dots u_M \quad u_{M+1} \dots u_N] \tag{5}$$

Since both $A^H A$ and R_s are full-rank matrices, meaning that $(A^H A)$ and ${R_s}^{-1}$ exist, Eq. (4) can be transformed to ${R_s}^{-1}(A^H A)^{-1} A^H \mathrm{A} R_s A^H \mathrm{u_i} = 0$, so

$$A^H u_i = 0 \quad (i = M + 1, M + 2, \dots, N) \tag{6}$$

which means the noise subspace is orthogonal to each column of the steering matrix A.

According to this orthogonality, a spatial spectrum function can be constructed as

$$P(\theta) = \frac{1}{\left\| a^H(\theta) \; U_n \right\|_2^{\,2}} \tag{7}$$

Since the above deduction is based on some assumptions to build an idealized mathematical model, the denominator of the function can never be exactly 0 in reality. The values of θ that maximize $P(\theta)$ are corresponding to the DOAs of all source signals. In other words, the DOA's of all source signals can be estimated by peak detection of the spatial spectrum.

4. Hardware Implementation

Figure 13 illustrates the system architecture of a MUSIC hardware implementation. First, a pretreatment will be performed on the signal data after A/D conversion. The purpose of the pretreatment is to get rid of complex computations and make it easier to be implemented on a FPGA. The FPGA implementation consists of three modules: Covariance Matrix Calculation (CMC), Eigen-decomposition (EVD) and Spectrum Peak Search (SPS).

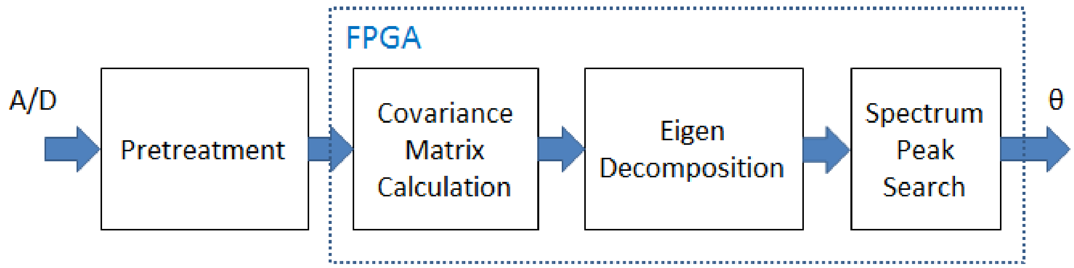

Figure 13. System hardware architecture

4.1 Pretreatment

As the steering matrix A contains complex elements, the MUSIC algorithm requires a large amount of complex-valued computations which make the hardware implementation complex and time-consuming especially for the EVD. To reduce the computational load, **[4]** introduces a pretreatment method to obtain a real-valued covariance matrix by a unitary transformation as

$$Y(n) = T^H X(n) \qquad (8)$$

where $T = \frac{1}{\sqrt{2}}\begin{bmatrix} I & jI \\ Q & -jQ \end{bmatrix}$ if there is an even number of antennas in the array, I is a $\frac{N}{2} \times \frac{N}{2}$ identity matrix and Q is a $\frac{N}{2} \times \frac{N}{2}$ anti-identity matrix (permutation matrix with all its anti-diagonal elements being 1). In this method, N is assumed to be an even number. After the pretreatment, we can obtain a real-valued steering vector as

$$a(\theta_k) = [\cos\left(\frac{\pi d \sin\theta_k}{\lambda}\right), \cos\left(\frac{3\pi d \sin\theta_k}{\lambda}\right), \dots, \cos\left(\frac{(2N-1)\pi d \sin\theta_k}{\lambda}\right),$$
$$\sin\left(\frac{\pi d \sin\theta_k}{\lambda}\right), \sin\left(\frac{3\pi d \sin\theta_k}{\lambda}\right), \dots, \sin\left(\frac{(2N-1)\pi d \sin\theta_k}{\lambda}\right)]^T \qquad (9)$$

where $k = 1, 2, \dots, M$.

4.2 Covariance Matrix Calculation

According to Eq. (3), the covariance matrix calculation is basically the multiplication of a vector and its transpose. After the pretreatment, the data vector $X(n)$ is real-valued. Each element of the covariance matrix can be calculated as

$$R_{\mathrm{ij}} = \frac{1}{M}\sum_{n=1}^{M} Y_i(n)\, Y_j(n) \tag{10}$$

where R_{ij} is the element in row i, column j of the covariance matrix, $Y_i(n)$ and $Y_j(n)$ denote the *n*-th data of the *i*-th and the *j*-th antennas respectively, and M is the number of snapshots. The calculations of the entire covariance matrix can be done in parallel by $N \times N$ multiply-accumulate (MAC) units. As shown in Figure 14, a MAC unit multiplies the two input values and adds the multiplication result with the previous output which is stored in a register. Since R is a symmetric matrix, the upper triangle is sufficient for the implementation of the MUSIC algorithm. Therefore, $\frac{N\times(N+1)}{2}$ MAC units are required.

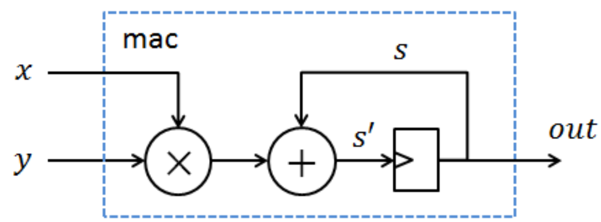

Figure 14. mac circuit

Figure 15 is the block diagram of the Covariance Matrix Calculation (CMC) module. Each input signal is combined with itself and the others. For example, with two elements `a` and `b`, the combinations will be `(a,a)`, `(a,b)` and `(b,b)`. Therefore, N input signals make $\frac{N\times(N+1)}{2}$ combinations and each combination is the input of a MAC unit.

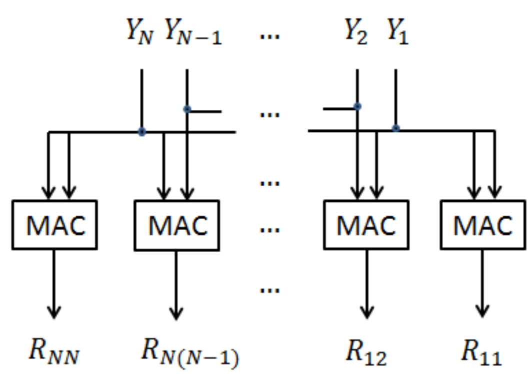

Figure 15. Covariance matrix calculation

4.3 Eigenvalue Decomposition

The Jacobi eigenvalue algorithm is an iterative method to calculate the eigenvalues and eigenvectors of a real symmetric matrix such as the covariance matrix. The Jacobi method repeatedly performs rotations (orthogonal transformations) until the matrix becomes almost diagonal.

4.3.1 The CORDIC Algorithm

Before discussing more about the Jacobi method, it is necessary to introduce the CORDIC (Coordinate Rotation Digital Computer) algorithm since it plays an important role in the implementation of the Jacobi method. CORDIC is a simple and efficient algorithm to calculate trigonometric functions. It is commonly used when no hardware multiplier is available (e.g., simple microcontrollers and FPGAs) as the only operations it requires are addition, subtraction, bit shift and table lookup.

Suppose a vector (x, y) is rotated by an angle α, the resulting vector (x', y') can be calculated as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{11}$$

Eq. (11) can be rewritten as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \cos a \begin{bmatrix} 1 & -\tan\alpha \\ \tan\alpha & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{12}$$

If $\alpha = a_0 \pm a_1 \pm \cdots a_n$, this rotation can be decomposed to iterative rotations by the angle $ai\ (i = 0, 1, \ldots, n)$. Each iteration can be depicted as:

$$\begin{bmatrix} x^{(i+1)} \\ y^{(i+1)} \end{bmatrix} = \cos a_i \begin{bmatrix} 1 & -\tan\alpha_i \\ \tan\alpha_i & 1 \end{bmatrix} \begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix} \tag{13}$$

with $x^{(0)} = x, y^{(0)} = y$.

Suppose a_i is chosen such that $\tan a_i = 2^{-i}$, then

$$\alpha_i = \arctan 2^{-i} \tag{14}$$

$$\alpha = \sum_{i=0}^{n} d_i \alpha_i \ \ (d_i = \pm 1) \tag{15}$$

Table 1 lists the possible values of a_i which can be stored in a look–up table (LUT). The accuracy of the final result of CORDIC is determined by the number of iterations, i.e. the number of angle values in the table. Eq. (13) can be rewritten as:

$$\begin{bmatrix} x^{(i+1)} \\ y^{(i+1)} \end{bmatrix} = \cos(\arctan 2^{-i}) \begin{bmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x^{(i)} \\ y^{(i)} \end{bmatrix} \tag{16}$$

Now the calculations do not require multiplications but only bit shifts, except for the first factor in Eq. (16): $\cos(\arctan 2^{-i}) = \frac{1}{\sqrt{1+2^{-2i}}}$.

i	0	1	2	3	4	5	6	7	8	9
2^{-i}	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512
$\arctan 2^{-i}$	45.0	26.6	14.0	7.1	3.6	1.8	0.9	0.4	0.2	0.1

Table 1. Angles for CORDIC rotation

The progress of a CORDIC rotation is tracked by an angle accumulator:

$$z^{(i+1)} = z^{(i)} - d_i\alpha_i \tag{17}$$

The product of $cos(arctan2^{-i})$ can be depicted as $\frac{1}{K}$ where $K = \prod_{i=1}^{n}\sqrt{1+2^{-2i}}$ and K converges to 1.647 [5]. Therefore, we can ignore $cos(arctan2^{-i})$ in each iteration and finally the original vector will be scaled by a factor of K. Eq. (18) is a summary of the equations in the CORDIC algorithm.

$$\begin{aligned} x^{(i+1)} &= x^{(i)} - d_i 2^{-i} y^{(i)} \\ y^{(i+1)} &= d_i 2^{-i} x^{(i)} + y^{(i)} \\ z^{(i+1)} &= z^{(i)} - d_i\alpha_i \end{aligned} \tag{18}$$

There are two computing modes of CORDIC: *rotation mode* and *vectoring mode*. In a rotation-mode CORDIC, the sign of d_i is determined by the angle accumulator: $d_i = 1$ when $z^{(i)} \geq 0$ and $d_i = -1$ otherwise. With the following initial values:

$$\begin{cases} x^{(0)} = x \\ y^{(0)} = y \\ z^{(0)} = \alpha \end{cases} \tag{19}$$

the final result will be:

$$\begin{aligned} x^{(n)} &= K(xcos\alpha - ysin\alpha) \\ y^{(n)} &= K(xsin\alpha + ycos\alpha) \\ z^{(n)} &= 0 \end{aligned} \tag{20}$$

In a vectoring-mode CORDIC, the sign of d_i depends on $y^{(i)}$: $d_i = -1$ when $\mathrm{y}^{(\mathrm{i})} > 0$ and $d_i = 1$ when $y^{(i)} \leq 0$. With the following initial values:

$$\begin{cases} x^{(0)} = x \\ y^{(0)} = y \\ z^{(0)} = 0 \end{cases} \tag{21}$$

the final result will be:

$$\begin{aligned} x^{(n)} &= K\sqrt{x^2 + y^2} \\ y^{(n)} &= 0 \\ z^{(n)} &= arctan\left(\frac{x}{y}\right) \end{aligned} \tag{22}$$

Figure 16 presents a CORDIC architecture which can be used for both the two modes. The left part of this architecture performs bit shifts according to Eq.(16) and the right part is an angle accumulator corresponding to Eq. (17).

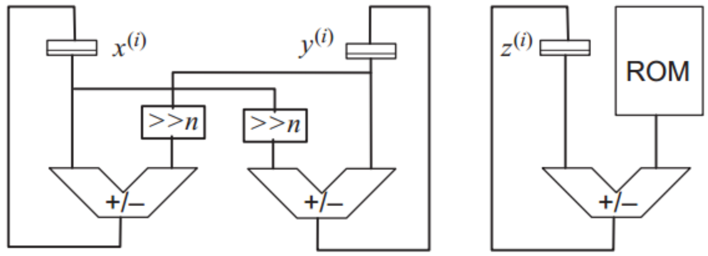

Figure 16. Iterative CORDIC architecture

4.3.2 The Classical Jacobi Method

$S^{(1)} = G^T S G$ is symmetric and similar to S, if S is an $M \times M$ real symmetric matrix and $G(i, j, \theta)$ is a rotation matrix of the form:

$$\begin{array}{cc} & \begin{array}{ccccccc} & & i & & j & & \end{array} \\ \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} & \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \end{array} \tag{23}$$

where $s = \sin\theta$ and $c = \cos\theta$. All the diagonal elements of G are unity except for the two elements in rows (and columns) i and j. All the off-diagonal elements of G are zeros except the two elements in row i, column j and row j, column i.

The elements of $S^{(1)}$ are calculated as

$$\begin{cases} S_{ii}^{(1)} = c^2 S_{ii} - 2scS_{ij} + s^2 S_{jj} \\ S_{jj}^{(1)} = s^2 S_{ii} + 2scS_{ij} + c^2 S_{jj} \\ S_{ij}^{(1)} = S_{ji}^{(1)} = (c^2 - s^2) S_{ij} + sc(S_{ii} - S_{jj}) \\ S_{ik}^{(1)} = S_{ki}^{(1)} = cS_{ik} - sS_{jk} \qquad k \neq i, j \\ S_{jk}^{(1)} = S_{kj}^{(1)} = sS_{ik} + cS_{jk} \qquad k \neq i, j \\ S_{kl}^{(1)} = S_{kl} \qquad\qquad\qquad\quad k, l \neq i, j \end{cases} \tag{24}$$

Since S is a symmetric matrix, we can concentrate on the upper triangle. One of the off-diagonal elements will be annihilated if $S_{ij}^{(1)}$ is set to 0, which means

$$\tan(2\theta) = \frac{2S_{ij}}{S_{jj} - S_{ii}} \tag{25}$$

If $S_{jj} = S_{ii}$, $\theta = \frac{\pi}{4}$.

The Jacobi method performs a sequence of orthogonal similarity transformations as shown in Eq. (26). Each transformation (a *Jacobi rotation*) is a plane rotation that annihilates one of the off-diagonal elements. Successive transformations undo the previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is almost diagonal.

The iterations of the Jacobi method can be depicted as

$$\begin{cases} S^{(1)} = G_1^T S G_1 \\ S^{(2)} = G_2^T S^{(1)} G_2 \\ \qquad \vdots \\ S^{(L)} = G_L^T S^{(L-1)} G_L \end{cases} \tag{26}$$

where L denotes the number of iterations, so

$$S^{(L)} = {G'}^T S G' \tag{27}$$

where $G'^T = G_1^T G_2^T \cdots G_L^T$ and $G' = G_1 G_2 \cdots G_L$.

After L iterations, $S^{(L)}$ is almost diagonal. The diagonal elements of $S^{(L)}$ are approximations of the eigenvalues and the corresponding eigenvectors are the columns of G'.

The original Jacobi method searches the whole upper triangle in each iteration and sets the largest off-diagonal element to zero. "This is a reasonable strategy for hand calculation, but it is prohibitive on a computer since the search alone makes each Jacobi rotation a process of order N^2 instead of N."**[6]** For a hardware implementation, $S_{ij}^{(n)}$which is the off-diagonal element to be annihilated in the *n*-th iteration, is determined by traversing the upper triangle in a fixed order, for example, in a 4×4 symmetric matrix:

$$S_{12} \rightarrow S_{13} \rightarrow S_{14} \rightarrow S_{23} \rightarrow S_{24} \rightarrow S_{34}$$

One such set of $L(L-1)/2$ Jacobi rotations is called a *sweep*. The diagonalization of the matrix will be finished after a few sweeps when all off-diagonal elements are smaller than a predefined threshold.

Eq. (24) can be rewritten as

$$
\begin{cases}
S_{ii}^{(1)} = c(cS_{ii} - sS_{ij}) - s(cS_{ij} - sS_{jj}) \\
S_{jj}^{(1)} = s(sS_{ii} + cS_{ij}) + c(sS_{ij} + cS_{jj}) \\
S_{ij}^{(1)} = S_{ji}^{(1)} = 0 \\
S_{ik}^{(1)} = S_{ki}^{(1)} = cS_{ik} - sS_{jk} & k \neq i,j \\
S_{jk}^{(1)} = S_{kj}^{(1)} = sS_{ik} + cS_{jk} & k \neq i,j \\
S_{kl}^{(1)} = S_{kl} & k,l \neq i,j
\end{cases}
\tag{28}
$$

By comparing Eq. (28) with the results of the rotation-mode CORDIC (Coordinate Rotation Digital Computer) in Eq. (20), it can be concluded that the calculations of the off-diagonal elements $S_{ik}^{(1)}$ and $S_{jk}^{(1)}$ can be done by a CORDIC rotation. The diagonal elements $S_{ii}^{(1)}$ and $S_{jj}^{(1)}$ can be calculated by performing the CORDIC rotation twice. And the rotation angle θ can be computed by the vectoring-mode CORDIC according to Eq. (22) and Eq. (25).

According to Eq. (27), the calculations of G' are iterative multiplications of the Jacobi rotation matrices. Eq. (29) shows an example of the first iteration.

$$
\begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} c_2 & 0 & -s_2 & 0 \\ 0 & 1 & 0 & 0 \\ s_2 & 0 & c_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c_1c_2 & -s_1 & -c_1s_2 & 0 \\ s_1c_2 & c_1 & -s_1s_2 & 0 \\ s_2 & 0 & c_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\tag{29}
$$

where c_1, s_1 represent the cosine and sine values in the first iteration and c_2, s_2 represent the cosine and sine values in the second iteration. It can be concluded that as long as the second matrix is of the form shown in Eq. (23), only column i and column j of the first matrix are changed during the multiplication of these two matrices. The result of each multiplication can be depicted as Eq. (30), where V_{ki} and V_{kj} represent the old values of column i and j, while V_{ki}' and V_{kj}' are the new values.

$$
\begin{bmatrix} V_{ki}' \\ V_{kj}' \end{bmatrix} = \begin{bmatrix} c_2 & -s_2 \\ s_2 & c_2 \end{bmatrix} \times \begin{bmatrix} V_{ki} \\ V_{kj} \end{bmatrix}
\tag{30}
$$

Eq. (30) is actually equivalent to a CORDIC rotation as shown in Eq.(20), which means that the calculation of $\boldsymbol{G}'$ can be done by a rotation-mode CORDIC.

4.3.3 The Improved Jacobi Method

As discussed in the previous section, the classic Jacobi method uses CORDIC 3 times (2 rotation-mode CORDIC and 1 vector-mode CORDIC) in each Jacobi rotation. An improved design that uses CORDIC only once will be presented in this section. It can significantly improve the efficiency of the Jacobi method.

The angle θ_i in each iteration of a CORDIC rotation is determined by the equation: $\theta_i = \theta - \sum_{j=0}^{i-1} d_i \alpha_i$, where $\alpha_i = tan^{-1}(2^{-i+1}), i = 0,1,2,\cdots,k$ and $d_i \in \{-1,1\}$. The rotation direction d_i is determined by the sign of θ_{i-1}. For the Jacobi method, the rotation angle θ can be restricted within $\pi/4$ **[7]**, so d_i is also determined by the sign of $tan\,2\theta_{i-1}$. By applying the trigonometric identities, $tan\,2\theta_i$ can be calculated as

$$
\begin{aligned}
tan\,2\theta_i &= tan(2\theta_{i-1} - 2d_i\alpha_i) \\
&= \frac{tan(2\theta_{i-1}) - d_i\,tan\,2\alpha_i}{1 + d_i\,tan\,2\theta_{i-1}\,tan\,2\alpha_i}
\end{aligned}
\tag{31}
$$

With $tan\,2\alpha_i = \frac{2^{1-i}}{1-2^{-2i}}$, $tan\,2\theta_i = \frac{\gamma_i}{\mu_i}$ and $tan\,2\theta_{i-1} = \frac{\gamma_{i-1}}{\mu_{i-1}}$, Eq. (31) can be rewritten as

$$
\frac{\gamma_i}{\mu_i} = \frac{(1-2^{-2i})\gamma_{i-1} - d_i 2^{1-i}\mu_{i-1}}{(1-2^{-2i})\mu_{i-1} + d_i 2^{1-i}\gamma_{i-1}}
\tag{32}
$$

Figure 17 illustrates the block diagram of a modified CORDIC algorithm used for calculating the off-diagonal elements. The `CORDIC_A` section computes the values of γ_i and μ_i according to Eq. (31) where the sign of d_i is determined by the sign of $\frac{\gamma_i}{\mu_i}$. The `CORDIC_B` section is a rotation-mode CORDIC that rotates in the direction indicated by the sign of d_i.

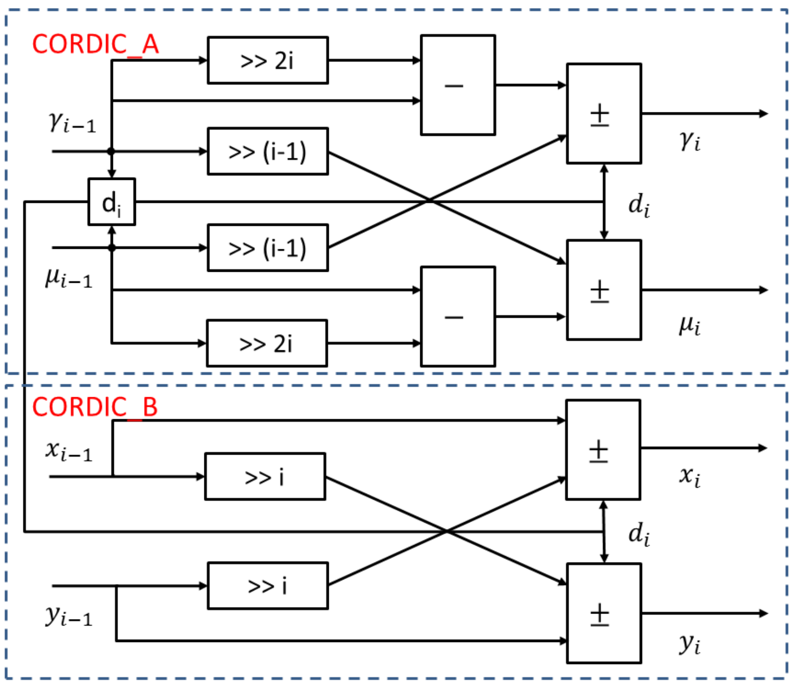

Figure 17. Modified CORDIC

With the following initial values:

$$
\begin{cases}
\gamma_0 = 2S_{ij} \\
\mu_0 = S_{ii} - S_{jj} \\
x_0 = S_{ik} \\
y_0 = S_{jk}
\end{cases}
\tag{33}
$$

the results of the modified CORDIC after n iterations will be:

$$\begin{cases} x_n = K\left(S_{ik}cos\theta - S_{jk}sin\theta\right) \\ y_n = K\left(S_{ik}sin\theta + S_{jk}cos\theta\right) \end{cases} \tag{34}$$

According to Eq. (24), the new off-diagonal elements $S_{ik}^{(1)}$ and $S_{jk}^{(1)}$ can be calculated by scaling x_n and y_n with a factor of K. As shown in Figure 18, the scaling is implemented according to the approximation:

$$\frac{1}{K} = 0.6073 \approx 2^{-1} + 2^{-3} - 2^{-6} - 2^{-9} - 2^{-13} \tag{35}$$

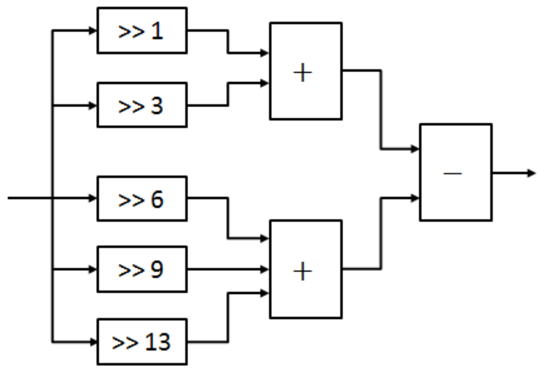

Figure 18. CORDIC scaling

For the diagonal elements, according to [6], it can be derived from Eq. (25) and (28) that

$$\begin{cases} S_{ii}^{(n)} = S_{ii}^{(n-1)} - S_{ij}^{(n-1)}\tan\theta_n \\ S_{jj}^{(n)} = S_{jj}^{(n-1)} + S_{ij}^{(n-1)}\tan\theta_n \end{cases} \tag{36}$$

The value of $tan\,\theta_n$ can be stored in a look-up table in which a set of d_i is mapped to $tan\,\theta_n$, as shown in Figure 19. In a hardware implementation, $d_i = -1$ is considered as 0.

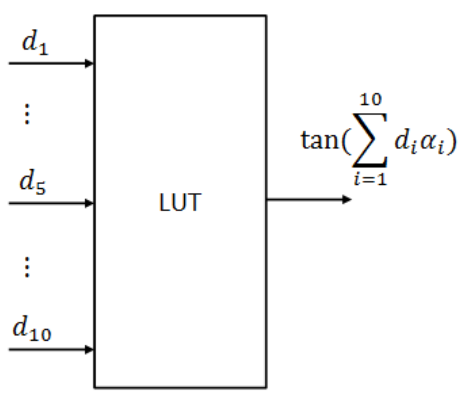

Figure 19 Look-up table of tangent

4.3.4 Systolic Array

According to Eq. (24), each Jacobi rotation affects only row (and column) i and j, which offers an opportunity of parallel processing. A systolic array design is proposed in **[7]** to implement the parallel Jacobi algorithm. Figure 20 shows a systolic array used for the EVD of a 4×4 symmetric matrix. Each PE (processing element) contains a 2×2 sub-matrix of the upper triangle of the matrix. For example, "12" in PE1 represents the element in row 1, column 2. The PEs on the diagonal line, namely PE1 and PE3, are called the *diagonal processors* and PE2 is called the *off-diagonal processor*. Using the CORDIC_A algorithm shown in Figure 17, the diagonal processors update the four diagonal elements in parallel ("12" and "34" are set to 0) and broadcast the values of d_i to the right and the top, as indicated by the wide arrows in Figure 20. Each off-diagonal processor has to wait for the arrivals of d_i from the left and the bottom to update the off-diagonal elements using CORDIC_B. After all the elements are updated, they will be relocated along the thin arrows and then the PEs will start the next iteration. Compared with the classical Jacobi method, the systolic array can significantly reduce the total computation time of EVD, especially for a big matrix.

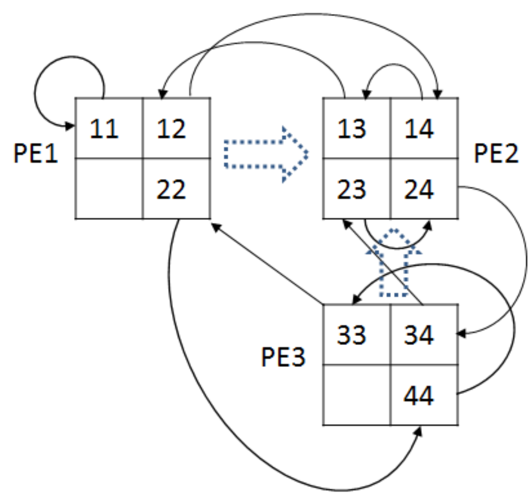

Figure 20. Systolic array for EVD

Since the systolic array annihilates 2 off0diagonal elements in each iteration, one sweep of a 4×4 symmetric matrix can be done by 3 iterations. In Figure 21, each off-diagonal element in the upper triangle of a 4×4 symmetric matrix is marked with a number that indicates in which iteration it will be annihilated. There are no conflicts between the calculations of the diagonal elements in each iteration. For example, according to Eq. (24), the diagonal elements (1, 1) and (2, 2) are required and will be changed to annihilate (1, 2) which is marked with '1'. To annihilate (3, 4) which is also marked with '1', the diagonal elements (3, 3) and (4, 4) are required and will be changed. So the two rotations do not affect the diagonal elements of each other. According to Eq. (24), both the two rotations affect the 4 elements in PE2, which means PE2 has to perform the CORDIC rotation twice in each iteration.

	1	2	3
		3	2
			1

Figure 21. Parallel processing of a 4x4 matrix

Figure 22 shows the hardware architecture of EVD of a 4×4 symmetric matrix. MUX is a multiplexer that chooses from the input and the previous output stored in the memory unit REG1. A *diagonal processor* consists of a CORDIC_A and an update block. The output of the CORDIC_A block is a set of the direction signals, namely $ds1$ or $ds2$ which are then used by the update block to get the tangent value from an internal look-up table and update the diagonal elements. An *off-diagonal* processor consists of 4 CORDIC_B blocks that update the off-diagonal elements. The EX1 block performs data exchanges between two iterations and stores the results in a memory unit called REG1. After a few iterations, the upper triangle of a diagonalized matrix will be found in REG1 and its diagonal elements are the eigenvalues of the input matrix R (upper triangle).

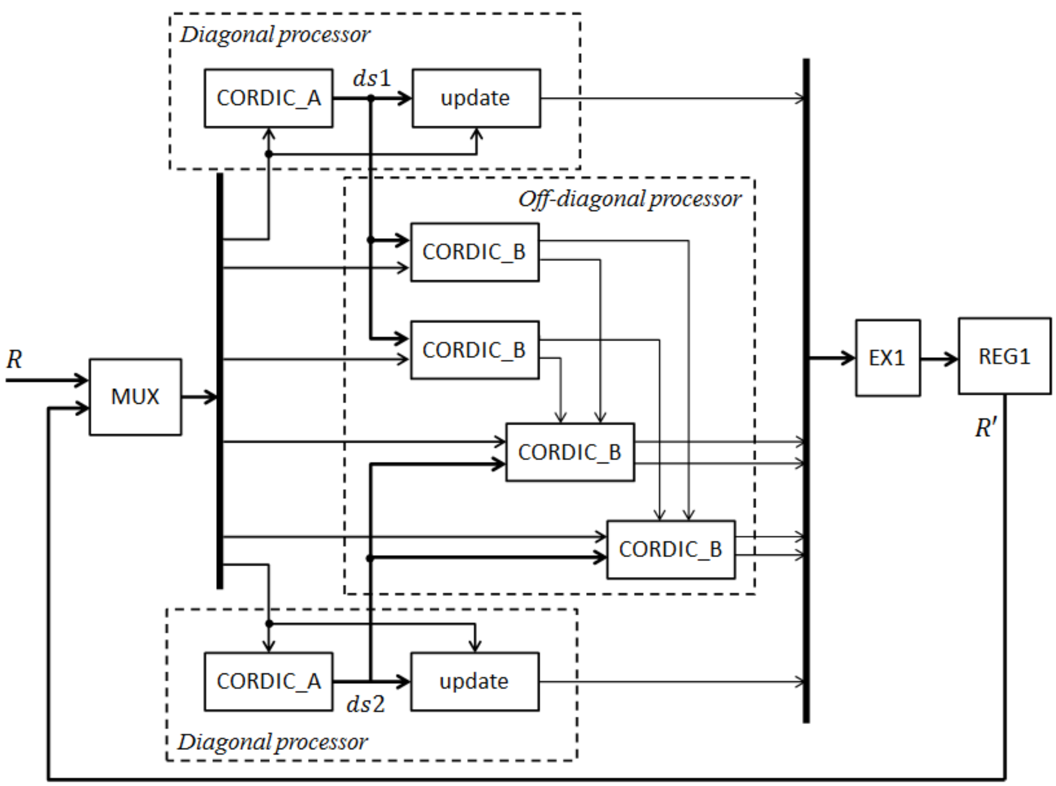

Figure 22. EVD (eigenvalue) architecture

Figure 23 is an extension to Figure 22. With this extension the eigenvectors can be calculated at the same time. There are 8 CORDIC_B blocks running in parallel: 4 take $ds1$ and the other 4 take $ds2$. The memory unit REG2 is initialized with an identity matrix. The 4 CORDIC_B blocks in the left pairwise update the elements in column 1 and column 2 and the 4 CORDIC_B blocks in the right pairwise update the elements in column 3 and column 4. Then EX2 will perform the data exchanges between the columns as shown in Figure 24, where each block represents a column. The result will be stored in REG2 for next iteration. When the eigenvalues calculation is finished, the corresponding eigenvectors can be found in REG2.

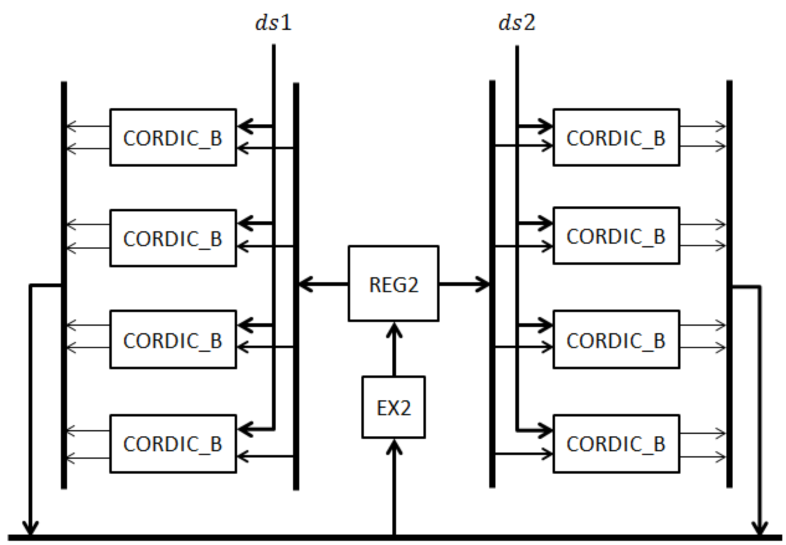

Figure 23. EVD (eigenvector) architecture

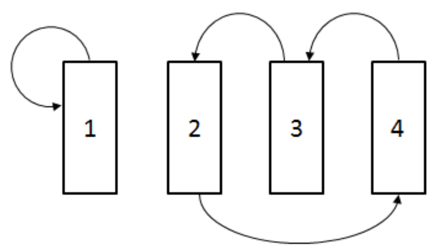

Figure 24. Column exchange

4.4 Spectral Peak Search

According to Eq. (7), the spectrum peaks can be detected by finding the minimum square of the 2-norm of $a^H(\theta)\ U_n$, which is equivalent to finding the maximum of the 2-norm of $a^H(\theta)\ U_s$ where the signal space U_s consists of the eigenvectors corresponding to the largest eigenvalues. The latter can reduce the amount of computations when the number of source signals is much smaller than the number of noises. Figure 25 presents the block diagram of the spectral peak search module. First, the EigSort block sorts the eigenvalues in a descending order and outputs the corresponding eigenvectors of the first M eigenvalues, making the signal space U_s, where M indicates the number of source signals. The Norm block takes the signal space U_s from EigSort and a steering vector $a^H(\theta)$ from the SvLUT block to calculate the 2-norm of $a^H(\theta)\ U_s$. For a hardware implementation, the angle θ can be chosen from a predefined set of angles, for example: $\frac{\pi}{512}, \frac{\pi}{256}, \dots, \frac{\pi}{2}$. According to Eq. (9), the steering vectors will be:

$$a^H(\theta_1) = [\cos\left(\frac{\pi}{2} sin\theta_1\right), \cos\left(\frac{3\pi}{2} sin\theta_1\right), \sin\left(\frac{\pi}{2} sin\theta_1\right), \sin\left(\frac{3\pi}{2} sin\theta_1\right)]$$

$$a^H(\theta_2) = [\cos\left(\frac{\pi}{2} sin\theta_2\right), \cos\left(\frac{3\pi}{2} sin\theta_2\right), \sin\left(\frac{\pi}{2} sin\theta_2\right), \sin\left(\frac{3\pi}{2} sin\theta_2\right)]$$

$$\vdots$$

$$a^H(\theta_k) = [\cos\left(\frac{\pi}{2} sin\theta_k\right), \cos\left(\frac{3\pi}{2} sin\theta_k\right), \sin\left(\frac{\pi}{2} sin\theta_k\right), \sin\left(\frac{3\pi}{2} sin\theta_k\right)] \tag{37}$$

where $\theta_k = \frac{k\pi}{512}, k = 1,2, \dots, 256$. These steering vectors are stored in `SvLUT` as constants. The result of the norm calculation will be sent to the `Compare` block and compared with the previous results to find out the peaks which indicate the DOAs.

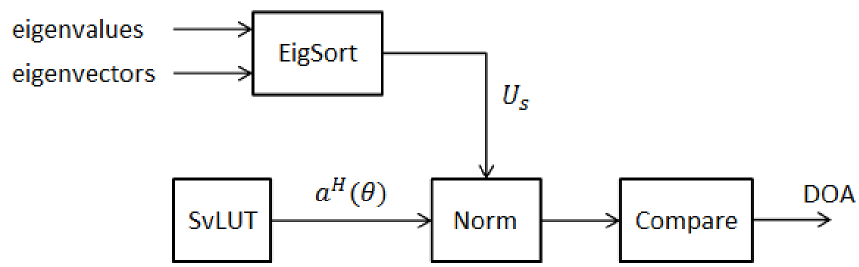

Figure 25. Spectral peak search

As shown in Figure 26, the `Norm` block first calculates the dot product of a steering vector and each eigenvector in the signal space. Then each dot product is squared and the output is the sum of the squares.

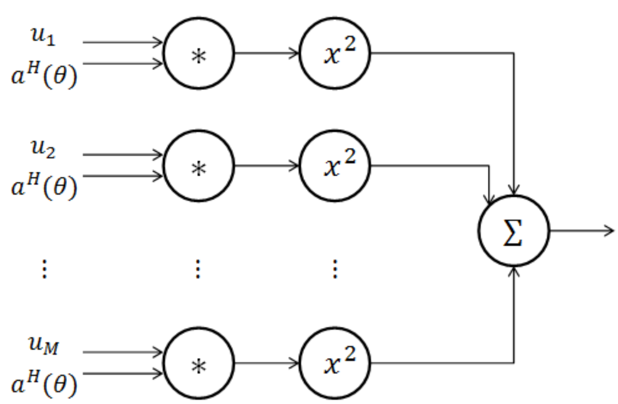

Figure 26. Norm calculation

The `Compare` block is elaborated in Figure 27. First `Comp1` compares V_{i-1} with V_i and V_{i-2} simultaneously. If $V_{i-1} < V_i$ and $V_{i-1} < V_{i-2}$ then `Comp2` will compare V_{i-1} with 2 (depends on the number of signal sources) current maximums and output the indexes of the maximums. Finally the DOA's can be found according to the indexes after traversing the entire angle set.

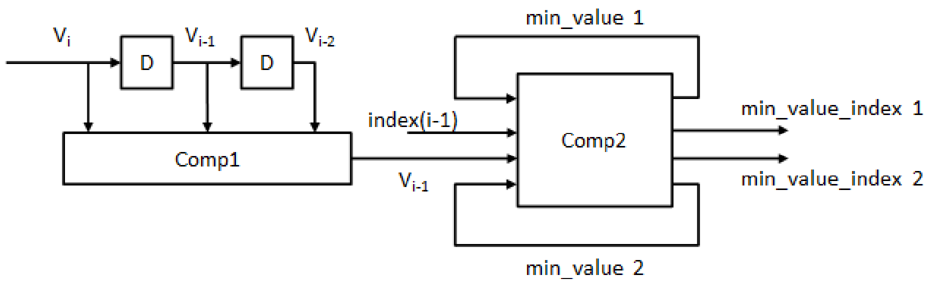

Figure 27. Architecture of the Compare block

5. CλaSH Implementation of MUSIC

This chapter describes the CλaSH implementation of the MUSIC algorithm according to the hardware designs shown in Chapter 4 and presents the simulation results. Each module of the MUSIC algorithm, such as Covariance Matrix Calculation (CMC), Eigen-decomposition (EVD) and Spectral Peak Search (SPS), is separately implemented in CλaSH. A two-step design method is proposed in [8] to implement a DSP application on an FPGA: firstly, the mathematical definition is translated to Haskell; secondly, minor changes are applied to the Haskell implementation so that it is accepted by the CλaSH compiler. For example, lists are replaced by vectors and map is replaced by vmap. The pure Haskell code is more concise and easier to use as it is free of the hardware-related restrictions in CλaSH. For example, in Haskell we can use double precision floating point operations while in CλaSH we use fixed point operations. Therefore, this chapter will use the Haskell code to describe the implementation of the MUSIC algorithm and the corresponding CλaSH code can be found in the Appendix. In this project, we assume that the number of antennas is 4 and the number of source signals is 1.

5.1 Covariance Matrix Calculation

5.1.1 CλaSH Implementation

According to the description in Sec. 4.2, the Covariance Matrix Calculation (CMC) module is modeled as a top-level function called `cmc` and the multiply-accumulate (MAC) circuit is modeled as a function called `mac` which is used in the top level. As shown in Listing 16, `mac` is a stateful function as the MAC circuit requires a register to store the current result temporarily for the next iteration. `s` and `s’` indicate the old and new states respectively.

```
mac s (x, y) = (s’, out)
    where
        s’   = x*y + s
        out = s
```

Listing 16. Definition of mac

Figure 28 illustrates a graphical representation of the `cmc` function according to its definition shown in Listing 17. First, it makes 10 combinations of the input signals in a list `pairs` by indexing the same list `ys` with two different index numbers `i1` and `i2` (Line 3-4) where `!!` is the indexing operator of lists in Haskell. Then it applies `mac` pairwise to the elements of `ss` and `pairs` where `ss` is a list of the old states: s_1, s_2,…, s_{10}. The output of `cmc` are two lists: `ss’` and `rs` (Line 5) where `ss’` is a list of the new states: ${s'}_1$, ${s'}_2$,…, ${s'}_{10}$ and `rs` is the upper triangle of the covariance matrix.

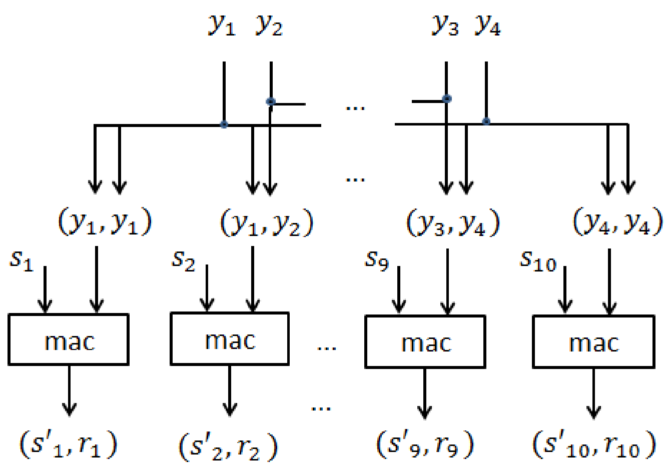

Figure 28. Structure of cmc

```
cmc ss ys = (ss',rs)
  where
      pairs   = [(ys !! i1, ys !! i2) |
                    i1 <- [0..3], i2 <- [0..3], i1 <= i2]
    (ss',rs) =  unzip $ zipWith mac ss pairs
```

Listing 17. Definition of cmc

5.1.2 Testing

In Haskell, a function that represents a sequential synchronous circuit can be simulated by the `simulate` function as defined in Listing 18. It recursively applies a function `f` to the state `s` and an element of the list `(x:xs)` till the end of the list, where the `:` operator adds an element to the head of a list. The list `(x:xs)` imitates an input signal that lasts for several clock cycles and each application of `f` simulates the behavior of the synchronous circuit in one clock cycle.

```
simulate f s [] = []
simulate f s (x : xs) = y : simulate f s' xs
    where
        (s', y) = f s x
```

Listing 18. Definition of simulate

As shown in Listing 19, the `cmc` function is simulated by the `simulate` function with an initial state `s_init` which is a list of 10 zeros (Line 1). `inps` (Line 2) is a list of lists where each sub-list is an input of `cmc`. Figure 29 shows the content of `test` which is a list of the simulation results in GHCI, a GHC (Glasgow Haskell Compiler) interactive environment. Note that the output is delayed by one clock cycle: the first output is the initial state. Therefore, the third sub-list of `inps`, i.e. `[7,8,9,10]`, does not affect the simulation result.

```
1    s_init = replicate 10 0
2    inps = [[1,2,3,4],[5,6,7,8],[7,8,9,10]]
3    test = simulate cmc s_init inps
```

Listing 19. Simulation of cmc in Haskell

```
Prelude> :l CMC.hs
[1 of 1] Compiling Main             ( CMC.hs, interpreted )
Ok, modules loaded: Main.
*Main> test
[[0,0,0,0,0,0,0,0,0,0],[1,2,3,4,4,6,8,9,12,16],[26,32,38,44,40,48,56,58,68,80]]
```

Figure 29. Simulation result of cmc in Haskell

Since the covariance matrix calculation is in principle the multiplication of a vector and its transpose, the simulation results can be verified with the transpose operator ' in MATLAB, as shown in Listing 20.

```
inp = [1,2,3,4;5,6,7,8]
outp = inp'*inp
```

Listing 20. CMC in MATLAB

After the CλaSH implementation is tested, the corresponding VHDL code is generated as well as a test bench. Figure 30 shows the simulation result of the generated VHDL code in ModelSim where `clk1000` is a 1 MHz clock signal, `inp_i1` contains the 4 input values and `topLet_o` is the output signal. In the test bench, the input values are assigned to be 1,2,3,4 at 100 ns (in the first clock cycle) and 5,6,7,8 after 1200 ns (in the second clock cycle). According to the definition of `mac` shown in Listing 16, each output of `cmc` is also the current state. Therefore, the output values are updated on every rising edge of the clock signal, as shown in Figure 30.

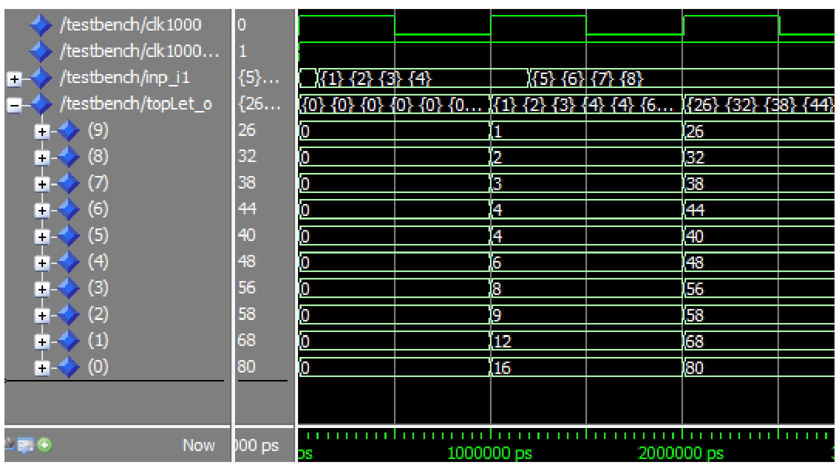

Figure 30. Simulation result of CMC in ModelSim

5.2 Eigenvalue Decomposition

5.2.1 CλaSH Implementation

The eigen-decomposition (EVD) module is modeled as a top-level function called `evd`. As shown in Listing 21, the `evd` function takes a list `rs` containing the upper triangle produced by the `cmc` function to calculate its eigenvalues `evals` and eigenvectors `evecs`. `s` indicates a state containing the intermediate results of each iteration. As shown in Figure 31, the EVD module consists of several components such as `CORDIC_A`, `update` and `CORDIC_B`. Each component is modeled as a function which is used in the top level. The complete definition of `evd` can be found in Appendix B.

```
evd s rs = (s', (evals, evecs))
```

Listing 21. Definition of evd

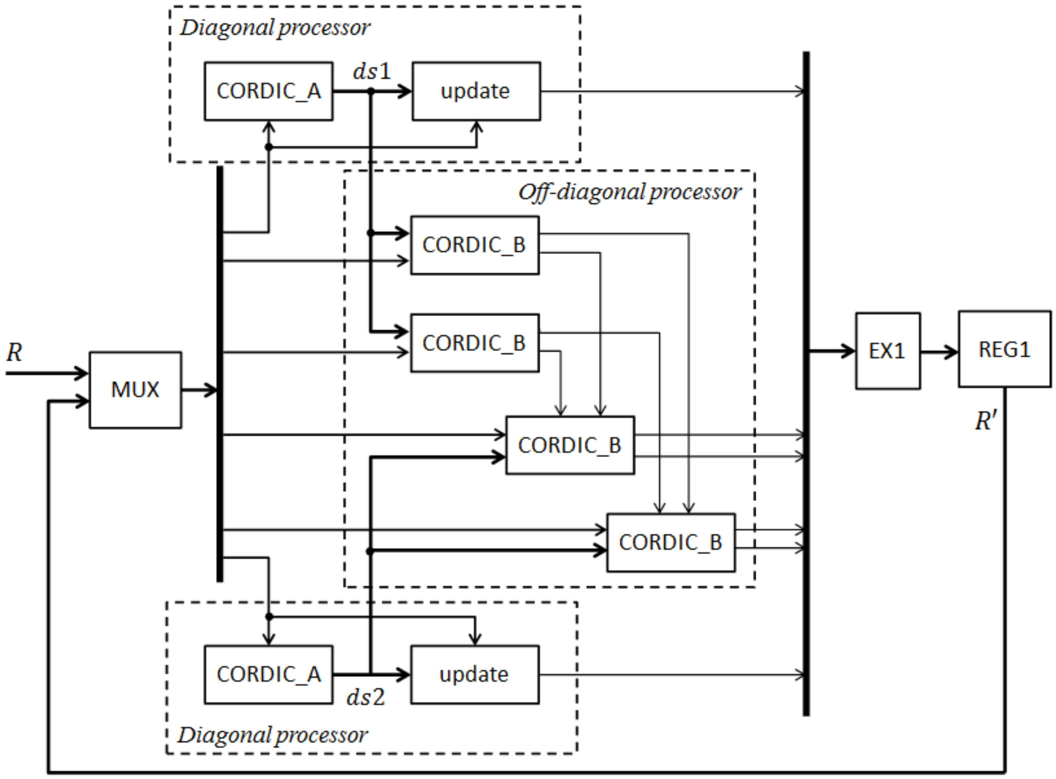

Figure 31. EVD architecture

As defined in Listing 22, the `ca1` function describes one iteration of the CORDIC_A algorithm according to Eq. (32). In the CλaSH implementation, the power of two operations in Line 3-4 will be implemented with the bit shift functions `shiftR` and `shiftL`. The `getSign` function (Line 8-9) determines the rotation direction `di` according to the signs of the two inputs.

```
1    ca1 (ri,ui) i = ((ri',ui'),di')
2      where
3        ri'  =  (1-2^(-2*i))*ri - di*(2^(1-i))*ui
4        ui'  =  (1-2^(-2*i))*ui + di*(2^(1-i))*ri
5        di   =  getSign ri ui
6
7    getSign x y = if x/y >= 0 then 1
8                  else -1
```

Listing 22. Haskell definition of cordica

Figure 32 illustrates a graphical representation of a CORDIC_A implementation with 10 iterations of the `ca1` function. As mentioned in Sec. 4.3.1, the accuracy of the CORDIC algorithm depends on the number of iterations. In this project, we perform 10 iterations as it shows a satisfactory accuracy. The structure shown in Figure 32 can be described by `foldl` with a slight modification to the function definition of `ca1` because `foldl` requires that the first input and the output of the function are of the same type. The input of `ca1` is a 2-tuple but the output is a 3-tuple. As shown in Listing 23, the first input of the modified `ca1`, namely `ca2`, is a 3-tuple of which the third element is a list `dsi` and the output is also a 3-tuple. The operator `:` (Line 7) appends the new direction value `di'` to the list `dsi` and the new list `dsi'` is the third element of the output. Figure 33 shows the structure of the CORDIC_A implementation with the `ca2` function and it can be described by the `cordic_a` function as shown in Listing 24, where `ids` is a list of index numbers in the range of 0 to 9 and `ds` is initialized with `[]`, an empty list.

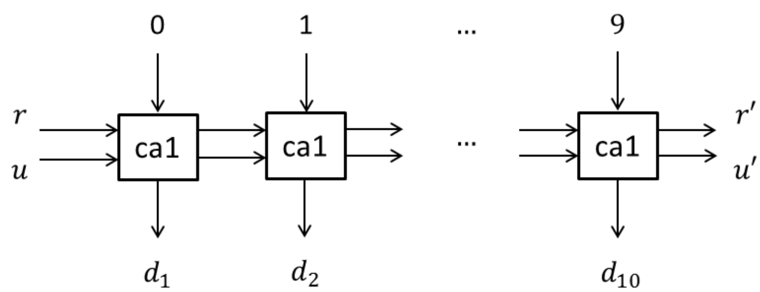

Figure 32. Structure of CORDIC_A

```
1    ca2 (ri,ui,dsi) i = (r',u',dsi')
2        where
3            ri'  =  (1-2^(-2*i))*ri - di*(2^(1-i))*ui
4            ui'  =  (1-2^(-2*i))*ui + di*(2^(1-i))*ri
5            di   =  getSign r u
6            dsi' =  di : dsi
7
8    getSign x y = if x/y >= 0 then 1
9                  else -1
```

Listing 23. Haskell definition of modified cordica

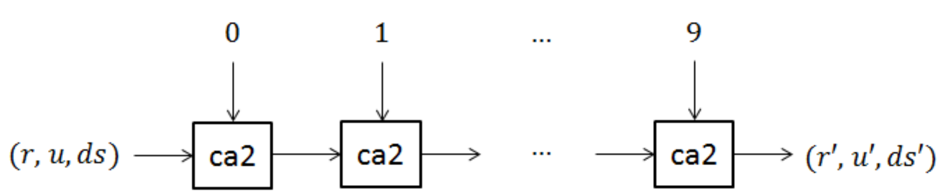

Figure 33. Modified structure of CORDIC_A

```
cordic_a r u = ds
  where
    ids = [0..9]
    (r',u',ds) = foldl ca2 (r,u,[]) ids
```

Listing 24. Definition of cordic_a with foldl

In fact, the structure shown in Figure 32 can be directly described by another built-in higher-order function: `mapAccumL` without modifying the definition of `ca1`, as shown in Listing 25. The `mapAccumL` function behaves like a combination of `map` and `foldl`. It applies a function which is `ca1` in this case, to each element of a list `ids`, passing an accumulating parameter `(r, u)` from left to right, and returning a final value of this accumulator together with the new list `ds`.

```
cordic_a r u = ds
  where
     ids = [0..9]
     ((r',u'),ds) = mapAccumL ca1 (r,u) ids
```

Listing 25. Definition of cordic_a with mapAccumL

Listing 26 shows the definition of the `update` function which takes a list `ds` produced by `cordic_a` and updates the diagonal elements `b` and `c` according to Eq. (36). `tanv` is a tangent value obtained from a list of tangent values created by the `lut` function and the index `ind` is an integer converted from `ds` (Line 5).

```
1    update (a, b, c) ds = (b', c')
2        where
3             b' = b + tanv * a
4             c' = c - tanv * a
5           ind  = toInt ds
6           tanv = (lut 10) !! ind
```

Listing 26. Haskell definition of update

Figure 34 is a graphical representation of the `lut` function defined in Listing 27. First, the `css` function creates a list of lists by recursively applying list comprehension and concatenation (Line 3-4). According to the results of `css 1` and `css 2` shown in Listing 28, it can be concluded that `css n` creates a list of 2^n lists where each sub-list contains n values being either 1 or -1. Then each sub-list produced by `css` is applied with the `tangent` function (Line 6) to calculate the corresponding tangent value, where `bs` is a list of rotation angles in radians (Line 7-8) and the `$` symbol is used to replace the brackets. Note that the implementation of `lut` will remain pure Haskell in the CλaSH implementation because it creates a list of constant numbers that are known at compile time.

```
1    lut n = map tangent (css n)
2
3    css 0 = [[]]
4    css n = concat [[-1:cs, 1:cs] | cs <- css (n-1)]
5
6    tangent cs = tan $ sum $ zipWith (*) bs cs
7    as = [45.0,26.6,14.0,7.1,3.6,1.8,0.9,0.4,0.2,0.1]
8    bs = [pi/180*x | x <- as]
```

Listing 27. Haskell definition of lut

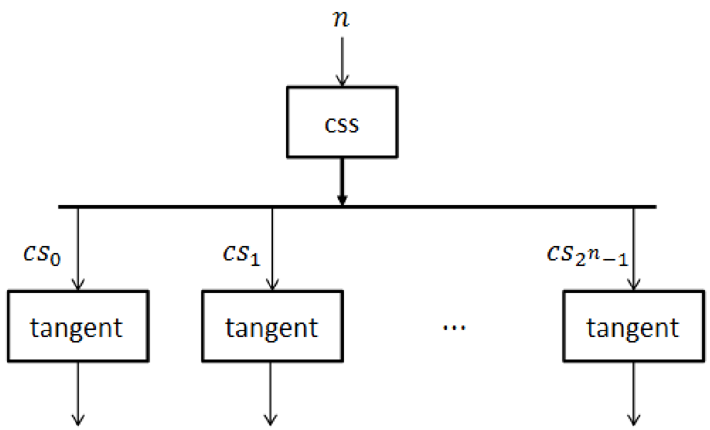

Figure 34. lut

```
css 1 = concat [[-1:cs,1:cs] | cs <- [[]]]
      = concat [[[-1],[1]]]
      = [[-1],[1]]

css 2 = concat [[-1:cs,1:cs] | cs <- css 1]
      = concat [[[-1,-1],[1,-1]],[[-1,1],[1,1]]]
      = [[-1,-1],[1,-1],[-1,1],[1,1]]
```

Listing 28. Examples of css

Listing 29 presents the definition of the `cordic_b` function which implements the CORDIC_B algorithm. As shown in Figure 35, `cordic_b` iteratively applies the `cb` function with an element of the list `ds` which is produced by `cordic_a` to update the off-diagonal elements `x` and `y`. The `cb` function, as defined in Listing 30, describes one iteration of the CORDIC_B algorithm according to Eq. (18).

```
cordic_b (x, y) ds = (x', y')
    where
        ids = [0..9]
        (x', y') = foldl cb (x, y) (zip ids ds)
```

Listing 29. Definition of cordic_b

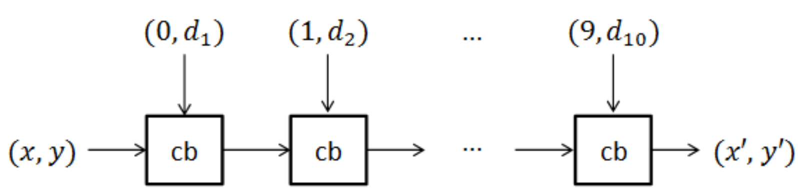

Figure 35. Structure of cordic_b

```
cb (xi, yi) (i, di) = (xi', yi')
    where
        xi' = xi - di*(2^(-i))*yi
        yi' = yi - di*(2^(-i))*xi
```

Listing 30. Definition of cb

5.2.2 Testing

Since `evd` is a stateful function, it can be simulated by the `simulate` function as shown in Listing 31 where `s_init` indicates an initial state and `rs` is the upper triangle shown in Eq. (38). Figure 36 presents the simulation results of the first 10 clock cycles. As shown in Figure 37, the simulation result of each clock cycle consists of three components: a list of eigenvalues ($< \cdots >$ denotes a list), a list of eigenvectors (each eigenvector is a sub-list) and an additional output `end`. When `end` becomes 1, meaning all the off-diagonal elements are (nearly) zeros, the EVD computation is finished. In this case, it takes 8 clock cycles to finish the computation. Note that the eigenvectors are initialized with an identity matrix multiplied by 1000, therefore the results of the eigenvectors are also scaled by 1000.

```
inps = replicate 10 rs
test = simulate evd s_init inps
```

Listing 31. Simulation of evd

$$R = \begin{bmatrix} 1261 & -401 & 859 & 247 \\ & 1403 & -715 & 189 \\ & & -160 & 87 \\ & & & 541 \end{bmatrix} \tag{38}$$

```
*EVD> take 10 test
[(<1261,1403,-160,541>,<<1000,0,0,0>,<0,1000,0,0>,<0,0,1000,0>,<0,0,0,1000>>,0),(<9
26,-170,551,1738>,<<768,641,0,0>,<0,0,993,-121>,<0,0,121,992>,<-642,768,0,0>>,0),(<
949,534,1755,-193>,<<762,635,140,-17>,<-77,94,121,986>,<-638,763,-15,-119>,<-108,-9
1,984,-119>>,0),(<1115,2227,-665,368>,<<636,604,180,450>,<-539,735,-412,-59>,<-358,
226,895,-156>,<-427,-216,41,878>>,0),(<1102,-675,378,2240>,<<573,681,136,441>,<-398
,204,894,-70>,<-390,-238,-46,889>,<-604,666,-429,-107>>,0),(<1114,367,2251,-687>,<<
604,662,63,447>,<-341,-291,-12,895>,<-633,646,-433,-38>,<-350,259,903,-34>>,0),(<11
16,2252,-688,365>,<<589,650,62,483>,<-638,649,-421,-38>,<-341,252,910,-33>,<-366,-3
16,-14,878>>,0),(<1116,-688,365,2252>,<<589,651,61,482>,<-340,252,911,-33>,<-368,-3
16,-14,878>,<-638,650,-421,-38>>,0),(<1117,365,2252,-689>,<<589,651,62,481>,<-367,-
317,-14,878>,<-640,650,-422,-37>,<-340,252,912,-33>>,1),(<1117,365,2252,-689>,<<589
,651,62,481>,<-367,-317,-14,878>,<-640,650,-422,-37>,<-340,252,912,-33>>,1)]
```

Figure 36. Simulation results of evd

$$(\underbrace{< \cdots >}_{\text{eigenvalues}}, \underbrace{<< \cdots >, < \cdots >, < \cdots >, < \cdots >>}_{\text{eigenvectors}}, end)$$

Figure 37. Components of simulation result

The simulation results can be verified in MATLAB with the built-in function `eig`, as shown in Listing 32. `evals`, as shown in Eq. (39), is a diagonal matrix of which the diagonal elements are the eigenvalues of the matrix `R` and `evecs`, as shown in Eq. (40), is a matrix of which each column is a corresponding eigenvector. It can be observed that the simulation result of the Haskell code is very close to the result in MATLAB and the average error is about 0.5% which is mainly caused by the fixed-point operations such as bitwise right shifts.

```
[evecs, evals] = eig (R)
```

Listing 32. EVD in MATLAB

$$evals = \begin{bmatrix} -685 & & & \\ & 368 & & \\ & & 1111 & \\ & & & 2251 \end{bmatrix} \tag{39}$$

$$evecs = 10^{-3} \times \begin{bmatrix} 345 & 364 & 586 & 636 \\ -247 & 316 & 650 & -646 \\ -905 & 21 & 63 & 420 \\ 33 & -876 & 479 & 42 \end{bmatrix} \tag{40}$$

The VHDL code generated from the CλaSH implementation is simulated in ModelSim with the same input. As shown in Figure 38, `inp_i1` is an input signal which contains the upper triangle shown in Eq. (38). The output signal `topLet_0` consists of 3 components: `product9_sel0` contains the eigenvalues, the corresponding eigenvectors are presented in `product9_sel1` and `product9_sel2` becomes high when the EVD computation is finished. As `clk1000` is a 1 MHz clock signal, it takes 8 clock cycles to finish the computation. The simulation result of the VHDL code is also very close to the result in MATLAB.

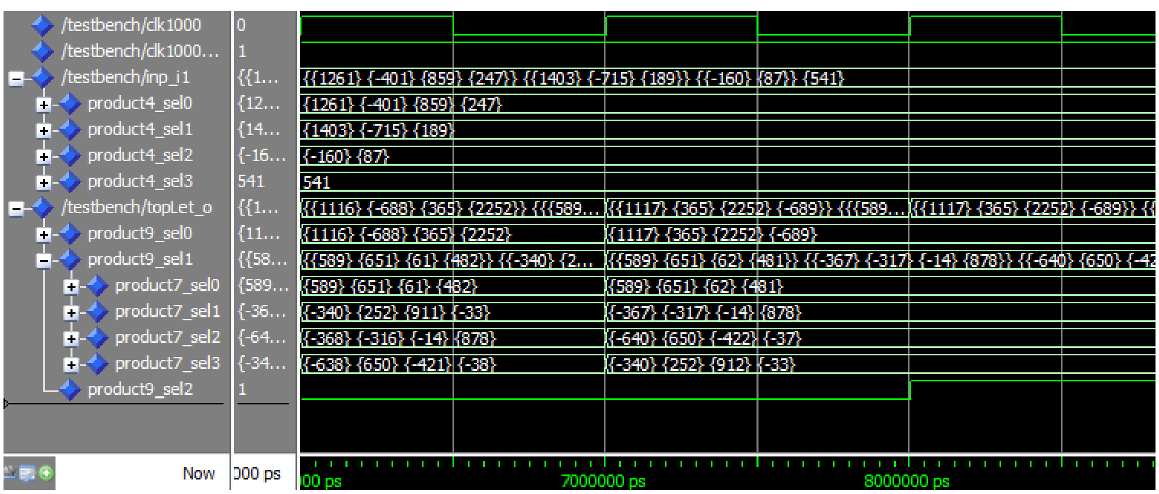

Figure 38. Simulation result of EVD

5.3 Spectral Peak Search

5.3.1 CλaSH Implementation

The spectral peak search module is modeled as a top-level function called `sps` which takes the eigenvalues `evals` and eigenvectors `evecs` produced by the `evd` function and outputs the index of the DOA (Direction of Arrival), as shown in Listing 33. First, the `eigsort` function finds the index of the maximum eigenvalue and the corresponding eigenvector `evec` is taken from `evecs` with this index (Line 3). Then the `norm` function calculates the norm based on `evec` and sv which is a steering vector with the index s3 stored in the look-up table svlut. The comp1 function finds the peak from 3 consecutive norm values (one is the current norm value and the other two are the previous values stored in the state s1) and comp2 compares the current peak with the previous one stored in s2. The second element of s2', i.e., the index of the maximum peak, is the output of sps (Line 10).

```
1  sps (s1,s2,s3) (evals, evecs) = ((s1',s2',s3'), ind)
2      where
3          evec = evecs !! (eigsort evals)
4            sv = svlut !! s3
5         normv = norm evec sv
6           tmp = comp1 s1 (normv, s3)
7           s1' = init & (normv, s3) : 1
8           s2' = comp2 s2 tmp
9           s3' = s3 + 1
10          ind = snd s2'
```

Listing 33. Definition of sps

The eigsort function defined in Listing 34 sorts the eigenvalues in the list evals and outputs the index of the maximum one. eigsort has a foldl structure as shown in Figure 39 where the sort function iteratively inserts each element of ys which contains an eigenvalue with its index (Line 4) to a sorted list which is initialized with an empty list [] and outputs the new sorted list. The sort function itself also has a foldl structure as shown in Figure 40, where the cswap function (Line 5-6, Listing 35) iteratively compares y with each element of vsi and inserts the larger one into a list which is initialized with an empty list.

```
1  eigsort evals = ind
2      where
3          inds = [0..3]
4            ys = zip evals inds
5           vs' = foldl sort [] ys
6           ind = snd $ last vs'
```

Listing 34. Definition of eigsort

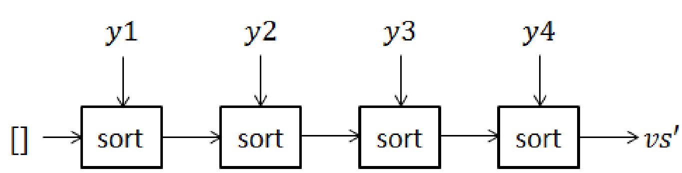

Figure 39. Structure of eigsort

```
1  sort vsi y = vsi'
2      where
3          (y',vsi') = foldl cswap (y,[]) vsi
4
5  cswap (yi, ts) vi = if fst yi > fst vi then (vi, yi : ts)
6                      else (yi, vi : ts)
```

Listing 35. Definition of sort

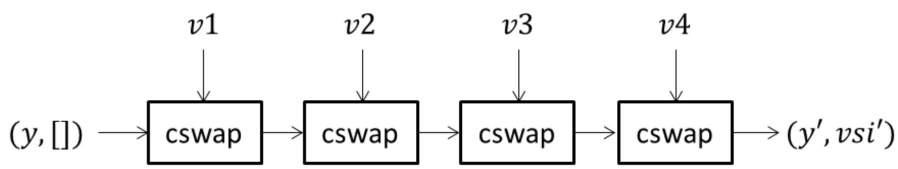

Figure 40. Structure of sort

The `svlut` function shown in Listing 36 creates a list of steering vectors according to Eq. (41). The `++` operator (Line 1) is used to append two lists.

```
1  svlut = [[cos $ (a*) $ sin b | a <- as] ++
2            [sin $ (a*) $ sin b | a <- as] | b <- bs]
3
4  as = [(2*n-1]*pi/2 | n <- [1,2]]
5  bs = [n/512*pi | n <- [0..255]]
```

Listing 36. SvLUT implementation

$$a^H(\theta_k) = [\cos\left(\frac{\pi}{2} sin\theta_k\right), \cos\left(\frac{3\pi}{2} sin\theta_k\right), \sin\left(\frac{\pi}{2} sin\theta_k\right), \sin\left(\frac{3\pi}{2} sin\theta_k\right)] \quad (41)$$

Listing 37 shows the definition of the function `norm` which calculates the norm, i.e. the square of the dot product of two lists. Figure 41 is a graphical representation of the dot product function `dotp` (Line 3-5) which pairwise multiplies the elements of two lists and outputs the sum of the multiplication results.

```
1    norm xs ys = (dotp xs ys)^2
2
3    dotp xs ys = foldl (+) 0 ws
4        where
5            ws = zipWith (*) xs ys
```

Listing 37. Haskell definition of norm

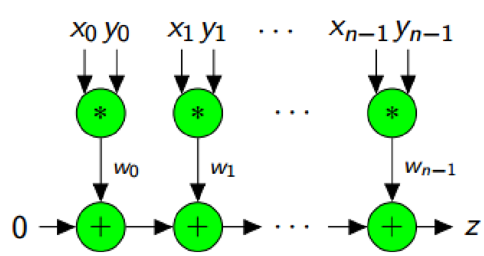

Figure 41. Dot product

The `comp1` function (Line1-2, Listing 38) finds the peak by comparing three consecutive input values `x1`, `x2` and `x3`: if `x2` is larger than both `x1` and `x3`, the output will be `x2` with its index, otherwise the output is (0, 0). Then `comp2` (Line 5-6, Listing 38) compares the current peak `p2` with the previous peak `p1` and outputs the index of the larger one.

```
1  comp1 (x1, x2) (x3, ind3) = if x2 > x3 && x2 > x1
2                              then (x2, ind2)
```

```
                               else (0,0)

comp2 (p1,ind1) (p2,ind2) = if p2 > p1 then ind2
                               else ind1
```

Listing 38. comp1 and comp2 in Haskell

5.3.2 Testing

To simulate the sps function, a signal model is created in MATLAB as shown in Listing 39 where the source signal has a DOA of $\frac{\pi}{6}$ (Line 4) and the final results are the eigenvalues and eigenvectors of the covariance matrix (Line 14) based on this signal model. Since sps is also a stateful function, it can be simulated by the simulate function as shown in Listing 40, where evals and evecs are the results of the MATLAB program and they are applied to sps 256 times since there are 256 possible DOAs according to Sec. 4.4. Figure 42 presents the simulation result which is 89. The corresponding angle value can be calculated as $\frac{89}{512} \times \pi \approx \frac{\pi}{6}$ according to Eq. (37).

```
1  M = 4;                              % number of antennas
2  N = 256;                            % number of snapshots
3  d = 0.5;
4  theta = pi/6;                       % DOA
5  f = 0.2;
6  snr = 10;                            % SNR = 10 dB
7  s = cos(2*pi*f*n);                   % source signal
8  alpha = pi*d*sin(theta);
9  % steering vector
10 A = [cos(alpha),cos(3*alpha),sin(alpha),sin(3*alpha)]';
11 x0 = A*s;
12 x = sqrt(10^(snr/10))*x0+randn(M,N)     % signal data matrix
13 R = x*x'/N;                             % covariance matrix
14 [V,D] = eig (R)                         % EVD
```

Listing 39. Signal model in MATLAB

```
inps = replicate 256 (evals, evecs)
test = last $ simulate sps s_init inps
```

Listing 40. Simulation of sps

```
*SPS> test
89
*SPS>
```

Figure 42. Simulation result of sps

The VHDL code generated from the CλaSH implementation is simulated in ModelSim with the same eigenvalues and eigenvectors produced by the MATLAB program. As shown in Figure 43, the input signal inp_i1 has 2 components: product3_sel0 and product3_sel1 which contains the eigenvalues and the corresponding eigenvectors respectively. topLet_o presents the final result which is also 89.

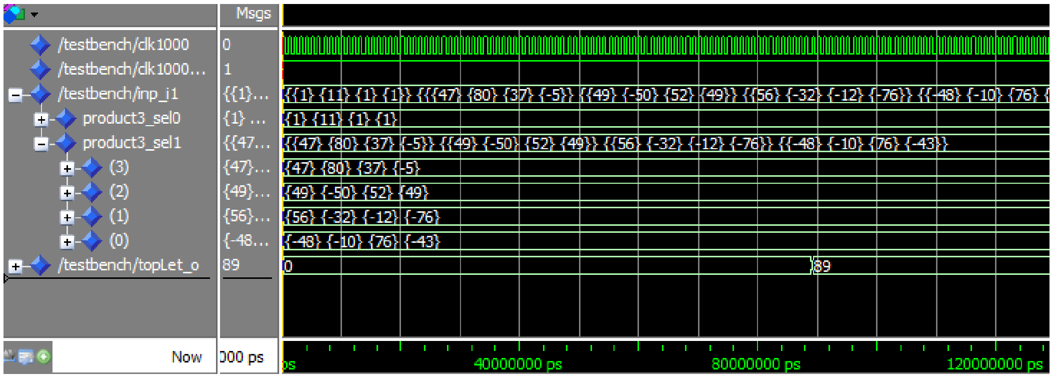

Figure 43. Simulation result of SPS in ModelSim

6. Evaluation

6.1 Hardware Description

In this section, we will discuss about the advantages and disadvantages of using CλaSH for hardware descriptions based on the implementation of the MUSIC algorithm presented in Chapter 5.

It can be found in Chapter 5 that the built-in higher-order functions such as `map`, `foldl` and `zipWith` play an important role in the implementation of the MUSIC algorithm. Many commonly used hardware structures can be described by these higher-order functions in a high abstraction level, which significantly reduces the amount of code. For example, if the CMC (Covariance Matrix Calculation) module is implemented in VHDL, each MAC (Multiply-accumulate) component has to be instantiated, which requires a large amount of code. In Haskell it can be implemented by the `zipWith` function in one line as shown in Listing 17. Although one can use a `for-generate` expression in VHDL to finish the instantiations in a for-loop, it is still not as concise as the higher-order function. Sometimes the same algorithm can be described by different built-in higher-order functions: as we discussed about the implementation of the CORDIC_A algorithm in Sec. 5.2.1, it can be implemented by either the `mapAccumL` function or the `foldl` function with a slight modification to the `ca1` function.

Besides the built-in higher-order functions, a user-defined function can also take other functions as parameters, which is a very powerful feature of CλaSH. Figure 44 is a graphical representation of the `dotp` function which calculates the dot product of two vectors, as defined in Listing 41. If the * operator and the + operator are represented by `f` and `g` respectively, as shown in Figure 45, this architecture can be described by the `arch` function defined in Listing 42 where `f` and `g` are taken as two parameters. Then the `dotp` function becomes an instance of the `arch` function, as shown in Listing 43. As `f` and `g` can be any function that takes two input values and outputs one value, the `arch` function can be used to describe all the hardware circuits with this architecture, which can reduce the amount of code and save the development time.

```
dotp xs ys = z
  where
     ws = zipWith (*) xs ys
      z = foldl (+) 0 ws
```

Listing 41. Definition of dotp

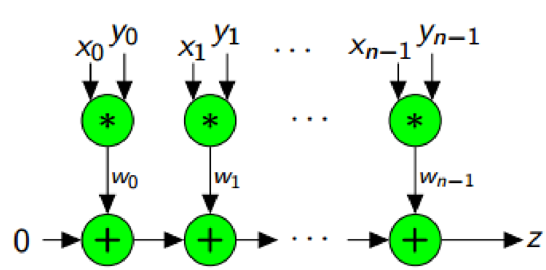

Figure 44. Architecture of dotp

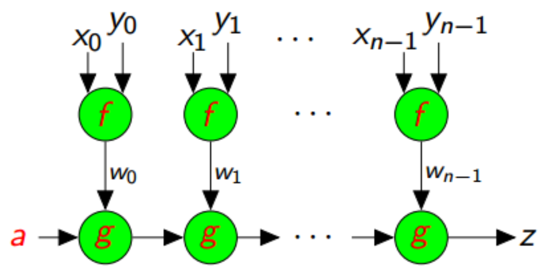

Figure 45. dotp-like architecture

```
arch f g a xs ys = z
  where
     ws = zipWith f xs ys
      z = foldl g a ws
```

Listing 42. Definition of arch

```
dotp xs ys = arch (*) (+) 0 xs ys
```

Listing 43. Definition of dotp with arch

In the implementation of the EVD module, a look-up table (LUT) of tangent values is created by list comprehension. Listing 44 shows a simple example which creates a LUT of tangent values of 256 angles in the range of $[0, \pi/2]$. The VHDL implementation of such a LUT usually takes two steps: first, calculate the tangent values in MATLAB (or other tools); secondly, assign these values to an array in VHDL. An alternative way is to use the `TAN` function provided by the MATH_REAL package. Unlike the `tan` function in Haskell, the `TAN` function in VHDL does not accept a parameterized input, which means each angle value has to be calculated first. Both the two ways in VHDL are not as easy as the Haskell implementation and are more time-consuming.

```
[tan pi/512*x | x <- [0..255]
```

Listing 44. LUT of tangent values

The CλaSH complier which is based on the GHC (Glasgow Haskell Compiler) provides an interactive user interface where one can test the Haskell implementation with the `simulate` function. In contrast, to test a VHDL implementation, one has to make a test bench which is then simulated in a simulation tool such as ModelSim.

Although CλaSH has many advantages, it still needs to be improved. Currently the CλaSH complier updates the state of a sequential circuit on every rising edge of the clock signal, while in VHDL one can also choose to update the state on the falling edges. Therefore, VHDL is better at describing the timing behavior. As some Haskell syntactic constructs such as list comprehensions are not supported by CλaSH (yet), in many cases, the conversion from a Haskell implementation to a CλaSH implementation is not straightforward. According to Sec. 5.1.1, a list comprehension is used in the Haskell implementation of the `cmc` function, as shown in Listing 45. It is difficult for the compiler to predict the hardware cost of this list comprehension as there is a filter `i1 <= i2`, which means it cannot be directly used in CλaSH. The solution to this problem can be found in Appendix A. On the other hand, list comprehensions without a filter should be supported by CλaSH as they have predictable hardware cost at compile time. And the CλaSH compiler is not very efficient in generating VHDL code.

```
pairs  = [(ys !! i1, ys !! i2) |
                i1 <- [0..3], i2 <- [0..3], i1 <= i2]
```

Listing 45. List comprehension with a filter

According to the above discussion, the comparison between the CλaSH implementation and the VHDL implementation is summarized in Table 2 where ++ means "very good", + means "good" and − means "not good".

	Conciseness	Development Time	Description of Timing behavior
CλaSH	++	+	-
VHDL	-	-	++

Table 2. CλaSH vs VHDL

6.2 Synthesis

To evaluate the synthesis results of the CλaSH implementation, a VHDL implementation has been provided by the author of [1] for comparison. However, it is likely that the provided VHDL code does not exactly implement the algorithm according to the hardware designs described in [1] as its simulation result turns out to be very different from the result presented in [1], which makes it not comparable with our CλaSH implementation. The solution is to focus on a smaller design, for example, CORDIC_A, instead of the complete MUSIC algorithm. The CλaSH implementation of the CORDIC_A algorithm shown in Sec. 5.2.1 is a non-pipelined design which finishes the 10 iterations in a long combinational path. However, a pipelined design is chosen for the evaluation of the synthesis result because the synthesis tool which is Quartus II cannot calculate the maximum clock frequency for a pure combinational circuit. As shown in Figure 46, the pipelined CORDIC_A has 10 stages and the result of each stage is stored in a register. Both the CλaSH and VHDL implementations of the pipelined CORDIC_A can be found in Appendix D. As presented in Table 3, the synthesis results of these two implementations are approximately equivalent. The CλaSH implementation uses a few more logic resources and registers than the VHDL implementation but achieves a little higher maximum clock frequency.

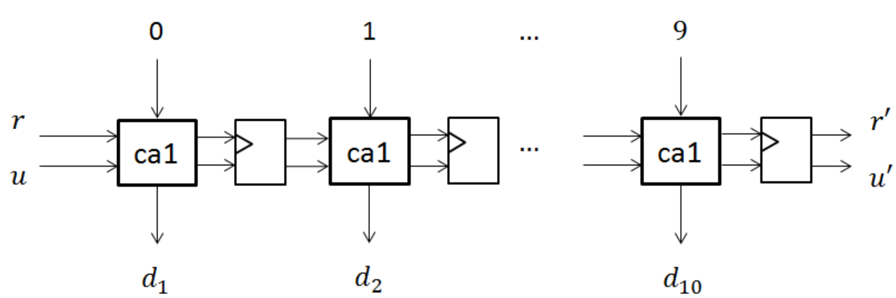

Figure 46. Pipelined CORDIC_A

	Fmax (MHz)	Logic utilization (in ALMs)	Registers	Pins
VHDL	170.68	331 (<1%)	402	76
CλaSH	174.34	342 (<1%)	412	76

Table 3. Synthesis result of CORDIC_A

7. Conclusions

In this project, the MUSIC algorithm is successfully implemented in CλaSH. As the MUSIC algorithm has many non-trivial aspects in hardware implementation, it proves the usability of CλaSH in hardware descriptions. With a higher abstraction level, the CλaSH implementation shows a better code conciseness than the VHDL implementation. The higher-order functions are found very useful in hardware descriptions as they can describe most of the commonly used hardware architectures in a very natural and concise way. Since a higher-order function takes other functions as parameters, the function definition can be reused for many different hardware designs as long as they have the same architecture, which significantly reduces the amount of code and saves the development time. The fact that the CλaSH compiler is also an interactive user interface where the designer can easily simulate the functions makes it more convenient to test a CλaSH implementation than a VHDL implementation which requires a test bench and a simulation tool. Although CλaSH has a limitation in describing the timing behaviors as it updates all states on every rising edge of the clock signal, in most cases this limitation is not a fatal defect. In the future, list comprehensions without a filter should be supported by CλaSH and the efficiency in generating the VHDL code needs to be improved. In general, CλaSH is a very suitable language for hardware descriptions.

References

[1] Tao Wang, "FPGA Implementation of the MUSIC Algorithm," Master's thesis, University of Electronic Science and Technology, Chengdu, China, May 2010. [Online] Available: http://www.docin.com/p-656385779.html

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arian Boeijink and Marco Gerards, "CλaSH: Structural Descriptions of Synchronous Hardware using Haskell," in *Proceedings of the 13th Conference on Digital System Design (DSD)*, Lille, France, Sept 1-3, 2010. pp. 714-721. IEEE Computer Society. ISBN 978-0-7695-4171-6.

[3] Jan Kuper, "Hardware Specification with CλaSH," *in DSL 2013.* Cluj, Romania, July 2013. Available: http://dsl2013.math.ubbcluj.ro/files/DSL13-CLASH.pdf

[4] Keh-Chiarng Huarng and Chien-Chung Yeh, "A Unitary Transformation Method for Angle-of-Arrival Estimation," *Signal Processing, IEEE Transactions on* vol. 39, issue 4, pp. 975-977, Apr, 1991.

[5] Sabih Gerez, "The CORDIC Algorithm and CORDIC Architectures," March, 2009. [Online]. Available: http://wwwhome.ewi.utwente.nl/~gerezsh/sendfile/sendfile.php/idsp-cordic.pdf?sendfile=idsp-cordic.pdf

[6] Zdzislaw Meglicki, "Jacobi Transformations of a Symmetric Matrix," Feb, 2001. [Online]. Available: http://beige.ucs.indiana.edu/B673/node24.html

[7] Tao Wang and Ping Wei, "Hardware Efficient Architectures of Improved Jacobi Method to Solve the Eigen Problem," in *Computer Engineering and Technology, 2010 2nd International Conference on* vol. 6, pp. 22-25, April 2010.

[8] Rinse Wester, Christiaan Baaij, Jan Kuper, "A Two Step Hardware Design Method Using CλaSH," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 29-31, 2012, Oslo, Norway. pages 181-188. IEEE Computer Society. ISBN 978-1-4673-2257-7.

Appendix A: CλaSH code of cmc

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module CMC (topEntity) where

import CLaSH.Prelude

type CMCI = Vec 4 (Signed 16)
type CMCS = Vec 10 (Signed 16)
type CMCO = CMCS

cmcInit :: CMCS
cmcInit = vcopyI 0

topEntity = cmc

cmc ys = rs
   where
         rs = (cmcCore <^> cmcInit) ys

cmcCore :: CMCS -> CMCI -> (CMCS, CMCO)
cmcCore ss ys = (ss', rs)
   where
       inds = vreverse $(v([(0,0),(0,1),(0,2),(0,3),(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] :: [(Int, Int)]))
          pairs = vmap (pair ys) inds
      (ss', rs) = vunzip $ vzipWith mac ss pairs

mac s (x,y) =  (s', out)
   where
         s' = x*y + s
         out = s

pair ys (i1,i2) = (ys ! i1, ys ! i2)
```

Appendix B: CλaSH code of evd

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module EVD (topEntity) where

import CLaSH.Prelude
import CordicA
import CordicB
import Update

type Ev = Vec 4 (Signed 16)
type Col = (Signed 16, Signed 16, Signed 16, Signed 16)
type EvdS = (Bit,EvdI,Matrix,Bit)
type EvdI = ((Signed 16,Signed 16,Signed 16,Signed 16),
```

```
                       (Signed 16,Signed 16,Signed 16),
                                 (Signed 16,Signed 16),
                                            Signed 16)
type EvdO = (Ev,Evecs,Bit)
type Matrix = (Col,Col,Col,Col)
type Evecs = Vec 4 Ev

topEntity = evd

uptri_init :: EvdI
uptri_init = ((0,0,0,0),(0,0,0),(0,0),0)

evsinit :: Matrix
evsinit = (1000,0,0,0),(0,1000,0,0),(0,0,1000,0),(0,0,0,1000))

evd inp = outp
    where
        outp = (evdCore <^> (L,uptri_init,evsinit,L)) inp

evdCore :: EvdS -> EvdI -> (EvdS,EvdO)
evdCore (rst,uptri,evs,end) inp =
((rst',uptri',evs',end'),(evals,evecs,end'))
    where
        rst' = H
        ((e11,e12,e13,e14),
             (e22,e23,e24),
                 (e33,e34),
                      e44) = mux rst inp uptri

        r1 = 2*e12
        u1 = e22 - e11
        r2 = 2*e34
        u2 = e44 - e33

        ds1 = cordic_a r1 u1
        ds2 = cordic_a r2 u2
        (e11',e22') = update (e12, e22, e11) ds1
        (e33',e44') = update (e34, e44, e33) ds2
        (e13_tmp,e23_tmp)  = cordic_b (e13,e23) ds1
        (e14_tmp,e24_tmp)  = cordic_b (e14,e24) ds1
        (e13',e14')        = cordic_b (e13_tmp,e14_tmp) ds2
        (e23',e24')        = cordic_b (e23_tmp,e24_tmp) ds2

        (e12', e34') = (0, 0)

        uptri_tmp = ((e11',e13',e14',e12'),
                          (e33',e34',e23'),
                               (e44',e24'),
                                     e22')

        uptri' = mux end' uptri_tmp uptri
```

```
        evals = e11 :> e22 :> e33 :> e44 :> Nil

        ( (v11,v21,v31,v41),
          (v12,v22,v32,v42),
          (v13,v23,v33,v43),
          (v14,v24,v34,v44) ) = evs

        evs' = mux end' ( (v11',v21',v31',v41'),
                          (v13',v23',v33',v43'),
                          (v14',v24',v34',v44'),
                          (v12',v22',v32',v42') ) evs

        (v11',v12') = cordic_b (v11,v12) ds1
        (v21',v22') = cordic_b (v21,v22) ds1
        (v31',v32') = cordic_b (v31,v32) ds1
        (v41',v42') = cordic_b (v41,v42) ds1

        (v13',v14') = cordic_b (v13,v14) ds2
        (v23',v24') = cordic_b (v23,v24) ds2
        (v33',v34') = cordic_b (v33,v34) ds2
        (v43',v44') = cordic_b (v43,v44) ds2

        evec1 = v11 :> v21 :> v31 :> v41 :> Nil
        evec2 = v12 :> v22 :> v32 :> v42 :> Nil
        evec3 = v13 :> v23 :> v33 :> v43 :> Nil
        evec4 = v14 :> v24 :> v34 :> v44 :> Nil

        evecs = evec1 :> evec2 :> evec3 :> evec4 :> Nil

        end' = if (e12,e13,e14,e23,e24,e34)==(0,0,0,0,0,0)
then H
                else L

mux s a b = if s == L then a
              else b

 stimuli = (replicate 10 ((1261,-401,859,247),(1403,-
715,189),(-160,87),541))
  test = simulate (pack.evd.unpack)stimuli :: [EvdO]
```

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds
#-}
module CordicA (cordic_a) where

import CLaSH.Prelude
import Resize

type CordicI = (Signed 16, Signed 16)
type CordicO = Vec 10 Bit
```

```
dsinit :: Vec 10 Bit
dsinit = vcopyI H

cordic_a :: Signed 16 -> Signed 16 -> CordicO
cordic_a r u = ds
   where
         ids = $(v ([1..10]::[Int]))
         ((r',u'),ds) = vmapAccumL ca (r,u) ids
         --(r',u',ds) = vfoldl ca (r,u,dsinit) ids

-- core function
ca (ri,ui) i = ((ri',ui'),di)
   where
         p1 = myshiftR ri ((i-1)*2)
         q1 = ri - p1
         q2 = myshiftR ui i
         q3 = myshiftR ri i
         p2 = myshiftR ui ((i-1)*2)
         q4 = ui - p2
         di = getSign ri ui
         ri' = addSub di q1 (shiftL q2 2)
         ui' = addSub (complement di) q4 (shiftL q3 2)

-- rotate according to the direction : H/L
addSub L a b = a + b
addSub H a b = a - b

-- determine rotation direction
getSign x y = if vhead (toBV x) == vhead (toBV y) then H
              else L
```

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module Update (update) where

import CLaSH.Prelude
import TanLUT
import Resize

type UpdateI1 = (Signed 16, Signed 16, Signed 16)
type UpdateI2 = Vec 10 Bit
type UpdateO  = (Signed 16, Signed 16)

update :: UpdateI1 -> UpdateI2 -> UpdateO
update (a, b, c) ds = (b', c')
    where
        tanv = getTan (fromBV (vreverse ds))
        tmp = mytrunc $ scale $ (myext a)*(myext tanv)
        b' = b + tmp
```

```
        c' = c - tmp

getTan :: Unsigned 10 -> Signed 16
getTan n = vreverse $(v (lut 10)) ! n

scale x = shiftR x 10
```

```
module TanLUT (lut) where

css 0 = [[]]
css n = concat [[-1:cs, 1:cs] | cs <- css (n-1)]

tangent cs = truncate $ 1024 * (tan $ sum $ zipWith (*) bs cs)

lut :: Int -> [Int]
lut n = map tangent (css n)

as = [45.0, 26.6, 14.0, 7.1, 3.6, 1.8, 0.9, 0.4, 0.2, 0.1]
bs = [pi/180*x | x<-as]
```

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds
#-}
module CordicB (cordic_b) where

import CLaSH.Prelude

type CordicI1 = (Signed 16, Signed 16)
type CordicI2 = Vec 10 Bit
type CordicO = (Signed 16, Signed 16)

cordic_b :: CordicI1 -> CordicI2 -> CordicO
cordic_b (x, y) ds = (x', y')
   where
        ids = $(v ([1..10]::[Int]))
        (xtmp,ytmp) = vfoldl cb (x, y) (vzip ids ds)
        x' = scale xtmp
        y' = scale ytmp

cb (xi,yi) (ind,di) = (xi',yi')
   where
        q5 = shiftR yi (ind-1)
        q6 = shiftR xi (ind-1)
        xi' = addSub di xi q5
        yi' = addSub (complement di) yi q6

-- scale by 1/K = 0.6073
scale x = y
   where
        s1 = shiftR x 1
```

```
        s2 = shiftR x 3
        s3 = shiftR x 6
        s4 = shiftR x 9
        s5 = shiftR x 13
        m1 = s1 + s2
        m2 = s3 + s4 + s5
        y  = m1 - m2

-- rotate according to the direction : H/L
addSub L a b = a + b
addSub H a b = a - b
```

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module Resize (myshiftR,myext,mytrunc) where

import CLaSH.Prelude

myshiftR :: Signed 16 -> Int -> Signed 16
myshiftR inp n | n == 0                     = inp
               | n > 15                     = 0
               | (toBV inp)!(n-1) == H      = (shiftR inp n)+1
               | otherwise                  = (shiftR inp n)


myext :: Signed 16 -> Signed 32
myext x = resize x

mytrunc:: Signed 32 -> Signed 16
mytrunc x = resize x
```

Appendix C: CλaSH code of sps

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module SPS (topEntity) where

import CLaSH.Prelude
import SvLUT
import Norm
import EigSort

topEntity = sps

type Row = Vec 4 (Signed 16)
type Matrix = Vec 4 Row
type SPSS = (Comp1S,Comp2S,Unsigned 8)
type SPSI = (Row,Matrix)
type SPSO = Unsigned 8
```

```
type Comp1S = Vec 2 (Signed 16, Unsigned 8)
type Comp2S = (Signed 16, Unsigned 8)

comp1Init :: Comp1S
comp1Init = vcopyI (0, 0)
comp2Init :: Comp2S
comp2Init = (0, 0)

sps inp = outp
   where
         outp = (spsCore <^> (comp1Init,comp2Init,0)) inp

spsCore :: SPSS -> SPSI -> (SPSS,SPSO)
spsCore (s1,s2,s3) (evals,evecs) = ((s1',s2',s3'), ind)
   where
        evec = (vreverse evecs) ! (eigsort evals)
        sv = vreverse $(mv svlut) ! s3
        normv = norm evec sv
        tmp = comp1 s1 (normv,s3)
        s1' = (normv,s3) +>> s1
        s2' = comp2 s2 tmp
        s3' = s3+1
        ind = snd s2'

-- comp1
comp1 s (x3,ind3) = outp
   where
        (x2, ind2) = vhead s
        (x1, ind1) = vlast s
        outp = compPattern (x2,ind2) (x2>x3) (x2>x1)

compPattern c True True = c
compPattern c _ _ = (0,0)

-- comp2
comp2 s inp = if fst inp > fst s then inp
              else s

-- pi/6 index:84
evals = $(v ([11,1,1,1]::[Int]))
ev1 = $(v ([49,-50,52,49]::[Int]))
ev2 = $(v ([47,80,37,-5]::[Int]))
ev3 = $(v ([56,-32,-12,-76]::[Int]))
ev4 = $(v ([-48,-10,76,-43]::[Int]))

-- pi/3 index:171
--evals = $(v ([11,1,1,1]::[Int]))
--ev1 = $(v ([-16,40,-70,57]::[Int]))
--ev2 = $(v ([28,69,59,31]::[Int]))
--ev3 = $(v ([-10,-57,32,75]::[Int]))
--ev4 = $(v ([94,-20,-26,8]::[Int]))
```

```

-- pi/4 index:127
--evals = $(v ([11,1,1,1]::[Int]))
--ev1 = $(v ([-32,70,-63,11]::[Int]))
--ev2 = $(v ([-76,18,61,13]::[Int]))
--ev3 = $(v ([55,54,42,48]::[Int]))
--ev4 = $(v ([-14,-42,-24,86]::[Int]))

stimuli = (evals,ev1:>ev2:>ev3:>ev4:>Nil)

test = simulate (sps.unpack) (replicate 256 stimuli) ::[SPSO]
```

```
{-# LANGUAGE GADTs, ScopedTypeVariables, TemplateHaskell,
DataKinds #-}
module EigSort (eigsort) where

import CLaSH.Prelude

type SortI = Vec 4 (Signed 16)
type Vinit = Vec 4 (Signed 16, Unsigned 8)
type SortO = Unsigned 8

vInit :: Vinit
vInit = vcopyI (0,0)

eigsort :: SortI -> SortO
eigsort evals = ind
   where
         inds = $(v ([0..3]::[Int]))
         ys = vzip evals inds
         vs' = vfoldl sort vInit ys
         ind = snd $ vhead vs'

sort vsi y = vsi'
   where
         (y', vsi') = vfoldl cswap (y,vInit) vsi

cswap (a,xs) b = if fst a > fst b then (b, xs <<+ a)
                             else (a, xs <<+ b)

stimuli = $(v ([6,19,10,4]::[Int]))
test = eigsort stimuli
```

```
{-# LANGUAGE TemplateHaskell #-}
module SvLUT (svlut,mv) where

import CLaSH.Prelude

as = [(2*n-1)*pi/2 | n <- [1,2]]
```

```
bs = [n/512*pi | n <- [0..255]]

svlut = [[round $ (128*) $ cos $ (a*) $ sin b | a <- as]++[round $ (128*) $ sin $ (a*) $ sin b | a <- as] | b <- bs]

mv []     = [| Nil |]
mv (r:rs) = [| $(v r) :> $(mv rs) |]
```

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module Norm (norm) where

import CLaSH.Prelude

type NormI = Vec 4 (Signed 16)
type NormO = Signed 16

norm :: NormI -> NormI -> NormO
norm xs ys = outp
   where
        dp = shiftR (dotp xs ys) 7
        outp = dp*dp

-- dot product
dotp xs ys = vfoldl (+) 0 ws
   where
        ws = vzipWith (*) xs ys
```

Appendix D: CλaSH & VHDL code of pipelined CORDIC_A

CλaSH code:

```
{-# LANGUAGE ScopedTypeVariables, TemplateHaskell, DataKinds #-}
module CordicA (topEntity) where

import CLaSH.Prelude

type CordicS = Vec 10 (Signed 16, Signed 16, Vec 10 Bit)
type CordicI = (Signed 16, Signed 16)
type CordicO = (Signed 16, Signed 16, Vec 10 Bit)

topEntity = cordic_a

cordic_a inp = outp
```

```
  where
        outp = (cordicA_core <^> sInit) inp

-- initial state
dsinit :: Vec 10 Bit
dsinit = vcopyI H
sInit :: CordicS
sInit = vcopyI (0,0,dsinit)

-- core function
cordicA_core :: CordicS -> CordicI -> (CordicS, CordicO)
cordicA_core s (ri,ui) = (s',outp)
  where
        ids = $(v ([1..10]::[Int]))
        pipeIns = vzip ids ((ri,ui,dsinit) +>> s)
        s' = vmap ca pipeIns
        outp = vlast s

 -- pipeline component
ca (pipeId,(ri,ui,dsi)) = (ro,uo,dso)
  where
        p1 = shiftR ri ((pipeId-1)*2)
        q1 = ri - p1
        q2 = shiftR ui pipeId
        q3 = shiftR ri pipeId
        p2 = shiftR ui ((pipeId-1)*2)
        q4 = ui - p2
        d = getSign ri ui
        dso = d +>> dsi
        ro = addSub d q1 (shiftL q2 2)
        uo = addSub (complement d) q4 (shiftL q3 2)

-- rotate according to the direction : H/L
addSub L a b = a + b
addSub H a b = a - b

getSign x y = if vhead (toBV x) == vhead (toBV y) then H
                else L
```

VHDL code:

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY CordicA IS
  PORT(rst : IN STD_LOGIC;
         clk : IN STD_LOGIC;
         ri  : IN SIGNED (15 DOWNTO 0);
         ui  : IN SIGNED (15 DOWNTO 0);
         ro  : OUT SIGNED (15 DOWNTO 0);
```

```
         uo  : OUT SIGNED (15 DOWNTO 0);
         ds  : OUT STD_LOGIC_VECTOR (9 DOWNTO 0));
END CordicA;

ARCHITECTURE struct OF CordicA IS

TYPE inds IS ARRAY (0 TO 9) OF INTEGER RANGE 1 TO 10;
TYPE ris IS ARRAY (0 TO 10) OF SIGNED (15 DOWNTO 0);
TYPE uis IS ARRAY (0 TO 10) OF SIGNED (15 DOWNTO 0);

CONSTANT inds1 : inds:= (1,2,3,4,5,6,7,8,9,10);
SIGNAL ris1  : ris;
SIGNAL uis1  : uis;

COMPONENT ca
   PORT ( rst : IN STD_LOGIC;
           clk : IN STD_LOGIC;
           ind : IN INTEGER RANGE 1 TO 10;
           ri  : IN SIGNED (15 DOWNTO 0);
           ui  : IN SIGNED (15 DOWNTO 0);
           ro  : OUT SIGNED (15 DOWNTO 0);
           uo  : OUT SIGNED (15 DOWNTO 0);
           d   : OUT STD_LOGIC
        );
END COMPONENT;

BEGIN

 cordics : FOR i IN 0 TO 9 GENERATE
          cordica_x : ca
          PORT MAP (rst,
                     clk,
                     inds1(i),
                     ris1(i),
                     uis1(i),
                     ris1(i+1),
                     uis1(i+1),
                     ds(i)
                     );
 END GENERATE;



 process (rst,clk)
 BEGIN
      if (rst = '1') then
         ro <= (others => '0');
         uo <= (others => '0');
         ris1(0) <= (others => '0');
         uis1(0) <= (others => '0');
      elsif (rising_edge(clk)) then
```

```
          ris1(0) <= ri;
          uis1(0) <= ui;
          ro <= ris1(10);
          uo <= uis1(10);
      end if;
end process;

END struct;
```

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY ca IS
   PORT(  rst : IN STD_LOGIC;
            clk : IN STD_LOGIC;
            ind : IN INTEGER RANGE 1 TO 10;
            ri  : IN SIGNED (15 DOWNTO 0);
            ui  : IN SIGNED (15 DOWNTO 0);
            ro  : OUT SIGNED (15 DOWNTO 0);
            uo  : OUT SIGNED (15 DOWNTO 0);
            d   : OUT STD_LOGIC
        );
END ca;

ARCHITECTURE behavioral OF ca IS

SIGNAL p1 : SIGNED (15 DOWNTO 0);
SIGNAL q1 : SIGNED (15 DOWNTO 0);
SIGNAL q2 : SIGNED (15 DOWNTO 0);
SIGNAL q3 : SIGNED (15 DOWNTO 0);
SIGNAL p2 : SIGNED (15 DOWNTO 0);
SIGNAL q4 : SIGNED (15 DOWNTO 0);
SIGNAL ro_tmp : SIGNED (15 DOWNTO 0);
SIGNAL uo_tmp : SIGNED (15 DOWNTO 0);

BEGIN

 compute : PROCESS (ri,ui,ind,p2,p1,q1,q2,q3,q4)
  BEGIN
        p1 <= shift_right(ri,2*(ind-1));
        q1 <= ri - p1;
        q2 <= shift_right(ui,ind);
        q3 <= shift_right(ri,ind);
        p2 <= shift_right(ui,2*(ind-1));
        q4 <= ui - p2;
        if ri(15) = ui(15) then
             ro_tmp <= q1 - (shift_left(q2,2));
             uo_tmp <= q4 + (shift_left(q3,2));
            d  <= '1';
        else
```

```
             ro_tmp <= q1 + (shift_left(q2,2));
            uo_tmp <= q4 - (shift_left(q3,2));
            d <= '0';
        end if;

 END PROCESS;

 update : PROCESS (clk,rst)
  BEGIN
        if (rst = '1') then
            ro <= (others => '0');
            uo <= (others => '0');
        elsif (rising_edge(clk)) then
            ro <= ro_tmp;
            uo <= uo_tmp;
        end if;

  END PROCESS;

END behavioral;
```