Refinement in ForSyDe

Master of Science Thesis
in Electronics System Design

by

Hua Tian

Stockholm, December 2002

Supervisor: Zhonghai Lu/Ingo Sander
Examinator: Axel Jantsch
Acknowledgments

I am greatly indebted to Prof. Axel Jantsch, for his vision and motivation in providing this Mater Thesis Project on System on Chip (SoC) Design, which helps me start an excellent career in this area.

My sincere gratitude gives to Mr. Lu zhonghai, my supervisor for his supervision and support. I am also especially thankful to Ingo Sander for his help and excellent suggestions throughout my thesis work.

At last, I am very grateful to my parents, sister and my boy friend for their support and love all these years.
Abstract

The ForSyDe (Formal System Design) methodology has been developed for system level design. It deals with the specification and refinement of a set of communication processes into embedded or single chip systems consisting of both hardware and software. The functional language Haskell is taken as the modelling language. Within the functional domain and high abstract level, a system specification is gradually refined into an implementation model that can be directly mapped onto VHDL and C descriptions and synthesized by traditional synthesis and compiler tools. The purpose of refinement is to achieve an efficient implementation. Based on previous work, this thesis discusses some design refinements in ForSyDe regarding data type transformations, clock domain refinement and resource sharing. And we develop some Haskell libraries to realize them. Moreover, we conduct a case study on parts of a given digital equalizer system model by using the developed libraries.

Keywords: Design Refinement, Design Transformation, Design Methodology, System Design.
# Contents

1 **An Introduction to ForSyDe** .................................................. 6  
1.1 Background ........................................................................ 6  
1.2 The System Model ................................................................. 7  
1.3 Modelling Language ............................................................. 9  
1.4 Process Constructors ............................................................ 10  
1.5 Refinement Techniques ......................................................... 12  
1.6 The Digital Equalizer Model .................................................. 13  
1.7 Project Objective ................................................................. 19  

2 **Data Type Transformation** .................................................... 21  
2.1 Introduction ........................................................................ 21  
2.2 Data Representation ............................................................ 21  
  2.2.1 Floating-point Representation ........................................... 21  
  2.2.2 Fixed-Point Representation .............................................. 25  
  2.2.3 Dynamic Range and Precision ........................................... 28  
  2.2.4 Overflows and Scaling .................................................... 29  
2.3 Comparisons between Floating-point and Fixed-point Numbers .... 29  
2.4 Transformation from Floating-point to Fixed-point .................. 30  
  2.4.1 Minimal Word Length Determination ................................. 30  
  2.4.2 Transformation Steps with Determined Word Length .......... 34  
  2.4.3 Error Discussion ........................................................... 35  
2.5 Fixed-point Arithmetic ......................................................... 36  
  2.5.1 Addition/Subtraction .................................................... 36  
  2.5.2 Multiplication ............................................................. 37  
  2.5.3 Division ...................................................................... 38  
  2.5.4 Data Type Transformations in ForSyDe ............................. 39  
  2.5.5 Fixed-point Arithmetic in FIR ....................................... 42  
  2.5.6 The Refined FIR Model after Data Type Transformation .... 47  

3 **Refinement in Clock Domain and Resource Sharing** .................. 49  
3.1 Refinement in Clock Domain .................................................. 49  
  3.1.1 The Multi-clock in ForSyDe ............................................. 49  
  3.1.2 Clock Domain Refinement in a Digital Equalizer ............... 51  
3.2 Resource Sharing .................................................................. 54  
  3.2.1 Domain Interfaces in the Refinement of Resource Sharing ... 54  
  3.2.2 Resource Sharing in a Digital Equalizer ........................... 59
4 Summary
4.1 Conclusion ....................................................... 62
4.2 Future Work ....................................................... 62
5 Appendix
  5.1 The Fixed Point Library ........................................ 64
  5.2 The Clock Domain Library ..................................... 68
  5.3 The Resource Sharing Library ................................. 70
  5.4 Case Study Code .................................................. 72
    5.4.1 Original Code ............................................... 72
    5.4.2 Refinement Code ............................................. 74
    5.4.3 Test Code .................................................... 75
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Specification Model of a Network of Concurrent Processes</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>A Synchronous Process in the Specification Model</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Process Construction with <code>mapSY</code></td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Process Construction with <code>zipwithSY</code></td>
<td>11</td>
</tr>
<tr>
<td>1.5</td>
<td>Process Construction with <code>scanSY</code></td>
<td>12</td>
</tr>
<tr>
<td>1.6</td>
<td>Process Construction with <code>mooreSY</code></td>
<td>12</td>
</tr>
<tr>
<td>1.7</td>
<td>Process Construction with <code>mealySY</code></td>
<td>13</td>
</tr>
<tr>
<td>1.8</td>
<td>Refinement by Process Merging and Splitting</td>
<td>13</td>
</tr>
<tr>
<td>1.9</td>
<td>The Equalizer and its Environment</td>
<td>14</td>
</tr>
<tr>
<td>1.10</td>
<td>Subsystems of the Equalizer</td>
<td>14</td>
</tr>
<tr>
<td>1.11</td>
<td>Subsystems of the Audio Filter</td>
<td>15</td>
</tr>
<tr>
<td>1.12</td>
<td>FIR-Filter</td>
<td>16</td>
</tr>
<tr>
<td>1.13</td>
<td>The Specification Model of the FIR-Filter</td>
<td>17</td>
</tr>
<tr>
<td>1.14</td>
<td>The Audio Analyzer</td>
<td>18</td>
</tr>
<tr>
<td>2.1</td>
<td>The Format of the Unsigned Fixed-point Number</td>
<td>25</td>
</tr>
<tr>
<td>2.2</td>
<td>The Circle Representation Method for an Unsigned Fixed-point Number with Integral Word Length $a = 1$</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>The Format of the Unsigned Fixed-point Number</td>
<td>27</td>
</tr>
<tr>
<td>2.4</td>
<td>The Circle Representation Method for a Signed Two's Complement Fixed-point Number with Integral Word Length $a = 1$</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>Dataflow Graph for the Function $F$</td>
<td>31</td>
</tr>
<tr>
<td>2.6</td>
<td>Derivation of Error Statistics</td>
<td>33</td>
</tr>
<tr>
<td>2.7</td>
<td>The Examples of Transformation Procedure from Floating-point to Fixed-point</td>
<td>35</td>
</tr>
<tr>
<td>2.8</td>
<td>Signed Fixed-point Addition/Subtraction with Defined Word Length</td>
<td>37</td>
</tr>
<tr>
<td>2.9</td>
<td>Addition of Signed Two's Complement Fixed-point Numbers</td>
<td>38</td>
</tr>
<tr>
<td>2.10</td>
<td>Signed Fixed-point Multiplication with Defined Word Length</td>
<td>38</td>
</tr>
<tr>
<td>2.11</td>
<td>Multiplication of Signed Two's Complement Fixed-point Numbers</td>
<td>39</td>
</tr>
<tr>
<td>2.12</td>
<td>Data Type Representation of Signed Two's Complement Fixed-point Number</td>
<td>40</td>
</tr>
<tr>
<td>2.13</td>
<td>Data Type Transformation between Two Processes</td>
<td>40</td>
</tr>
<tr>
<td>2.14</td>
<td>Data Type Refinement Models for Signals and Vectors</td>
<td>41</td>
</tr>
<tr>
<td>2.15</td>
<td>FIR Arithmetic in the 1st Cycle</td>
<td>43</td>
</tr>
<tr>
<td>2.16</td>
<td>FIR Arithmetic in the 2nd Cycle</td>
<td>44</td>
</tr>
<tr>
<td>2.17</td>
<td>FIR Arithmetic in the 3rd Cycle</td>
<td>46</td>
</tr>
<tr>
<td>2.18</td>
<td>The Data Type Refinement Model of the FIR</td>
<td>47</td>
</tr>
</tbody>
</table>
3.1 Domain Interfaces for Sampling .......................... 50
3.2 An Example for Domain Interfaces .................. 51
3.3 Direct Implementation of the Audio Analyzer .......... 52
3.4 Implementation of Clock Domain Refinement Using Identity .... 54
3.5 The Audio Analyzer after Refinement .................. 55
3.6 Domain Interfaces for Parallel/Serial Conversion .......... 55
3.7 A Transformation for Resource Sharing ............... 57
3.8 A Refined Adder Sharing Model using Parallel-to-serial and Serial-to-parallel Domain Interfaces ............... 58
3.9 A Resource Sharing Solution using Parallel-to-Serial and Down-sampling Domain Interfaces .................. 58
3.10 A Multiplier Sharing Model using Parallel-to-serial and Down-sampling domain Interfaces .................. 59
3.11 Direct Implementation of the FIR-Filter Specification Model ........ 60
3.12 Resource Shared FIR by Vector Rotation ............... 61

4.1 The Refinements on the Digital Equalizer ............... 63
Chapter 1

An Introduction to ForSyDe

1.1 Background

Today more and more ICs are System-on-Chip designs - integrating processors, DSPs, memories, I/O, dedicated logic, and even embedded software. This architecture allows for more new application area, but how to design such applications is not so obviously.

Keutzer et. al discuss the system-level design in [1]. They point out that "to be an effective design methodology that addresses complex systems, it must start at high levels of abstraction" and underline that "an essential component of a new system design paradigm is the orthogonalization of concerns", i.e. the separation of various aspects of design to allow more effective exploration of alternative solutions. In particular, a design methodology should separate (1) function (what the system is supposed to do) from architecture (how it does it) (2) communication from computation.

They promote to use formal models and transformations in system design so that verification and synthesis can be applied to the advantage of the design methodology and believe that the most important point for functional specification is the underlying mathematical model of computation.

These arguments strongly support the ForSyDe (Formal System Design) methodology [3] [4] [5] [6] [7] [8] as many of their main requirements on a system design methodology are not only the part of the methodology, but establish the foundations of ForSyDe.

The ForSyDe methodology addresses the design of SoC applications. It is a system level design methodology which starts with a formal specification model that captures the functionality of the system. It provides refinement methods inside the functional domain to transform the abstract specification into an efficient implementation model which can then be synthesized into hardware, software or a mixed hardware/software design.
Formal approaches to system design have been discussed [2] but not been generally adopted. They are frequently on a too high abstraction level to allow for efficient synthesis with today’s techniques, and also a large gap from an abstract specification to an implementation. In paper [3] Ingo and Axel proposed a formal system design approach based on the synchrony hypothesis, functional models and skeletons. A system is modelled using the \textit{perfect synchrony hypothesis}, and the functional language Haskell. In particular, the use of skeletons in conjunction with a proper computational model allows a system model to have an interpretation in hardware and software. The functional language was chosen as the modelling language because it is based on formal semantics and purely functional, supports higher-order functions, has a lazy evaluation mechanism and so on. It is an executable specification which allows verifying the system functionality by simply running programs. In ForSyDe, a system application is expressed as concurrent communicating processes. A skeleton is used to construct a process. It can be viewed as a higher order function in the model. A skeleton is later also called \textit{process constructor} which directly assigns its function to its name. A skeleton allows separating communication from computation. Paper [4] furthers these discussions by proposing a hardware synthesis method. In addition, it describes a design methodology which uses the modelling concepts and the synthesis method. It contains a design exploration phase and defines how and when design decisions are formally introduced into the synthesis process by synthesizing a FIFO component taken from an ATM switch. Paper [5] continues the system synthesis discussions. It put forward a layered synthesis corresponding to the layered functional model. The layers inside a functional model are: the system layer, the skeleton layer and the elementary layer. Based on the work described above, paper [8] extends the hardware synthesis into both hardware and software synthesis, and conducted a case study on a digital equalizer model [9] resulting three implementations, a pure hardware implementation in VHDL, a pure software implementation in C, and a mixed hardware and software implementation. The refinement methodology in ForSyDe is presented in papers [6] and [7]. Paper [6] describes a systematic refinement methodology based on \textit{transformations}, which gradually transforms a high-level function oriented system description into a synthesizable model. In paper [7], two classes of design transformations are defined: (1) semantic-preserving transformations and (2) design decisions. In particular, it presents and illustrates communication and clock domain refinement by the digital equalizer.

\subsection{1.2 The System Model}

Two kinds of system model are used in ForSyDe: specification model and implementation model.

The \textit{specification model} is a network of concurrent synchronous processes shown in Figure 1.1. The processes communicate with each other synchronously by means of signals. All signals are triggered by a global clock, which makes them have the same data rate. Normally the data rate equals to 1. The signal is defined as a set
of events. Every event has both value and tag. There is a strict ordering of events in the synchronous model. We may denote the sequence of events in the signal by the tags.

\[ (s_2, s_3) = P_1(s_1) \]
\[ s_5 = P_2(s_2, s_4) \]
\[ s_4 = P_3(s_3) \]

The specification model is based on perfectly synchronous hypothesis. The basic synchronous assumption is that the outputs of a system are synchronized with the system input. This means the reaction of the system takes no observable time. Simply, neither computation nor communication takes time. Every event cycle, the process takes one event with tag \( n \) as input, after function computation, the output is emitted at the same tag \( n \) (Figure 1.2). A special symbol "\( \bot \)" is introduced to model the absence of a value. Absent values can be used to model signals with different data rate. For any kind of functional operations with the input \( \bot \), the output must also be an absent value. A synchronous process or S-process is a functional mapping of \( n \) input signals with signal rate \( r \) into \( n \) output signals with the same signal rate \( r \). Following the signal definition we define a data type Signal \( a \), that models signals as lists of events, where the tag corresponds to the position in the list.

```haskell
data Signal a = NullS
    |  a: ~(Signal a)
```

A signal with values of a data type \( a \) is either empty(NullS) or recursively composed by a value of type \( a \) and a signal of type \( a \), e.g.

1::2::3:: NullS

is a signal with a set of integer events separated with ':-'.

The signal of timed events can be donoted as

Prist 1::Prist 2::Prist 3:: Abst::NullS

"Prist" and "Abst" indicate the presence and absence of the event at every clock cycle in respective.
An *implementation model* is a refinement result of a specification model. The specification model has only one global clock and all the signals in the concurrent synchronous process network have the same data rate, while the implementation model may have multiple clock domains by establishing domain interfaces [7]. A Domain interface is a functional mapping of *n* input signals with signal rate *r* into *n* output signals with another signal rate *u* \( r \neq u, u \in Q \).

Different clock domains may have different data rates. In each local clock domain, the process network still has the synchronous property as the specification model. Other refinement methodology, such as data type transformation and resource sharing, can be also applied to make the original system model more efficient in hardware and software implementation.

1.3 Modelling Language

The *functional language* Haskell [10] [11] has been selected as modelling language. The functional program such as C, Java, Pascal, and so on are all *imperative* languages. They are ”imperative” in the sense that they consist of a sequence of commands, which are executed strictly one after the other. An *imperative* program deals with *how* something is to be done in a strict order. The focus on the high-level *what* rather than the low-level *how* is a distinguishing characteristic of functional programming languages.

Haskell is defined by a formal semantics and purely functional. It has the following important characteristics which are suitable for system modeling at a high abstract level.

- Haskell is based on the *lambda calculus*. It is *purely functional*. A Haskell program is a list of equation which has to be satisfied with. And it has
no predefined execution order. This feature makes us concentrate on the functionality of a system.

- Haskell supports higher order functions. In Haskell a functions is a first-class citation: It can freely be passed to other functions, returned as the result of a function, stored in a data structure, and so on. It turns out that the judicious use of higher-order functions can substantially improve the structure and modularity of many programs. We take advantage of this feature to build skeletons.

- Haskell is a non-strict functional language. That means it has another powerful feature: It only evaluates as much of the program as is required to get the answer. This is called lazy evaluation. It allows us to write more modular programs. This is used to module signals of possibly infinite length and data structures of possibly infinite size.

- Haskell is strongly typed, eliminating a huge class of easy-to-make error at compile time.

- Haskell provides the language facilities needed to model both data flow and control flow applications. A functional language naturally fits for modeling data-flow application. In Haskell a variety of the control constructors such as pattern matching enable us to model control-oriented applications.

In addition to those mentioned above, there are other persuasive reasons to use Haskell. Much of a software product’s life is spent in specification, design and maintenance, and not in programming. Functional languages are superb for writing specifications which can actually be executed. Functional languages are also relatively easy to maintain, because the code is shorter, clear, and the rigorous control of side effects eliminating a huge class of unforeseen interactions.

1.4 Process Constructors

Process constructors [3] are implemented with higher-order functions. There are two kinds of process constructors: combinational and sequential constructors.

The combinational process constructors have no internal state and the events of input signal(s) are one-by-one mapped into output signals by a certain combinational function.

The basic combinational process constructors include \( \text{mapSY} \) and \( \text{zipwithSY} \).

\[
\begin{align*}
\text{mapSY} & : (a \rightarrow b) \rightarrow \text{Signal} a \rightarrow \text{Signal} b \\
\text{mapSY} f \text{ Null}S & = \text{Null}S \\
\text{mapSY} f \text{ Null}S & = f \text{ Null}S \\
\text{mapSY} f \text{ Null}S & = f x : \text{mapSY} f \text{ xs}
\end{align*}
\]

The higher-order function \( \text{mapSY} \) (Figure 1.3) has two arguments. The first argument is a function \( f \) and the second one is a signal of any type. The definition of \( \text{mapSY} \) uses pattern matching. The first pattern matches, if the signal is empty (NullS). The second pattern matches, if the signal has at least one value, i.e. it
is constructed by a head value \( x \) and a signal tail \( \text{xs} \) which represents the whole signal list except the head value \( x \). In this case \( f \) is applied to \( x \) and the result of this function will be the first value of the output signal. The rest of the output signal is calculated recursively by the function \( \text{mapSY} f \text{xs} \).

\[
\text{mapSY}(f) \quad s = \{v_0, v_1, \ldots\} \xrightarrow{f} s' = \{f(v_0), f(v_1), \ldots\}
\]

Figure 1.3: Process Construction with \( \text{mapSY} \)

Let’s illustrate \( \text{mapSY} \) with the function \( f = 2x \) with an integer signal.
\[
\text{mapSY} (*2) \quad (1:2:3:\text{-}\text{NullS}) \Rightarrow (2:4:6:\text{-}\text{NullS})
\]

The process constructor \( \text{zipWithSY} \) (Figure 1.4) applies a ‘two-operand-function’ on elementwise on two input signals.

\[
\text{zipWithSY} :: (a \to b \to c) \to \text{Signal a} \to \text{Signal b} \to \text{Signal c}
\]

\[
\text{zipWithSY \_ \_ \_ NullS \_ \_ NullS = NullS}
\]

\[
\text{zipWithSY f (x:xs) (y:ys) = f x y \_ \_ (zipWithSY f xs)}
\]

\[
\text{zipWithSY}(f) \quad s_1 = \{x_0, x_1, \ldots\} \xrightarrow{f} s' = \{f(x_0, y_0), f(x_1, y_1), \ldots\}
\]

\[
\text{zipWithSY}(f) \quad s_2 = \{y_0, y_1, \ldots\}
\]

Figure 1.4: Process Construction with \( \text{zipWithSY} \)

In the following example, we define the function \( f = x + y \),
\[
\text{zipWithSY (+)} \quad (1:2:3:\text{-}\text{NullS}) \Rightarrow (2:4:6:\text{-}\text{NullS})
\]

In contrast to combinational process constructors, the sequential process constructors have a internal state as an argument. \( \text{scandSY} \) (Figure 1.5) is to describe simple FSM state machine. The process constructor has input arguments: state function, initial state, and input signal. Output is type of Signal. Current output value will be next state value of automata.

\[
\text{scandSY} :: (a \to b \to a) \to \text{Signal a} \to \text{Signal b} \to \text{Signal a}
\]

\[
\text{scandSY \_ \_ \_ NullS \_ \_ NullS = NullS}
\]

\[
\text{scandSY f mem (x:xs) = mem \_ \_ (scandSY f newmem xs)}
\]

where \( \text{newmem} = f \text{ mem} x \)

The following is an example for \( \text{scandSY} \) process constructor. Let’s describe state function as
\[ s = \{v_0, v_1, \ldots\} \]

\[ f \rightarrow m_0 \]

\[ s' = \{v'_0, v'_1, \ldots\} \]

\[ v'_0 = m_0 \]

\[ v'_i = m_i (i > 0) \]

\[ m_{i+1} = f(v_i, m_i) \]

Figure 1.5: Process Construction with \textit{scandSY}

\[
f \times y | x < 5 = x + 1 \quad \text{otherwise} = 0
\]

\textit{scandSY} \( f \ 0 \ (0: -1; -3: -5; \text{NullS} \Rightarrow (0: -1; -4: -0: \text{NullS})

For more complex process constructors, we can model them by the basic constructors \textit{mapSY}, \textit{zipWithSY} and \textit{scandSY}. For example, \textit{mooreSY} can be viewed as a combination of \textit{mapSY} and \textit{scandSY}. It is used to model a \textit{moore FSM}.

The \textit{mooreSY} (Figure 1.6) has five arguments, the first one is state function, the second is output function, the third is initial state, the fourth is input Signal and the last is output Signal.

\[
\textit{mooreSY} :: (a \to b \to c) \to (a \to c) \to a \to \text{Signal} \ b \to \text{Signal} \ c
\]

\textit{mooreSY} nextState output initial = \textit{mapSY} output . (\textit{scandSY} nextState initial)

\[ s = \{v_0, v_1, \ldots\} \]

\[ f \rightarrow m_0 \]

\[ s'' = \{v''_0, v''_1, \ldots\} \]

\[ v''_0 = m_0 \]

\[ v''_i = m_i (i > 0) \]

\[ m_{i+1} = f(v_i, m_i) \]

Figure 1.6: Process Construction with \textit{mooreSY}

The \textit{mealySY} (Figure 1.7) is modelled by a combination of \textit{zipWithSY} and \textit{scandSY}. The difference between \textit{mealySY} and \textit{mooreSY} is that the output of \textit{mealySY} depends on both state and input signal.

\[
\textit{mealySY} :: (a \to b \to c) \to (a \to b \to c) \to a \to \text{Signal} \ b \to \text{Signal} \ c
\]

\textit{mealySY} nextState output initial signal = \textit{zipWithSY} output . (\textit{scandSY} nextState initial signal)

\[ \text{1.5 Refinement Techniques} \]

There are two classes of transformation techniques [7]:

- \textit{Semantic Preserving Transformations} do not change the characteristic of a model or process network. The refined model and original model are semantically equivalent. Semantic preserving transformations are mainly used to
optimize the model for synthesis. For example, we can merge two processes into a signal optimized process by the approach (Figure 1.8)

\[ \text{mapSY}(f) \circ \text{mapSY}(g) = \text{mapSY}(f \circ g). \]

This semantic preserving transformation is based on (1) Processes can be easily be moved over block borders and (2) two processes can be combined and possibly optimized by use of function composition.

- **Design Decisions** change the semantics of a model or process network, but the refined model may still behave in the same way as the original model. A typical design is the refinement of an infinite buffer into a fixed-size buffer with \( n \) elements. While such a design decision clearly modifies the semantics, the transformed model has the same behavior as the original model. For instance, if it is possible to prove that a certain buffer will never contain more than \( n \) elements, the ideal buffer can be replaced by a finite one of size \( n \).

### 1.6 The Digital Equalizer Model

A digital equalizer has been modelled in terms of ForSyDe methodology [3] [8] [9]. The Figure 1.9 shows the digital equalizer and its external environment. In this
case, an audio input audio stream is modified and converted into an output audio stream by the controlled bottoms.

![Diagram of equalizer and its environment]

Figure 1.9: The Equalizer and its Environment

The equalizer system adjusts the audio signal according to the control signal (Bass up, Bass down, Treble up and Treble Down) to prevent it exceeding a predefined threshold which may result in the damage to the speakers. This specification model consists of four subsystems (Audio Filter, Audio Analyzer, Button Control and Distortion Control). The internal structure of the digital equalizer is shown in Figure 1.10.

![Diagram of equalizer subsystems]

Figure 1.10: Subsystems of the Equalizer

equalizer buttons audioIn = audioOut where
levels = buttonControl buttons override
audioOut = audioFilter levels audioIn
disFlag = audioAnalyzer audioOut
overrides = distortionControl (Abst :- disFlag)

In this digital equalizer:

- The *Audio Filter* receives the control signal from the subsystem *Button Control*, then filters and amplifies the input audio signal with corresponding levels.
The **Audio Analyzer** analyzers with the output audio signals from the **Audio Filter** and determines if the bass exceeds a predefined threshold.

The **Button Control** monitors the button inputs and the override signal from the subsystem **Distortion Control**, in turning passing the current amplification level to **Audio Filter**.

The **Distortion Control** decides if a minor or major violation is encountered and issues the necessary commands to the **Button Control** subsystem.

The subsystem **Button Control** and **Distortion Control** are control dominated, while the **Audio Filter** and **Audio Analyzer** are data flow dominated subsystems. In this thesis, since the applied refinement methodology in the following section is focus on the subsystems of **Audio Filter** and **Audio Analyzer** in the digital equalizer, we will illustrate them with Haskell description.

Figure 1.11 shows the structure of the **Audio Filter**. The task of this subsystem is to amplify different frequencies of the audio signal independently according to the control levels decided by **Button Control**. The audio signal is split into three identical signals and fed into different filters: Low Pass Filter (LPF), Band Pass Filter (BPF) and High Pass Filter (HPF). A parametric Finite Impulse Response (FIR) is used to realize the three filters. According the control levels, the signals are filtered and amplified. As the equalizer in this design only has a bass and treble control, the middle frequencies are not amplified. The output signal from the **Audio Filter** is the addition of the three filtered and amplified signals.

![Diagram of Audio Filter Subsystems](image)

**Figure 1.11: Subsystems of the Audio Filter**

```haskell
audioFilter levels audioIn = audioOut where
audioOut = sumSignals lowPath bandPath highPath
lowPath = (amplify bass . lowPass) audioIn
bandPath = bandPass audioIn
highPath = (amplify treble . highPass) audioIn
```
(bass, treble) = levels

Here lp, bp and hp are the parametric n-order FIR LPF, BPF and HPF respectively. The following example is a 8-order FIR Filter where FIR is modelled by another Haskell library \textit{fir.lhs}.

\begin{verbatim}
bp = fir (vector [
    0.06318761339784, 0.08131651217682,
    0.09562326700432, 0.10478344432968,
    0.10793629404886, 0.10478344432968,
    0.09562326700432, 0.08131651217682,
    0.06318761339784])
\end{verbatim}

An FIR-Filter is described by the equation

\[
y_n = \sum_{m=0}^{k} x_{n-m} h_m
\]

and can be visualized as Figure 1.12.

![FIR-Filter Diagram](image)

\textbf{Figure 1.12: FIR-Filter}

The FIR-Filter is constructed with two processes \textit{sipoSY} and \textit{mapSY ipVh} (Figure ??). The state of the FIR-Filter is modelled by the process constructor \textit{scandSY} as a state machine. The process \textit{sipoSY} takes \textit{shiftrV} as the function and the initial state is a vector with \(k+1\) zeroes. Every cycle, a value \(x_{n+1}\) is shifted into the vector from right, all other elements are shifted one place to left, and the previous head value \(x_{n-k}\) leaves the vector. The number of elements in the vector keeps \(k+1\). With the state forwarded by the process \textit{sipoSY}, the process \textit{mapSY ipVh} calculates the inner product with two sub-processes. The multiplication process \textit{mapSY mulV(h)} multiplies the state and the coefficients vector \((h_0, ..., h_k)\) of size \(k+1\) and send the product to accumulation process which implements the addition between the new input value and the accumulated value of previous cycle.

\begin{verbatim}
shiftrV :: Vector a -> Vector a
shiftrV vs v = tailV vs <+: v
\end{verbatim}
–shiftV shifts a value from the right into a vector.

The FIR Filter process can be described in Haskell as following.

```
firm s = (mapSY (ipV h)) . sipoSY k s
      where k = lengthV h

siposY m s = scadsY shiftV (copyV m 0.0) s

ipV h = (accV . mulV h) v

accV NullV = 0
accV (x :- xv) = x + accV xv
mulV NullV NullV = NullV
mulV (x:-xv) (y:-yv) = x * y :> (mulV xv yv)
```

Figure 1.14 shows the Audio Analyzer subsystem, which analyzes the audio output signal from Audio Filter and determines if the bass exceeds a predefined threshold. The procedure for analyzing the audio signal is:

- Group Sampling process reads n event of the input signal and groups them into a vector in terms of the points of the Fast Fourier Transform (FFT).
- FFT transforms time-domain discrete signals into frequency-domain signals to determine the frequency spectrum of a signal.
- Power Spectrum process computes the power spectrum of the signals from FFT.
- Check Low Freq. process compares the low frequency parts against a predefined threshold value. If the bass exceed the threshold value, a *Fail* event is generated, otherwise a *pass* event occurs.

The core of Audio Analyzer is FFT which realizes Discrete Fourier Transform (DFT). In ForSyDe system model, FFT carries out a complex-valued, radix-2, decimation-in-time algorithm with bit-reversed output.

In ForSyDe, the Audio Analyzer subsystem is expressed as below.

```plaintext
module AudioAnalyzer where

import FFT
import ForSyDeStdLib
import Complex
import EqualizerTypes

limit = 10.0

nLow = 4

audioAnalyzer :: Integer -> Signal Double -> Signal (AbstExt AnalyzerMsg)
audioAnalyzer fftLevel = mapSY (checkLimit limit) . mapSY sumVector .
set logV . mapSY (selectLow nLow) . mapSY squareV . mapSY absV .
mapSY selectHalf . mapSY (fftT k) . (distributor (2^k)).mapSY fromDouble
where k = fftLevel

fftT k Abst = Abst
fftT k (Prst vs) = Prst (fft k vs)

distributor n = groupSY n

selectHalf Abst = Abst
selectHalf (Prst (x:>xs)) = Prst (takeV ((lengthV xs - 1) / 2) xs)

absV Abst = Abst
absV (Prst vs) = Prst (mapV magnitude vs)

squareV :: AbstExt (Vector Double) -> AbstExt (Vector Double)
squareV Abst = Abst

squareV (Prst vs) = Prst (mapV sqr vs)
```

Figure 1.14: The Audio Analyzer
where \( \text{sqr } x = x \times x \)

\[
\begin{align*}
\text{selectLow } n \text{ Abst} &= \text{Abst} \\
\text{selectLow } n \ (\text{Prst } xs) &= \text{Prst} \ (\text{takeV } n \ xs)
\end{align*}
\]

\[
\begin{align*}
\logV \ \text{Abst} &= \text{Abst} \\
\logV \ (\text{Prst } vs) &= \text{Prst} \ (\text{mapV } \text{id } vs)
\end{align*}
\]

\[
\begin{align*}
\text{sumVector } \text{Abst} &= \text{Abst} \\
\text{sumVector } (\text{Prst } vs) &= \text{Prst} \ (\text{foldV } (+) \ 0.0 \ vs)
\end{align*}
\]

\[
\begin{align*}
\text{checkLimit } \text{limit } \text{Abst} &= \text{Abst} \\
\text{checkLimit } \text{limit } (\text{Prst } x) | x > \text{limit} &= \text{Prst } \text{Fail} \\
| x \leq \text{limit} &= \text{Prst } \text{Pass}
\end{align*}
\]

1.7 Project Objective

Today the basic concepts of the refinement methodology have been defined. However, some details have not been worked out. And some library functions have to be developed. Although a refinement in general covers a wide range of topics, this work mainly deal with: data type transformations, clock domain and resource sharing refinement. The data type transformation deals with the transformations from floating point to fixed point, and vice versa, also the some fixed point arithmetic. A fixed point operation results in a cheaper solution at the expense of mathematical precision. The clock domain and resource sharing have been documented in [7]. The refinement in clock domain results in the introduction of multiple clock domains which may more suit practical needs. The resource sharing aims at reducing chip area while lowering the performance. This work uses the theoretical results in the publications to develop a basic set of implementation libraries in Haskell. With the digital equalizer model, this thesis uses the developed libraries to conduct a case study.

Specifically, this thesis will achieve:

- Data type transformation: A transformation method on conversions between floating point data type and fixed point data type, and also some fixed point operation arithmetic. For use in ForSyDe, a fixed point library will be established.

- Refinement in clock domain and resource sharing: Based on the developed theoretical work, this thesis will develop some library functions in Haskell.

- A Case study with the two data flow subsystems of the Audio Filter and the Audio Analyzer in the digital equalizer. It is used to show the validity of the data type transformations, and the refinement in clock domain and resource sharing.

The rest of the thesis is organized as follows. Chapter 2 deals with floating point-to-fixed-point transformations and fixed-point arithmetic. Chapter 3
describes the refinement in clock domain and resource sharing. Chapter 4 summarizes the thesis with conclusion and future work. The last part is the appendix of the developed libraries including the fixed-point, clock domain, resource sharing library functions, and the code for the case study.
Chapter 2

Data Type Transformation

This chapter discusses data type transformations including conversions between floating point and fixed point numbers, and fixed point arithmetic. As a starting point, we first present floating point data type representations in IEEE format, and then compare them with fixed point representations, followed by some operation principles of fixed point numbers and discussions on the possible errors introduced by rounding and overflows. Based on this theoretical work, we develop a fixed point library for both data type transformation and fixed point operations. As a case study, we use the FIR filter to show the feasibility of the fixed point library.

2.1 Introduction

While most digital signal processing algorithms are developed using floating-point arithmetic, their implementation using very large scale implementation (VLSI) or fixed-point digital signal processors usually required fixed-point arithmetic for the sake of hardware cost and speed. However, fixed-point implementation can suffer from excessive finite word length effects, due to overflows and quantization noise, unless all signals are scaled properly and enough word length are assigned.

The transformation utilizes data to capture the dynamic range of floating-point variable and intermediate calculations to guide in generation of scaling operations to prevent overflows while maintaining the accuracy. An algorithm for generating shift operations results in a minimization error due to truncation or rounding.

2.2 Data Representation

2.2.1 Floating-point Representation

Floating point format has the remarkable property of automatically scaling all numbers by moving, and keeping track of, the radix point so that all numbers use the full word length available but never overflow.

In computer hardware, a floating-point number has four elements:

- The sign
• The significand
• The radix of the exponent
• The exponent

The exponent provides scaling, and the significand encodes the scalar data itself. The typical format of binary floating-point representation is shown in figure 2.2.1.

\[
\begin{array}{c|c|c}
\text{sign} & \text{exponent} & \text{significand} \\
\hline
p & q & \\
\end{array}
\]

Any binary floating-point number can be represented in floating-point using scientific notation form as \( x = (-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}} \), where 2 is the radix or base (binary in this case). Assume the word length of significand and the exponent are \( p \) and \( q \) respectively. The range of the floating-point number is \(- (2^p - 1) \times 2^{(2q-1)}\). The floating-point word length \( w = p + q + 1 \). (The extra bit is for the sign bit). In the example below, represents the fixed-point number \( x = (-1)^1 \times (2^4 + 2^3 + 2^1 + 2^{-1}) \times 2^{(2^3+2^1)} = -26.5 \times 1024 = -27136 \).

\[
\begin{array}{c|c|c}
1 & 1010 & 11010.100 \\
\end{array}
\]

The IEEE Format

The IEEE 754 Standard for binary floating-point arithmetic has been widely adopted for use on DSP processors.

This standard specifies exactly the single and double floating-point formats and it defines a class of extended format for each of these two basic formats: extended-precision and quadruple-precision.

**Single-Precision Format**

The IEEE 754 single precision floating-point format is a 32-bit word divided into:

• 1-bit sign indicator \( s \)
• 8-bit biased exponent \( E \)
• 23-bit fraction \( F \)

A representation of this format is given below.

\[
\begin{array}{c|c|c|c|c}
b_{31} & b_{30} & b_{22} & b_0 \\
\hline
s & E & F & \\
\end{array}
\]
<table>
<thead>
<tr>
<th>Number Characterization</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized, $0 &lt; E \leq 255$</td>
<td>$( -1 )^s \times ( 2^{E-127} ) \times ( 1, F )$</td>
</tr>
<tr>
<td>Denormalized, $E = 0; F \neq 0$</td>
<td>$( -1 )^s \times ( 2^{-126} ) \times ( 0, F )$</td>
</tr>
<tr>
<td>Zero, $E = 0, F = 0$</td>
<td>$( -1 )^s \times ( 0 )$</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Exceptional value</td>
</tr>
</tbody>
</table>

The relationship between this format and the representation of real numbers is given below.

**Double-Precision Format**

The IEEE 754 double precision (64-bit) floating-point format consists of:

- 1-bit sign indicator $s$
- 11-bit biased exponent $E$
- 52-bit fraction $F$

A representation of this format is given below.

\[
\begin{array}{cccc}
 b_{63} & b_{62} & b_{51} & b_0 \\
 s & E & F &
\end{array}
\]

The relationship between this format and the representation of real numbers is given below.

<table>
<thead>
<tr>
<th>Number Characterization</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized, $0 &lt; E \leq 2047$</td>
<td>$( -1 )^s \times ( 2^{E-1023} ) \times ( 1, F )$</td>
</tr>
<tr>
<td>Denormalized, $E = 0; F \neq 0$</td>
<td>$( -1 )^s \times ( 2^{-1022} ) \times ( 0, F )$</td>
</tr>
<tr>
<td>Zero, $E = 0, F = 0$</td>
<td>$( -1 )^s \times ( 0 )$</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Exceptional value</td>
</tr>
</tbody>
</table>

**Extended-Precision Format**

The IEEE 754 extended-precision (80-bit) floating-point format consists of:

- 1-bit sign indicator $s$
- 15-bit biased exponent $E$
- 1-bit explicit leading significand bit $j$
- 63-bit fraction $F$

A representation of this format is given below.

\[
\begin{array}{ccccccc}
 b_{79} & b_{78} & b_{63} & b_{62} & b_0 \\
 s & E & j & F &
\end{array}
\]

The relationship between this format and the representation of real numbers is given below.
<table>
<thead>
<tr>
<th>Number Characterization</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized, $j = 1,0 &lt; E \leq 2047$</td>
<td>$(-1)^s \times (2^{E-1023}) \times (1.F)$</td>
</tr>
<tr>
<td>Denormalized, $j = 0, E = 0; F \neq 0$</td>
<td>$(-1)^s \times (2^{-1022}) \times (0.F)$</td>
</tr>
<tr>
<td>Pseudo-Denormalized, $j = 1, E = 0$;</td>
<td>$(-1)^s \times (2^{-1022}) \times (1.F)$</td>
</tr>
<tr>
<td>Zero, $j = 0, E = 0, F = 0$</td>
<td>$(-1)^s \times (0)$</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Exceptional value</td>
</tr>
</tbody>
</table>

**Quadruple-Precision Format**

The IEEE 754 quadruple-precision (128-bit) floating-point format consists of:

- 1-bit sign indicator $s$
- 15-bit biased exponent $E$
- 115-bit fraction $F$

A representation of this format is given below.

<table>
<thead>
<tr>
<th>$b_{127}$</th>
<th>$b_{126}$</th>
<th>$b_{111}$</th>
<th>$b_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$E$</td>
<td>$F$</td>
<td></td>
</tr>
</tbody>
</table>

The relationship between this format and the representation of real numbers is given below.

<table>
<thead>
<tr>
<th>Number Characterization</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized, $0 &lt; E \leq 32767$</td>
<td>$(-1)^s \times (2^{E-1027}) \times (1.F)$</td>
</tr>
<tr>
<td>Denormalized, $E = 0; F \neq 0$</td>
<td>$(-1)^s \times (2^{-1022}) \times (0.F)$</td>
</tr>
<tr>
<td>Zero, $E = 0, F = 0$</td>
<td>$(-1)^s \times (0)$</td>
</tr>
<tr>
<td>Otherwise</td>
<td>Exceptional value</td>
</tr>
</tbody>
</table>

**IEEE Floating-Point Data Type Parameters**

The floating-point hardware adjusts the exponent automatically to ensure that the numbers are properly aligned for each arithmetic operation. Scaling and normalization take place according to a pre-determined set of rules built into the hardware. As a floating-point number grows, the exponent expands, allowing the number to encompass a larger dynamic range. Overflows occur when the exponent exceeds for positive limit. To avoid overflows, large numbers are accommodated by decreasing the precision, or increasing the significance, of each bit in the significand.
<table>
<thead>
<tr>
<th>Floating Point Data Type</th>
<th>Normalized Minimum</th>
<th>Maximum</th>
<th>Exponent Bias</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>$2^{-128} \approx 10^{-38}$</td>
<td>$(2 - 2^{-85}) \cdot 2^{127} \approx 3 \times 10^{38}$</td>
<td>127</td>
<td>$2^{-23} \approx 10^{-7}$</td>
</tr>
<tr>
<td>Double</td>
<td>$2^{-1022} \approx 2 \times 10^{-308}$</td>
<td>$(2 - 2^{-53}) \cdot 2^{1023} \approx 1.7 \times 10^{308}$</td>
<td>1023</td>
<td>$2^{-102} \approx 10^{-16}$</td>
</tr>
<tr>
<td>Extended</td>
<td>$2^{-6382} \approx 3.4 \times 10^{-4032}$</td>
<td>$(2 - 2^{-53}) \cdot 2^{16383} \approx 1.2 \times 10^{4032}$</td>
<td>16383</td>
<td>$2^{-61} \approx 10^{-20}$</td>
</tr>
<tr>
<td>Quadruple</td>
<td>$2^{-6382} \approx 3.4 \times 10^{-4032}$</td>
<td>$(2 - 2^{-115}) \cdot 2^{16383} \approx 1.2 \times 10^{4032}$</td>
<td>16383</td>
<td>$2^{-112} \approx 10^{-28}$</td>
</tr>
<tr>
<td>Custom</td>
<td>$2^{1-663} \approx (2 - 2^{-6} \cdot 2^{663})$</td>
<td>$2^{e-1} - 1$</td>
<td>$2^{-f}$</td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.2 Fixed-Point Representation

The representation of a number is known as a “fixed-point” representation because the radix point is fixed. Fixed-point representations differ from floating-point in that the location of the radix point separating the integer and fraction components of a number is implied in the usage of a number rather than explicitly represented using a separate exponent and significand.

#### Unsigned Fixed-point Number

An $n$-bit binary word, when interpreted as an unsigned fixed-point number, and take on values from a subset $P$ of the non-negative number given by

$$P = \{p/2^b | 0 \leq p \leq 2^n - 1, p \in \mathbb{Z}\}$$  \hspace{1cm} (2.1)

Note that $P$ contains $2^n$ elements. We denote such a representation $U(a, b)$ [12], where $a$ and $b$ represent the integral word length and fractional word length of the fixed-point number in respective. Here $a = n - b$. The format of the unsigned fixed-point number is shown in Figure 2.1.

![Figure 2.1: The Format of the Unsigned Fixed-point Number](image)

A $U(a, b)$ representation is given by the expression

$$x = (1/2^b) \sum_{i=1}^{n-1} x_i 2^i$$

$x_i \in \{0, 1\}, i = 0, 1, \ldots, n - 1$
Where \( x_i \) represents bit \( i \) of \( x \). The range of a \( U(a, b) \) representation is from 0 to \( (2^n - 1)/2^b = 2^b - 2^a \). For example, the 3-bit unsigned fixed-point \( U(1, 2) \) has the form \( b_2.b_1.b_0 \).

Note that since \( b = 2 \), the binary point is to the right of the second bit from the right (counting from zero), and thus the number has one integer bit and three fractional bits. This representation has a range of from 0 to \( 2^1 - 2^{-2} = 2 - 0.25 = 1.75 \)

In the following, we would like to introduce a circle representation method for the fixed-point numbers. With this method, all the numbers inside the dynamic ranges can be represented by a circle. The values inside the circle are represented with unsigned binary format without binary point, while those values outside it are the corresponding representations in decimal of the fixed-point numbers. Figure 2.2 illustrates the circle representation method for an unsigned fixed-point number with integral word length \( a = 1 \).

![Circle Representation Method for an Unsigned Fixed-point Number with Integral Word Length a = 1](image)

Figure 2.2: The Circle Representation Method for an Unsigned Fixed-point Number with Integral Word Length \( a = 1 \)

The unsigned integer representation can be viewed as a special case of the unsigned fixed-point rational representation where \( b = 0 \). Specifically, an \( n \)-bit unsigned integer is identical to a \( U(n, 0) \) unsigned fixed-point rational. Thus the range of an \( n \)-bit unsigned integer is

\[
0 \leq U(n, 0) \leq 2^n - 1 \tag{2.2}
\]

And it has \( n \) integer bits and 0 fractional bit. The unsigned integer representation is sometimes referred to as ”natural binary”.

Examples:
1. \( U(6, 2) \). This number has \( 6 + 2 = 8 \) bits and the range is from 0 to \( 2^6 - 2^{-2} = 63.75 \). The value of 100010.10 is

\[
(1/2^2)(2^1 + 2^3 + 2^7) = 34.5
\]

2. \( U(16, 0) \). This number has \( 16 + 0 = 16 \) bits and the range is from 0 to \( 2^{16} - 1 = 65535 \). The value 10010111100 is
\[(1/2^0)(2^2 + 2^3 + 2^4 + 2^5 + 2^7 + 2^{10} = 1212/2^0 = 1212\]

**Signed Two’s Complement Fixed-point Number**

An \(n\)-bit binary word, when interpreted as a signed two’s complement fixed-point rational, can take on values from a subset \(P\) of the rational given by

\[P = \{p/2^b | -2^{n-1} \leq p \leq 2^{n-1} - 1, p \in \mathbb{Z}\}\]

Note that \(P\) contains \(2^n\) elements. We denote such a representation \(A(a, b)\), where \(a = n - b - 1\). The format of the signed two's complement fixed-point number is shown in Figure 2.3.

![Figure 2.3: The Format of the Unsigned Fixed-point Number](image)

The value of a specific \(n\)-bit binary number \(x\) in an \(A(a, b)\) representation is given by expression [13],

\[x = 2^{n-b-1}(-x_{n-1} + \sum_{i=0}^{n-2} x_i2^{i-b})\]

where \(x_i\) represents bit \(i\) of \(x\). The range of an \(A(a, b)\) representation is

\[-2^{n-1-b} \leq x \leq 2^{n-1-b} - 2^{-b}\] (2.3)

From the equation, we note that the number of the bits in the magnitude term of the sum above (the summation, that is) has one less bit than the equivalent prior unsigned fixed-point representation and these bits are the \(n - 1\) least significant bits. It is for these reasons that the most significant bit in a signed two’s complement number is usually referred to as the sign bit.

For example, the 3-bit signed two’s complement fixed-point \(A(1, 1)\) has the form \(b_2b_1b_0\). This representation has a range of from \(-2^1 = -2\) to \(2^1 - 2^{-1} = 2 - 0.5 = 1.5\)

Figure 2.4 shows the circle representation method for a signed two’s complement fixed-point number with integral word length \(a = 1\). The values inside the circle are represented with signed two’s complement binary format without binary
point, while those values outside it are the corresponding representations in decimal of the fixed-point number.

![Circle Representation Method](image)

Figure 2.4: The Circle Representation Method for a Signed Two’s Complement Fixed-point Number with Integral Word Length $a = 1$

Example:

1. $A(5, 2)$. This number has $5 + 2 + 1 = 8$ bits and the range is from $-2^5 = -32$ to $2^5 - 2^{-2} = 31.75$. The value of 100110.01 is

\[2^5 \times (-1) + 2^0 \times 2 + 2^3 \times 2^{-2} + 2^4 \times 2^{-2} = -26.25\]

2. $A(13, 2)$. This number has $13 + 2 + 1 = 16$ bits and the range is from $-2^{13} = -8192$ to $2^{13} - 1/4 = 8191.75$. The value of 1000000010000.10 is

\[2^{13} \times (-1) + 2^1 \times 2 + 2^6 \times 2^{-2} = 4224.5\]

2.2.3 Dynamic Range and Precision

A fixed-point quantization scheme determines the dynamic range of the numbers that can be applied to it. Numbers outside of this range are always mapped to fixed-point numbers within the range when you quantize them. The precision is the distance between successive numbers occurring within the dynamic range in a fixed-point representation.

The dynamic range and precision depend on the word length and the fraction length.

- For an unsigned fixed-point number with word length $n$ and fractional length $b$, the range is from 0 to $2^n - 2^{-b}$.

- For a signed two’s complement fixed-point number with the word length $n$ and fractional length $b$, the range is from $-2^{n-1} - b$ to $2^{n-1} - 2^{-b}$.

- In either case the precision is $b$.  

28
2.2.4 Overflows and Scaling

When you quantize a number that is outside of the dynamic range for your specified precision, overflows occur. At this time, some machine will cause a HW error (exception). Overflows occur more frequently with fixed-point quantization than with floating-point, because the dynamic range of the fixed-point number is much less than that of the floating-point number with equivalent word length. So it is very important to select a proper word length of the fixed-point number to get the high precision without overflows.

Overflows can occur when you create a fixed-point quantized filter from an arbitrary floating-point design. You can normalize your fixed-point filter coefficients and introduce a corresponding scaling factor for filtering to avoid overflows in the coefficients.

2.3 Comparisons between Floating-point and Fixed-point Numbers

Let's compare the characteristics of floating-point and fixed-point numbers (assume signed two's complement) in the range and precision with the same word length $n = 64$ bits.

For the IEEE normalized double-precision floating-point number which consists of 1-bit sign bit, 11-bit biased exponent and a 52-bit fraction, the range is from $2^{-1022}$ to $(2 - 2^{-52})2^{1023}$, the precision is $2^{-52}$. With the fixed word length, there is a relation between the range and precision for a signed two's complement fixed-point number: the higher the precision is, the smaller is the range. These two parameters are all affected by the fractional word length $b$ which lies in the range $[0, n-1]$, here the word length $n = 64$. If we select the maximum value of $b = 62$, the highest precision is $2^{-62}$ and the smallest range is from $-2^1$ to $2^1 - 2^{-62}$. On the contrary, if the fractional word length is minimized to $b = 0$, the precision has the minimum value $2^{-0}$ and the range is maximized to $[-2^{63}, 2^{63} - 1]$. Now you may find

- The range of the floating-point number is much larger than that of the fixed-point number with equivalent word length.
- Overflows occur more frequently with fixed-point representation.

In this example, let's consider all the cases of overflows one by one.

Case 1: $x \leq 2^{-1022}$ or $x \geq (2 - 2^{-52})2^{1023}$.
Overflows occurs both in floating-point and fixed-point representations

Case 2: $2^{63} - 1 \leq x \leq (2 - 2^{-52})2^{1023}$ or $2^{-1022} \leq x \leq -2^{63}$.
Overflows occur in fixed-point representations of $x$, but they don't occur in floating-point representations
Case 3: otherwise
Both fixed-point value $x_{fi}$ and floating-point value $x_{fl}$ may be represented without overflows.

In the last case, the fixed-point value $x_{fi}$ can be obtained after rounding or truncation the floating-point value $x_{fl}$. The range of fractional word length $b$ in fixed-point representation is from 0 to 52 for different ranges of the numbers. The divergence between $x_{fl}$ and $x_{fi}$ increases when the value of $b$ decreasing.

2.4 Transformation from Floating-point to Fixed-point

Fixed-point digital signal processors are suitable for implementing a large volume of products economically because they are usually much cheaper and faster than floating-point signal processors. So the data type transformation from floating-point to fixed-point in high abstraction level would be useful to make the synthesis efficient.

2.4.1 Minimal Word Length Determination

In order to prevent overflows, we need to determine the minimal word length before implementing transformation operations. The word lengths not only impact the efficiency of the arithmetic but also relate to the cycle time and die area of the required processor.

We introduce the "worst case estimation" technique [14] to find an optimal integral word length($L_i$). The fractional word length($L_f$) of the fixed-point number is determined by the "statistical" method. The word length $n$ is represented as $n = L_i + L_f$.

Determination of the Integral Word Length ($L_i$)

The "worst case estimation" technique is used to estimate a number range by determining the minimum and maximum values of it. Then a safe integral word length can be selected to prevent unwanted overflows. It starts with range information for the source operators of the data flow graph and then combines range information by analyzing the data flow using the range propagation. A "worst case estimation" analysis is carried out at each operation, whereby the maximum and minimum result values are determined from the maximum and minimum values of the source operands.

When declaring numbers with type information, their range is automatically determined. Alternatively, a range of the source operands can be explicitly attributed. The operators which perform the arithmetic operations will now perform
the range propagation. The table below shows the range propagation (maximum-side only) for several operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>( Max = a\cdot max + b\cdot max )</td>
</tr>
<tr>
<td>a − b</td>
<td>( Max = a\cdot max − b\cdot min )</td>
</tr>
<tr>
<td>a ∗ b</td>
<td>( Max = \max(a\cdot max \ast b\cdot max, a\cdot max \ast b\cdot min, a\cdot min \ast b\cdot max, a\cdot min \ast b\cdot min) )</td>
</tr>
<tr>
<td>a / b</td>
<td>( Max = \max(a\cdot max / b\cdot max, a\cdot max / b\cdot min, a\cdot min / b\cdot max, a\cdot min / b\cdot min) )</td>
</tr>
</tbody>
</table>

The range of every number (including source inputs, intermediate results and outputs) in the data flow graph can be determined after finish analyzing DFG in a bottom-up fashion. Let’s define the integral word length \( L_i \) of signed two’s complement fixed-point numbers with the range values as the following equation.

\[
L_i = b + 1 : 2^{b-1} \leq MAX(|\min_R|, |\max_R|) \leq 2^b \quad (2.4)
\]

Here \( \min_R \) and \( \max_R \) mean the maximum and minimum value of all the numbers in the data flow graph.

Example: Determine the integral word length of the numbers in the function \( F = ab + (c - d) \).

Assume the ranges of the source fixed-point inputs are:
- \( a \in \{0, 1\} \)
- \( b \in \{-5, 4\} \)
- \( c \in \{8, 10\} \)
- \( d \in \{-3, -1\} \)

The data flow graph of the function \( F \) is shown in Figure 2.5.

![Dataflow Graph for the Function F](image)

Figure 2.5: Dataflow Graph for the Function F

In the bottom of the DFG, the arithmetic operation is an addition. The output range of the function \( F \) can be obtained by ”worst case estimation” technique.
- \( F_{Max} = e\cdot max + f\cdot max \)
\[ F_{\text{Min}} = e_{\text{min}} + f_{\text{min}}. \]

The worst case of \( e \) and \( f \) can be estimated by analyzing range propagation in the arithmetic of multiple and subtraction according to the ranges of source inputs in respect.

- \[ e_{\text{max}} = \max(a_{\text{max}} \times b_{\text{max}}, a_{\text{max}} \times b_{\text{min}}, a_{\text{min}} \times b_{\text{max}}, a_{\text{min}} \times b_{\text{min}}) \]
  \[ = 1 \times 4 = 4 \]
- \[ e_{\text{min}} = \min(a_{\text{max}} \times b_{\text{max}}, a_{\text{max}} \times b_{\text{min}}, a_{\text{min}} \times b_{\text{max}}, a_{\text{min}} \times b_{\text{min}}) \]
  \[ = 1 \times (-5) = -5 \]
- \[ f_{\text{max}} = c_{\text{max}} - d_{\text{min}} = 10 - (-3) = 13 \]
- \[ f_{\text{min}} = c_{\text{min}} - d_{\text{max}} = 8 - (-1) = 9 \]

Now we get the maximum and minimum value of \( F \).

- \[ F_{\text{Max}} = e_{\text{max}} + f_{\text{max}} = 13 + 4 = 17 \]
- \[ F_{\text{Min}} = e_{\text{min}} + f_{\text{min}} = 9 - 5 = 4 \]

The ranges of all the numbers in the DFG are listed below.

- \( a \in \{0, 1\}, b \in \{-5, 4\} \)
- \( c \in \{8, 10\}, d \in \{-3, -1\} \)
- \( e \in \{-5, 4\}, f \in \{9, 13\} \)
- \( F \in \{4, 17\} \)

In this example,

- \( \min_R = \min(0, -5, 8, -3, -5, 9, 4) = -5 \)
- \( \max_R = \max(1, 4, 10, -1, 4, 13, 17) = 17 \)

So \( \text{MAX}(|\min_R|, |\max_R|) = \max(-5, 17) = 17 \).

Since \( 2^4 \leq 17 \leq 2^5 \), the proper integral word length can be selected as \( m = 6 \) with the 2's complement binary representation.

**Determination of the Fractional Word Length \( (L_f) \)**

The "statistical" method [14] is based on a combined fixed-point/floating-point simulation. The goal is to determine and minimize precision loss due to the finite word lengths.

Each data contains two representations: fixed-point value \( x_f \) and floating-point value \( x_f \). The algorithm is executed with two simultaneous calculations for each operation in the same simulation. One is performing computations with
specified fixed-point behavior and another with the floating-point values. At the same time, we keep track of the difference between floating-point and fixed-point values $(\triangle)$. 

During the simulations, the error $\triangle$ from the input and already quantized error propagates through all data in the system. Let’s take the system in Figure 2.6 as an example. During data assignments the difference error statistics are collected, both for

- input difference error $\triangle_1 = a_{fi} - a_{ji}$, $\triangle_2 = b_{fi} - b_{ji}$, $\triangle_4 = d_{fi} - d_{ji}$
- intermediate difference error $\triangle_3 = c_{fi} - c_{ji}$
- output difference error $\triangle_5 = e_{fi} - e_{ji}$

The statistics includes the mean error($\mu$) and the standard deviation($\sigma$). These measures are directly related to the fractional word length $L_f$.

$$2^{-L_f} \leq \sigma \cdot \kappa_{em}$$ (2.5)

The $\kappa_{em}$ is the empirical constant which was found to give optimal results for the range (1,4). The smaller $\kappa_{em}$ is applied, the more conservative determination of the fractional word length is obtained.

In the system shown in Figure 2.6, the mean error $\mu = \frac{\triangle_1 + \triangle_2 + \triangle_3 + \triangle_4 + \triangle_5}{5}$. The standard deviation $\sigma^2 = \frac{\sum_{i=1}^{5} (\triangle_i - \mu)^2}{5}$. Assume

- $a_{fi} = 2.234e - 1$, $a_{ji} = 0.2$
- $b_{fi} = 3.512e - 1$, $b_{ji} = 0.3$
- $d_{fi} = 1.121e - 1$, $d_{ji} = 0.1$
- $\kappa_{em} = 2$

After implementing of arithmetic operations in the data flow graph, we get

- the intermediate difference error
  $\triangle_3 = c_{fi} - c_{ji} = a_{fi}b_{fi} - a_{ji}b_{ji} = 0.07845808 - 0.06 \approx 0.0185$
- output difference error
  $\triangle_5 = e_{fi} - e_{ji} = (c_{fi} + d_{fi}) - (c_{ji} + d_{ji}) = 0.19055808 - 0.16 \approx 0.0306$

Now we get the mean error($\mu = \approx 0.0251$) and the standard deviation $\sigma \approx 1.362e - 2$. $L_f \geq -\log_{2}(\frac{0.251}{1.362e - 2}) \approx 5.2$. So the minimal value of is determined to be $L_f = 6$. 

33
2.4.2 Transformation Steps with Determined Word Length

After determining the proper word length of the fixed-point, we may avoid the overflow effectively. The data type transformation from floating-point to fixed-point applies the semantic preserving refinement technique. The only change after transformation is the precision of the number, which will not affect the meaning of a model or process network. The detailed transformation steps with a determined word length are described as below.

Assume we need to transform the floating-point number $x$ into a signed two's complement fixed-point number $y$ with a defined word length $n$.

From the introduction of minimal integral word length determination above, we may decide the minimal value of integral word length $m = \lceil \log_2 |p| + 1 \rceil$. If the integral word length is minimized then $y$ has the highest precision $(n - m - 1)$.

Step 1: Break down the floating-point number $x$ into sign bit $s$, significand $f$ and exponent $e$ in binary. $x = (-1)^s \times f \times 2^e$.

Step 2: Shift the significand $f$ $e$ bits into $f'$ (if $e \geq 0$, right shift $e$ bits, else left shift $|e|$ bits).

Here the shifting operations are defined as shifting to multiply or divide by a power of two, this means moving the binary point in the shift direction, preserving the binary bit pattern.

Step 3: Assume the word length of $f'$ is $w$. If $n <= w$, truncate/round the low $(w - n + 1)$ bits, else fill the top bits of $f'$ with $(n - w - 1)$ sign bits. Then we get the fixed-point number $y$ after the transformation steps.

**Example 1**: Transform the single-precision floating-point number $x$ into a signed two's complement fixed-point number $y$ with the word length 8 bits. Assume $x = 20.375$.

Step 1: $x$ can be break down into the sign bit $s = 1$ (1 bit), significand $f = 0.10110011_2$ (23 bits), the exponent $e = 0..0101$ (8 bits).

Step 2: Right shift $f$ 5 bit, $f' = 10110.011_2$ (23 bits).

Step 3: Since the word length of $f'$ equals 23 which is larger than the required word length 8. We get the fixed-point value $y = 010110.01 = 20.25$ after truncating the low 16 bits of $f'$.

**Example 2**: Transform the single-precision floating-point number $x$ into a fixed-point number $z$ with word length 32 bits. Assume $x = 20.375$.

In this example, the word length requirement of fixed-point number is changed.
The only difference of transformation steps between example 1 and example 2 is in the step 3.

In this example, since the required word length \( n = 32 \) which is larger than the significant word length 23, in step 3 we need to fill the top bits of \( f' \) with 8 zeros. Then we get the fixed-point value \( z = 0 \ldots 0 \ 10110.011 \ 0 \ldots 0 = 20.375 \).

2.4.3 Error Discussion

In Figure 2.7, we illustrate the transformations from single-precision floating-point to fixed-point different with three cases.

![Diagram showing the transformation from floating-point to fixed-point](image)

Figure 2.7: The Examples of Transformation Procedure from Floating-point to Fixed-point
Case 1: 20.25 → 20.25 (m = 8)
Case 2: 20.375 → 20.25 (m = 8)
Case 3: 20.375 → 20.375 (m = 16)

Here m means the word length of the fixed-point number.

From the example, we find the errors after data type transformations \( \Delta_1 = \Delta_3 = 0, \Delta_2 = 0.125 \). The error depends on both the value of the floating-point number and the required word length of fixed-point.

In Case 1, we translate the floating-point number 20.25 to a fixed-point number with word length \( m = 8 \). Since the significand of the floating-point number 20.25 has 8-bit effective binary value, during the transformation, we keep all these 8-bit effective binary value and only the last 16-bit '0' are truncated.

In case 2, the significand of the floating-point number 20.375 has 9-bit effective binary value. With the same required word length of the fixed-point number, during the transformation, we only keep the first 8-bit binary value and the last one bit is truncated. In this case, error occurs.

In case 3, we increase the required word length of fixed-point number to 16, since the effective bits of the significand in the floating-point number 20.375 is less than 16, all these effective bits are kept and the error can be avoided.

From the discussion above, we find if select a proper word length, it is possible to decrease the value of error after the transformation from floating-point to fixed-point.

### 2.5 Fixed-point Arithmetic

The following are practical rules of fixed-point arithmetic [12]. Assume all the source input and output values are represented as the fixed-point representations with the defined word length \( m \).

#### 2.5.1 Addition/Subtraction

Two binary numbers must be pre-aligned by the binary point before they are added/subtract. The addition/subtraction is valid only when two operands has the same integral word length and fractional word length. Assume the word lengths of source operands are \( m \).

- **Unsigned Addition/Subtraction**
  
  \[
  U(a, m - a) \pm U(b, m - b) = U(c, m - c)
  \]

  where \( c = \max(a, b) + 1 \)

- **Signed Addition/Subtraction**
  
  \[
  A(a, m - 1 - a) \pm A(b, m - 1 - b) = A(c, m - 1 - c)
  \]

  where \( c = \max(a, b) + 1 \)
Each operands scales properly by right shift operations to make the two operands have the same integral word length \( \max(a, b) + 1 \) (the extra 1 bit is reserved to avoid overflow after addition/subtraction).

For the unsigned addition/subtraction, the shifting operations of \( X = U(a, m-a) \) and \( Y = U(b, m-b) \) are shown below

- Right shift \( X \ (\max(c, e) + 1 - a) \) bits, the extra 1 bit is reserved to avoid overflow after addition/subtraction.
- Right shift \( Y \ (\max(c, e) + 1 - b) \) bits.

For the signed addition/subtraction, the shifting operations of two operands are almost same with those of the unsigned addition/subtraction. The only difference between them is the meaning of the right shifting operation. In unsigned addition/subtraction, it means filling the top bits with ‘0’ and truncating the low bits with the same account of bits, while in signed addition/subtraction, the filling element is the sign bit instead of ‘0’ and the truncating operations are same with those in unsigned addition/subtraction.

The process of the fixed-point addition/subtraction with defined word length is shown in Figure 2.8

![Figure 2.8: Signed Fixed-point Addition/Subtraction with Defined Word Length](image)

Example: Add two signed two's complement fixed-point number \( X(1,2) = 01.1101, Y(2,1) = 010.101 \).

In this example, \( a = 1, b = 2, m = 6, \max(a, b) + 1 = \max(1, 2) + 1 = 3 \).

Align them by

- Right shift \( X \ 3 - 1 = 2 \) bits, 01.1101 \( \rightarrow \) 0001.11
- Right shift \( Y \ 3 - 2 = 1 \) bits, 010.101 \( \rightarrow \) 0010.10

Then we get the new operand \( X'(3, 2) = 0001.11, Y'(3, 2) = 0010.10 \). The addition is shown in Figure 2.9.

### 2.5.2 Multiplication

Assume the word lengths of source operands are \( m \).

- **Unsigned Multiplication**
\[
\begin{align*}
X'(3,2) & \quad 0001.11 \\
Y'(3,2) & + \; 0110.10 \\
Z(3,2) & \quad 1000.01
\end{align*}
\]

Figure 2.9: Addition of Signed Two’s Complement Fixed-point Numbers

\[
U(a, m - a) \times U(b, m - b) = U(a + b, 2m - (a + b))
\]

- **Signed Multiplication**

\[
A(a, m - 1 - a) \times A(b, m - 1 - b) = A(a + b + 1, 2m - 2 - (a + b))
\]

After the multiplication operations above, the word lengths of results will be \(2m\). To keep the word length of outputs unchanged, we only take the upper \(m\) bits and truncate the low ones [15]. So the final result of multiplications will be:

- **Unsigned Multiplication**

\[
U(a, m - a) \times U(b, m - b) = U(a + b, m - (a + b))
\]

- **Signed Multiplication**

\[
A(a, m - 1 - a) \times A(b, m - 1 - b) = A(a + b + 1, m - 2 - (a + b))
\]

The process of fixed-point multiplication with defined word length is shown in Figure 2.10.

Figure 2.10: Signed Fixed-point Multiplication with Defined Word Length

Example: Multiple two’s complement number \(X(0,3)\) with \(Y(1,2)\). \(X(0,3) = 1.001\), \(Y(1,2) = 00.11\). We illustrate the arithmetic procedure in Figure 2.11.

### 2.5.3 Division

The division arithmetic can be actually regarded as the multiplication between the first operand and the reciprocal of the second operand.
\[
\begin{array}{c|c}
X(0,3) & 1.001 \\
Y(1,2) & 0.011 \\
\hline
& \begin{array}{c}
+ \\
11001 \\
1001 \\
\hline
1101011 \\
+ \\
0000 \\
\hline
11101011 \\
+ \\
0000 \\
\hline
Z(2,5) & 111.01011 \\
\hline
\end{array}
\end{array}
\]

\[Z(2,1) \Downarrow \text{truncate} \]

Figure 2.11: Multiplication of Signed Two's Complement Fixed-point Numbers

- **Unsigned Division**
  \[ U(a_1, b_1)/U(a_2, b_2) = U(a_1 + a_2, \lfloor \log_2(2^{a_2 + b_1} - 2^{b_1 - b_2}) \rfloor) \]

- **Signed Division**
  \[ A(a_1, b_1)/A(a_2, b_2) = A(a_1 + b_2 + 1, a_2 + b_1) \]

### 2.5.4 Data Type Transformations in ForSyDe

In Haskell language, the real number is represented with the data type listed below.

- **Int** - integer which fits into one processor word, typically 32 bits
- **Integer** - integers of arbitrary bits
- **Float** - single-precision floating-point
- **Double** - double-precision floating-point

Since there is no definition of fixed-point representations in Haskell, we need to build the fixed-point modules with the libraries present in Haskell at first.

The module Fixedpoint defines the data types `Fix32`, `Fix16` and `Fix8`. They represent 32-bit,16-bit or 8-bit 2’s complement fixed-point respectively.

- **data Fix32 = Fix32 Int Int Int32**
- **data Fix16 = Fix16 Int Int Int16**
- **data Fix8 = Fix8 Int Int Int8**

In each data type, the value is specified by three elements. The first and second elements of it represent the whole word length and the integer word length of the fixed-point number separately. Both of them have the data type 'Int'. The third element is the binary value of the fixed-point with the data type 'Int8','Int16' or 'Int32' defined in the ”Bits” library.

For example, the value `Fix8 8 2 64` corresponds to 20. In this example the fixed-point has 8-bit word length, 2-bit integral word length and the binary value is 10000000. The binary point position locates in the 3rd bit counting from MSB(sign
bit). The precision of the fixed-point number is $8 - 1 - 2 = 5$ bits. The representation is shown in Figure 2.12.

![Figure 2.12: Data Type Representation of Signed Two’s Complement Fixed-point Number](image)

Figure 2.13 shows the procedure of the data type transformations in a functional model. Let’s denote all the functional operations with a uniform symbol “⊗” (including arithmetic, shifting and so on). The upper figure is the original functional model with the inputs and outputs data type—’Double’. The lower one implements data type transformations from ’Double’ to ’Fixed’ for the inputs and inserts a transformation model which implements the data type transformation from ’Fixed’ to ’Double’ before getting the outputs.

The model ”D/F” models data transformation from ’Fixed’ to ’Double’, while the data type transformation from ’Fixed’ to ’Double’ is implemented by the ”F/D” model.

![Figure 2.13: Data Type Transformation between Two Processes](image)

After building the data type of the fixed-point, the functions which implement the data type transformation from floating-point to fixed-point can be described in Haskell language using the method mentioned above.

In the special example ”digital equalizer”, the input signal and the filter coefficients of the FIR are present with data type—’Double’. To implement the data type transformations, we need to define two processes. In process ”D/F”, we define the data type transformation function from double-precision floating-point number to Fixed-point number as `doubleToFix m x`, where `m` is the word length of the desired fixed-point number and `x` is the double-precision floating-point value.
In process $"F/D"$, we define the data type transformation function from fixed-point number to double-precision floating-point number as \( \text{fixToDouble} \ (\text{Fix} \ m \ n \ x) \), where \( (\text{Fix} \ m \ n \ x) \) is the fixed-point value.

With the functions obtained, we construct them with the proper skeletons in the synchronous library. The data type refinement models are show in Figure 2.14.

![Figure 2.14: Data Type Refinement Models for Signals and Vectors](image)

In the following, we show the data type transformation functions in Haskell expression (here the word length of the fixed-point number is 16).

\[
\text{doubletoFix16} \ m \ x = \text{Fix16} \ m \ n \ (\text{round16} (\text{truncate} (a \times 2^{(m - n - 1 + b)}))) \\
\text{where} \ n = \text{toInt} (\text{ceiling} (\text{logBase} \ 2 \ (\text{fromInteger} (\text{fst} (\text{properFraction} (\text{abs} \ x))) + 1))) \\
a = \text{significand} \ x \\
b = \text{exponent} \ x
\]

\[
\text{fix16toDouble} \ (\text{Fix16} \ m \ n \ x) = (\text{fromInt} (\text{toInt} \ x)) / (2^{(m - n - 1)})
\]

In this code:

max:: Ord a => a -> a -> a
The larger of two items.

properFraction:: (RealFrac a, Integer b) => a -> (b, a)
Split a number into its whole and fractional parts.

fst:: (a, b) -> (a, b) -> a
Extract the first item of a pair.

abs:: Num a => a -> a
Find the absolute value of a number.

fromInteger:: Num a => Integer -> a
Convert numbers from Integer to another.
\[ y_n = \sum_{m=0}^{k} x_{n-m} h_m \quad (2.6) \]

The state of the FIR-Filter (Figure 1.12) can be seen as a queue with the finite size \( k + 1 \). The tail element in the queue at cycle \( n \) is \( x_n \) and the head element is \( x_{n-k} \). In the next cycle a new value \( x_{n+1} \) is shifted in the queue from the left, all other elements are shifted one place to the right, and the value \( x_{n-k} \) leaves the queue. All the source input values and intermediate calculation results are represented as fixed-point numbers with the word length \( m \) which is larger than the minimal word length decided by the method in section 2.4.1.

Here we introduce the fixed-point arithmetic in FIR in Haskell language. The module "Fixed" (see Appendix) is described in the fixed-point library. In this library, we not only defined the data type 2's complement fixed-point but also the some arithmetic operations, e.g. addition, subtraction, multiplication. The data types are instances of the class "Num", which means, that they can be used in all operations defined by "Num" such as multiplication. Every fixed-point number is described with three elements. \( X = \text{Fix} m n v \), here \( m \) is the word length, \( n \) is the integral word length and \( v \) means the binary value. The range of the fixed-point number is from \(-2^m - 1\) to \(2^m - 1\). For the detailed data type description of fixed-point, we have introduced in section 2.2.2. In the FIR Filter there are two kinds of fixed-point arithmetic: addition and multiplier. Assume the word length of fixed-point number \( m = 16 \) bits. The arithmetic can be expressed in Haskell as follows.

\[
\begin{align*}
\text{(Fix16 m n x) + (Fix16 m n' y)} & = \text{Fix16 m (i+1) ((shift x (n-i-1)) + (shift y (n'-i-1)))} \\
& \quad \text{where } i = \max n n' \\
\text{(Fix16 m n x) * (Fix16 m n' y)} & = \text{Fix32 2*m (n+n'+1) (toInt32 (toInt x) * (toInt y))}
\end{align*}
\]
\[
= \text{Fix16} \, m \, (n+n'+1) \, (\text{int32To16}(\text{shift} \, (\text{intTo32}(\text{toInt} \, x) \, \ast \, (\text{toInt} \, y))) \, (-m)))
\]

In this code:
\[
\text{shift}: \text{Bits} \, a \Rightarrow a \rightarrow \text{Int} \rightarrow a
\]
\[
\text{intTo32} : \text{Int} \rightarrow \text{Int32}
\]
\[
\text{int32To16} : \text{Int32} \rightarrow \text{Int16}
\]

**Arithmetic in the 1st cycle**

Let's take a 2-order FIR Filter with ranges of input signal and coefficients in [-1,1] as an example.

In the first cycle, \( y_0 = x_0 h_0 \).

Assume:
- \( x_0 = \text{Fix16} \, m \, n_{x0} \, v_{x0} \)
- \( h_0 = \text{Fix16} \, m \, n_{h0} \, v_{h0} \)

Here: \(-1 \leq x_0, h_0 \leq 1\).

The integral word length \( n \) of the fixed-point number \( y \) with the range between \( a \) and \( b \) can be calculated by:

\[
n = \text{toInt}(\text{ceiling}(\text{logBase} \, 2 \, (\text{fromInteger}(\text{fst} \, (\text{properFraction} \, (\text{abs} \, x))))+1)))
\]

After calculation, \( n_{x0} = n_{h0} = 0 \), the operands may be represented as
- \( x_0 = \text{Fix16} \, 16 \, 0 \, v_{x0} \)
- \( h_0 = \text{Fix16} \, 16 \, 0 \, v_{h0} \)

The process of arithmetic is shown in the Figure 2.15.

![Figure 2.15: FIR Arithmetic in the 1st Cycle](image)

**Step 1:** Multiple \( x_0 \) with \( h_0 \).

\[
x_0 \ast h_0 = (\text{Fix16} \, m \, n_{x0} \, v_{x0}) \ast (\text{Fix16} \, m \, n_{h0} \, v_1)
\]

\[
= (\text{Fix16} \, 16 \, 0 \, v_{x0}) \ast (\text{Fix16} \, 16 \, 0 \, v_1)
\]

\[
= \text{Fix32} \, 32 \, 1 \, v'
\]

43
where \( v' = \text{intTo16} \left( (\text{tolnt} \ v_{x0}) \ast (\text{tolnt} \ v_{h0}) \right) \).

The result of the multiplication \( v' \) has 32-bit word length.

Step 2: Truncate the low 16 bits by right shifting \( v' \) to keep the word length 16. The shifting result will be \( y_0 = \text{Fix} \times 16 \ v \), where

\[
v = \text{int32To16}(\text{shift} \ v' (-16))
= \text{int32To16}(\text{shift} \ (\text{intTo32} ((\text{tolnt} \ v_{x0}) \ast (\text{tolnt} \ v_{h0}))) (-16))
\]

After the arithmetic operations, the result has the range from -1 (exclusive) to 1 (exclusive).

**Arithmetic in the 2nd cycle**

In the second cycle, \( y_1 = x_0 h_1 + x_1 h_0 \).

The multiplication \( y_{10} = x_0 h_1 \), \( y_{11} = x_0 h_1 \) can be implemented using the same arithmetic in the first cycle and the results of the multiplication \( y_{10}, y_{11} \) have the same range \((-1, 1)\).

The addition operation procedure includes two steps.

Assume:

- \( y_{10} = \text{Fix} \times 16 \ m \ n_{y10} \ v_{10} \)
- \( y_{11} = \text{Fix} \times 16 \ m \ n_{y11} \ v_{11} \)

In the second cycle, the integral word lengths of both operands in the addition are 1. \( n_{y10} = n_{y11} = 1 \). The operands of the addition are represented as

- \( y_{10} = \text{Fix} \times 16 \ 1 \ v_{10} \)
- \( y_{11} = \text{Fix} \times 16 \ 1 \ v_{11} \)

The process of arithmetic is shown in the Figure 2.16.

![Figure 2.16: FIR Arithmetic in the 2nd Cycle](image)

Step 1: Scale both \( v_{10} \) and \( v_{11} \) with \( 2^{-1} \) time. The objective of scaling is to prevent overflows when adding them up.
\[ p = \text{shift } v_{10} (-1) \]
\[ q = \text{shift } v_{11} (-1) \]

This operation means shifting \( x \) to the left if \( i \) is positive and to the right otherwise. Here shifting operations perform sign extension on signed number types. That is, right shifting fill the top bits with 1 if the number is negative and with 0 otherwise.

Step 2: Add up \( p \) and \( q \) to get the third element of \( y_1 \). The integral word length \( n \) of \( y_1 \) is represented as \( n = \max(n_{10}, n_{11}) + 1 = 2 \).

Then the result of the second cycle can be represented as
\[ y_1 = Fix16 \ m \ n \ (p + q) = Fix16 \ 16 \ 2 \ ((\text{shift } v_{10} (-1)) + (\text{shift } v_{11} (-1))) \]

After the addition operation, the result \( y_1 \) lies in the range \((-2, 2)\).

**Arithmetic in the 3rd cycle**

In the 3rd cycle, \( y_2 = x_2 h_0 + x_1 h_1 + x_0 h_2 \)
Assume:

- \( y_{20} = Fix16 \ m \ n_{20} \ v_{20} \)
- \( y_{21} = Fix16 \ m \ n_{21} \ v_{21} \)
- \( y_{22} = Fix16 \ m \ n_{22} \ v_{22} \)

All the multiplier operations are implemented using the same method in the first cycle. The integral word length \( n_{20} = n_{21} = ... = n_{22} = 3 \).

The operands of the addition are represented as

- \( y_{20} = Fix16 \ 16 \ 1 \ v_{20} \)
- \( y_{21} = Fix16 \ 16 \ 1 \ v_{21} \)
- \( y_{22} = Fix16 \ 16 \ 1 \ v_{22} \)

The process of arithmetic is shown in the Figure 2.17.

The integral word length is decided by \( n = \max(\max(n_{20}, n_{21}) + 1, n_{22}) + 1 = 3 \).

The addition operations in this step are implemented with the same method in the second cycle. In this cycle, there are two addition operations. We consider the addition between \( y_{20} \) and \( y_{21} \) at first. After scaling and addition operations, the result will be \( Fix16 \ 16 \ 2 \ (p + q) \). It is needed to scale the result into \( Fix16 \ 16 \ 3 \ p' \) before adding with the scaling result of the third operands.

The result of the third cycle \( y_2 = Fix16 \ 16 \ 3 \ v \), where

\[ v = (\text{shift } ((\text{shift } v_{20} (-1)) + (\text{shift } v_{21} (-1))) (-1)) + (\text{shift } v_{22} (-2)) \]
Arithmetic in the 4th cycle

In the 4th cycle, \( y_3 = x_2 h_1 + x_1 h_2 = y_{40} + y_{42} \)

Assume:
- \( y_{30} = \text{Fix16 } m \ n_{30} \ v_{30} \)
- \( y_{31} = \text{Fix16 } m \ n_{31} \ v_{31} \)

Since the range of the input signal and coefficient in every cycle are same, we may implement the operations in the 4th cycle using the same method in the 2nd cycle. The word length and integral word length of the result in the 4th cycle are also same as those in the 2nd cycle.

After all the operations, the result will be
\[
y_3 = \text{Fix16 } 16 \ 2 \ ((\text{shift } v_{30} (-1)) + (\text{shift } v_{31} (-1)))
\]

Arithmetic in the 5th cycle

In the 5th cycle, \( y_4 = x_2 h_2 \)

Assume:
- \( x_2 = \text{Fix16 } m \ n_{x2} \ v_{x2} = \text{Fix16 } 16 \ 1 \ v_{x2} \)
- \( h_2 = \text{Fix16 } m \ n_{h2} \ v_{h2} = \text{Fix16 } 16 \ 1 \ v_{h2} \)

The operations in the 5th cycle are implemented with the same method in the 1st cycle. The word length and integral word length of the result in the 5th cycle are same as those in the 1st cycle.

After all the operation, the results will be \( y_4 = \text{Fix16 } 16 \ 1 \ v \), where
\[
v = \text{shift} \ (\text{intTo} \ 32 \ 32 \ ((\text{intTo } v_{x2} \) \ * \ (\text{intTo } v_{h2}))) \ (-16)
\]
2.5.6 The Refined FIR Model after Data Type Transformation

To refined the FIR with data type transformation, we need to insert two "D/F" models before the inputs in the original model to speed up arithmetic operations and we may also need to put a "F/D" model before the output of original model to keep the data type of the output unchanged.

The data type refinement model can be expressed in Haskell. Here the word length of the fixed-point numbers is defined as the parameter $m = 16$. $h$ is the vector of coefficients. Figure 2.18 shows the refined model.

![Figure 2.18: The Data Type Refinement Model of the FIR](image)

```haskell
module FIR where

import ForSyDeStdLib
import Fixed
import Vector

fir m h s = mapSY fix16toDouble' (mapSY (ipV (mapV (doubletoFix16' m) h)) (mapSY (doubleFix16V m) (sipoSY (lengthV h) s)))
    where k = lengthV h

doubleFix16V m v = mapV (doubletoFix16' m) v

doubleFix16SY m s = mapSY (doubletoFix16' m) s

sipoSY n s = scandSY shiftrV (copyV n 0.0) s

ipV h = (accV . mulV h) v

accV NullV = 0

accV (x:+xv) = x + accV xv

mulV NullV NullV = NullV

mulV (x:+xv) (y:+yv) = x * y :+ (mulV xv yv)

In the following, we will illustrate a concrete FIR example with the coefficients and inputs. By the Matlab tool, we specify a 2-order FIR filter with amplification 0 between 0 and 100 Hz and 1 between 150 and 22050 Hz. The coefficients is
```

47
shown below.

```matlab
>> fs = 44100

>> lp = firls(2, [0 100 150 22050] / (fs/2), [0 0 1 1])

lp =
-0.0046 0.9954 -0.0046
```

Assume the input signal of the FIR $s_{(0.0:-1.0:-2.0:-NullS)}$, the coefficient vector $h=(-0.0046;>0.9954;>-0.0046;>$ NullV).

You may find the Haskell code for data type transformation, Fixed-point arithmetic and the test code for the arithmetic in every clock cycle of refined FIR model in Appendix.
Chapter 3

Refinement in Clock Domain and Resource Sharing

This chapter deals with refinement in clock domain and resource sharing. Ingo Sander et. al. have developed and documented the theoretical work in [7]. Using the theoretical work, the thesis works on building some functions in Haskell to do clock domain refinement and resource sharing.

3.1 Refinement in Clock Domain

Asynchronous hardware allows separate execution units to run on different clock signals. This can be useful in situations where the tasks being performed by execution units take an unpredictable amount of time to complete. It may also be useful where execution units operate at very different rates and it is more cost-effective to drive them with different clocks than to divide down one master clock for the slower execution unit. Operating some execution units with a slower local clock can reduce power consumption. Another benefit of globally asynchronous hardware is that the problem of clock distribution over a large circuit is simplified, so that clock signals do not need to be coherent throughout the circuit in order for it to function properly. Also the asynchronous circuits which are bounded by timing constraints, can be synthesized more efficiently.

3.1.1 The Multi-clock in ForSyDe

The ForSyDe methodology is based on the perfect synchrony hypothesis. The specification model is a network of concurrent synchronous processes which have the same data rate normalized to 1. After refining the processes, we may get an implementation model which is a network of concurrent synchronous processes and domain interfaces. They communicate with each other by means of signals which may be of any rate \( r \in Q \). In the specification model, all the signals in the system have one global clock. The occurrence of all events is mapped onto cycles of a single global clock. While in the implementation of the system, the processes may have different clock domains: main domain and sub domain. From this point of view,
the system is globally asynchronous. But in each clock domain, all the signals are perfectly synchronous with the clock. We may consider it as locally synchronous. For such a system we called globally-asynchronous, locally synchronous (GALS).

Figure 3.1 shows four domain interfaces for up-sampling and down-sampling which either use head($D_H(n), U_H(n)$) or tail($D_T(n), U_T(n)$) synchronization.

\[
S = \{v_0, v_1, \ldots\} \rightarrow U_H(n) \quad S' = \{v_0, v_1, \ldots\} \\
S = \{v_0, v_1, \ldots\} \rightarrow U_T(n) \quad S' = \{v_0, v_1, \ldots\} \\
S = \{v_0, v_1, \ldots, v_{n-1}, v_n\} \rightarrow D_H(n) \quad S' = \{v_n, v_{2n}, \ldots\} \\
S = \{v_0, v_1, \ldots, v_{n-1}, v_n\} \rightarrow D_T(n) \quad S' = \{v_{n-1}, v_{2n-1}, \ldots\}
\]

Figure 3.1: Domain Interfaces for Sampling

In up-sampling interface, the data rate of the output signal is $n$ times the input signal data rate. Every event of the input signal is emitted follow or after $n-1$ 'Abst' value at each output clock cycle. In contrast to up-sampling interface, down-sampling interface has a inverse data rate characteristic. The head/tail of every $n$ events of the input signal is produced at each output clock cycle.

Domain interfaces are also expressed with Haskell. Here we give the Haskell expression for $D_H(n), D_T(n), U_H(n)$ and $U_T(n)$ respectively.

1. down-sampling $D_T(n)$ domain interface with head synchronization

```haskell
downTSY n NullS = NullS
downTSY n xs = (atS (n-1) xs) :: (downHSY n (dropS (2*n-1) xs))
```

```haskell
atS _ NullS = error " atS: Signal has not enough elements"
atS 0 (x:_) = x
atS n (_:xs) = xs
```

The function `atS` returns the $n$-th event in a signal.

2. down-sampling $D_H(n)$ domain interface with tail synchronization
downHSY n NullS = NullS
downHSY n xs = (headS xs) - (downHSY n (dropS n xs))

headS NullS = error "headS: Signal is empty"
headS (x:_) = x

The function headS gives the first value — the head of a signal.

3. up-sampling $U_T(n)$ domain interface with head synchronization

\[
\text{upTSY n NullS} = \text{NullS} \\
\text{upTSY n (x:xs)} = \text{delaynSY} \text{ Abst (n-1) (unitS x)} + \text{upTSY n xs}
\]

delaynSY:: a -> Int -> Signal a -> Signal a

\[
\text{delaynSY e n xs} \mid n < 0 = \text{xs} \\
| \text{otherwise} = e \cdot \text{delaynSY e (n-1) xs}
\]

The skeleton delaySY delays the signal e event cycles by introducing n identical default value at the beginning of the output signal.

The operator (+-+) concatenates two signals into one signal.

4. up-sampling $U_H(n)$ domain interface with tail synchronization

\[
\text{upHSY n NullS} = \text{NullS} \\
\text{upHSY n (x:xs)} = \text{unitS x} + \text{takeS (n-1) (delaynSY Abst (n-1) NullS)} + \text{upHSY n xs}
\]

In the following Figure 3.2, we illustrate the domain interfaces with an example.

![Diagram](image)

Figure 3.2: An Example for Domain Interfaces

### 3.1.2 Clock Domain Refinement in a Digital Equalizer

Figure 1.14 shows the Audio analyzer subsystem, which includes a Fast-Fourier Transform (FFT) algorithm. This function fft takes a vector of $n = 2^k$ samples and produces the corresponding FFT result in form of a vector of size $n$ that is denoted as $[x_0, x_1, ..., x_{n-1}]$. The process Group Samples($G(n)$) reads $n$ samples and groups them into a vector of size $n$. This computation of this vector takes $n$ event cycles and serves as input for the FFT. But since we use a synchronous computational model in the specification model, an output needs to be produced for each input event, which leads to the occurrence of $n - 1$ absent values ($\perp$) in
the output. $G(n)$ is formally defined as [7].

$$G(n)(<v_0, v_1, ..., v_{n-1}, ...>^T) = \langle \underbrace{\ldots, \ldots, \ldots}_{n-1}, \underbrace{v_0, v_1, ..., v_{n-1}}_{1}, ... >^T$$

The process $P_\perp(n)$ has to process all absent values. This is not a drawback for the specification phase, but a direct implementation as shown in Figure 3.3 can make no use of the fact, that the FFT has only to be calculated at each $n$-th clock cycle.

![Figure 3.3: Direct Implementation of the Audio Analyzer](image)

Such an implementation will be very slow, since the computation of the FFT function is clearly the most time consuming and will determine the overall system performance.

In order to get a more efficient specification, the ForSyDe methodology allows introducing synchronous sub-domains into the system model during the refinement process. These synchronous sub-domains use another set of tags. The introduction of a new sub-domain is done by well defined transformations, which are semantic-preserving though they introduce an additional timing domain into the model.

Using the special characteristic of the grouping process $G(n)$ [7], it can be derived the identity

$$G(n) = U_T(n) \circ D_T(n) \circ G(n)$$

Using another identity which can easily be proven

$$P_\perp(n) \circ U_T(n) = U_T(n) \circ P_\perp$$

it follows that

$$P_\perp \circ G(n) = P_\perp(n) \circ U_T(n) \circ D_T(n) \circ G(n)$$

$$= U_T(n) \circ P_\perp(n) \circ D_T(n) \circ G(n)$$

$$= U_T(n) \circ P(n) \circ D_T(n) \circ G(n)$$

In the last step $P_\perp(n)$ is replaced with $P(n)$ since $D_T(n) \circ G(n)$ does not produce any absent values. By the use of semantic preserving transformations, it
follows that the refined process network has the same semantic meaning as the
process network of Figure 1.14 and thus the same characteristic functions. An-
alyzing the equation above, it can be concluded that the process \( P(n) \) processes
events only at each \( n \)-th tag and thus can be implemented with a slower clock.

To verify the equation above, we illustrate one example in the following with
the Haskell language. In this example, the process \( \text{Group Sample}(G(n)) \) reads \( n \)
samples and groups them into a vector of size \( n \). The function \( \text{groupSY} \) groups
values into a vector of size \( n \) and output it every \( n \)-th cycle. For the other cycles,
the outputs from this process are absent values. Since the \( \text{groupSY} \) is modelled
by a Moore FSM, so there is one cycle delay due to the initial state. The function
of \( \text{Sum Vector} \) in process \( P_n \) is to get the sum of all the elements in the input Vector.

```haskell
module ClockDomain where
import ForSyDe.StdLib
sumVector Abst = Abst
sumVector (Prst vs) = Prst (foldLV (+) 0 vs)
fSum n s = mapSY sumVector (groupSY n s)
quSum n s = mapSY sumVector (upTSY n (downTSY n (groupSY n s)))
gSum n s = upTSY n (mapSY sumVector (downTSY n (groupSY n s)))
pSum n s = upTSY n (mapSY abstExt (mapSY f (downTSY n (groupSY n s)))
                                   (downTSY n (tailS(groupSY n s))))

g \ x = fromAbstExt 1 (sumVector x)

abstExt v = Prst v

fromAbstExt x Abst = x
fromAbstExt _ (Prist y) = y

sumVector Abst = Abst
sumVector (Prist vs) = Prist (foldLV (+) 0 vs)

foldLV _ a NullIV = a
foldLV f a (x:xs) = foldLV f (f a x) xs

The test code and result of this module are shown in the following.

ClockDomain> fSum 2 (1:2:3:4:5:NullS) {3,7}

ClockDomain> qSum 2 (1:2:3:4:5:NullS) {3,7}

ClockDomain> gSum 2 (1:2:3:4:5:NullS) {3,7}

ClockDomain> pSum 2 (1:2:3:4:5:NullS) {3,7}
```

53
From the test result, we can confirm the clock domain transformation is semantic preserved.

The input signal, intermediate results and output signal of model \( f_{Sum} \) and \( g_{Sum} \) are shown in Figure 3.4

\[ \text{Figure 3.4: Implementation of Clock Domain Refinement Using Identity} \]

The synchronous sub-domain is implemented with a clock frequency \( f_{C2} \) that is \( n = 2^k \) times slower than the clock frequency \( f_{C1} \) of the main synchronous domain. The implementation of this transformation in the Audio Analyzer is illustrated in Figure 3.5.

### 3.2 Resource Sharing

It is commonly observed that the most revolutionary applications of the resource sharing become increasingly evident. Resource sharing enables new possibilities for collaboration and mutual benefit by reusing one function unit instead of many identical function units. It reduces the amount of hardware needed to implement operators such as addition (+) or multiplication (*) in hardware synthesis.

#### 3.2.1 Domain Interfaces in the Refinement of Resource Sharing

The main scheme of resource sharing is pipeline. In order to implement resource sharing, we need a transformation from parallel to series. In the perfectly synchronous model, the event of the signal is emitted every cycle sequentially. After the parallel-to-series transformation, the operation of serial events in the process in some sense can be considered as an implementation of pipeline. Here we introduce
Figure 3.5: The Audio Analyzer after Refinement

the domain interfaces of $P_n/S$ and $S/P_n$ [7] for parallel/series conversion (Figure 3.6) in the implementation model.

![Diagram](image)

Figure 3.6: Domain Interfaces for Parallel/Serial Conversion

$$P_n/S (s_1^\Gamma, s_2^\Gamma, ..., s_n^\Gamma) =  \langle s_1^\Gamma \uparrow 0, s_2^\Gamma \uparrow 0, ..., s_n^\Gamma \uparrow 0, s_1^\Gamma \uparrow 1, s_2^\Gamma \uparrow 1, ... \rangle^{nr}$$

$$S/P_n s_r = (s_1^{r/n}, s_2^{r/n}, ..., s_n^{r/n})$$

where

$$s_k^{r/n} = s \uparrow k - 1, s \uparrow n + k - 1, s \uparrow 2n + k - 1 \rangle^{nk}$$
In the $P_n/S$ interface, a serial output signal is produced by $n$ parallel inputs. The data rate of output is $n$ times that of the input signals. In the $S/P_n$ interface, the serial signal is converted to $n$ parallel output signals. The data rates of the outputs is $1/n$ times that of the input.

These two domain interfaces can also be described in Haskell. Here we give the Haskell expression for both $P_n/S$ and $S/P_n$ when $n = 2$.

1. $P_n/S$ domain interface ($n=2$)

   \[
   \text{parSer2SY} :: \text{Signal } a \rightarrow \text{Signal } a \rightarrow \text{Signal } a
   \]

   \[
   \text{parSer2SY} \; \text{NullS} \; \text{NullS} = \text{NullS}
   \]

   \[
   \text{parSer2SY} \; (x \cdot \text{xs}) \; (y \cdot \text{ys}) = x \cdot y \cdot \text{parSer2SY} \; \text{xs} \; \text{ys}
   \]

   \text{parSer2SY} takes two signals as inputs and produces another signal as output. Each event cycle a value from both input signals is taken and two output values are produced.

2. $S/P_n$ domain interface ($n=2$)

   \[
   \text{serPar2SY} :: \text{Signal } a \rightarrow (\text{Signal } a, \text{Signal } a)
   \]

   \[
   \text{serPar2SY} \; \text{NullS} = (\text{NullS}, \text{NullS})
   \]

   \[
   \text{serPar2SY} \; \text{xs} = (\text{headS} \; \text{xs}, \text{fst}(\text{serPar2SY} \; (\text{dropS} \; 2 \; \text{xs})), (\text{AtS} \; 1 \; \text{xs}), \text{snd}(\text{serPar2SY} \; (\text{dropS} \; 2 \; \text{xs})))
   \]

   \text{serPar2SY} takes one signal as input and produces two signals as outputs. Each event cycle two values from one input signal are taken and one value is produced for both outputs signals.

In Figure 3.7, we illustrate a transformation for resource sharing. In the specification model, there are two 2-input function units. Every clock cycle, two function results are produced with four input events. In ForSyDe, we may implement resource sharing by establishing the synchronous sub-domain which includes two parallel-to-serial and one serial-to-parallel domain interfaces in the implementation model. After transformation, we only need one function unit and reduce the amount of hardware.

The refined model with the function $f$ can be expressed in Haskell as follow.

\[
\text{resourceshare2SY} \; \text{NullS} \; \text{NullS} \; \text{NullS} \; \text{NullS} = (\text{NullS}, \text{NullS})
\]

\[
\text{resourceshare2SY} \; s_1 \; s_2 \; s_3 \; s_4 = \text{serPar2SY} \; (\text{zipWithSY} \; f \; (\text{parSer2SY} \; (s_1 \; s_2) \; (\text{parSer2SY} \; s_3 \; s_4)))
\]

In the following, we illustrate an example of the refined adder sharing model. (Figure
A synchronous sub-domain can be also be established by the use of a parallel-to-serial and a down-sample domain interface (Figure 3.9) [7]. Inside the synchronous sub-domain, an FSM-process based-on the process constructor is used \textit{mooreSY} in order to implement a serial version of the combinational \(n\)-input process.

In the following, the serial FSM implementation of the combinational \(n\)-input process is given. This process \(P_{FSM}\) performs the corresponding operation in \(n\) event cycles. The states of this process is a tuple where the first value represents the calculation step and the second value is the intermediate result.

The Haskell code of the process \(pFSM\) which implements the function units sharing together with a parallel-to-serial and a down-sample domain interface can be expressed as below. Using the refined model, we can implement for four kinds of function units sharing. They are adder sharing, subtractor sharing, multiplier sharing, and divider sharing. Some other functions can be also included in this process. The parameter \(k\) represents the type of operations.

\[
pFSM \ k \ s = \text{tailS} \ (\text{mooreSY} \ (f \ k) \ g \ (0,1) \ s) \text{ where }
f \ k \ x \ y = \text{if} \ (\text{fst} \ x = 0) \ \text{then}
\begin{align*}
(1, y) \\
\text{else if} \ ((\text{fst} \ x) < n-1) \ \&\& \ ((\text{fst} \ x) > 0) \ \text{then}
(\text{fst} x + 1, \text{tran} \ (\text{snd} x) y) \\
\text{else} \ (0, \text{tran} \ (\text{snd} x) y)
\end{align*}
\]
Figure 3.8: A Refined Adder Sharing Model using Parallel-to-serial and Serial-to-parallel Domain Interfaces

Figure 3.9: A Resource Sharing Solution using Parallel-to-Serial and Down-sampling Domain Interfaces

\[
\text{trans } k \times y | \begin{cases} 
  k \rightarrow ' + ' = x + y \\
  k \rightarrow ' - ' = x - y \\
  k \rightarrow ' \times ' = x \times y \\
  k \rightarrow '/ ' = x / y 
\end{cases}
\]

\[
g \ x \ = \ \text{if} \ (\text{fst} \ x) = 0 \ \text{then} \\
\quad \text{abstExt (snd} \ x) \\
\quad \ \text{else} \\
\quad \text{Abst}
\]

The process pFSM modelled by the process constructor \texttt{moorSY} has an initial state, which leads to one clock cycle delay in the output signal. To drop this unexpected event and avoid the effect to the next process (\texttt{downTSY}), we can im-
implement it by of the function $\text{tail}$. The refined model of the multiplier sharing for three input signals is shown in Figure 3.10.

![Diagram](image)

Figure 3.10: A Multiplier Sharing Model using Parallel-to-serial and Down-sampling domain Interfaces

### 3.2.2 Resource Sharing in a Digital Equalizer

Using the hardware semantics we have translated the FIR-Filter specification model of Figure 1.13 into synthesizable VHDL. The hardware structure is shown in Figure 3.11 and contains $k + 1$ multipliers and $k$ adders. While this is a fast FIR-Filter implementation it also consumes a large amount of area.

If performance is not the critical issue the FIR-Filter specification model can be refined into a structure that reuses one multiply-accumulate function block, while the process $\text{sipoSY}_k$ remains unchanged.

Using the definition of the process $mapSY\ ipV(h)$

$$
mapSY(ipV(h)) = P
$$

where

- $s' = P(s)$
- $h = [h_0, h_0, ..., h_0]$  
- $s \uparrow i = [x_i(0), x_i(1), ..., x_i(k)]$
Figure 3.11: Direct Implementation of the FIR-Filter Specification Model

\[ s' \uparrow i = h_0 x_i(0) + h_1 x_i(1) + h_k x_i(k) \]

After refinement, the implementation model of the FIR-Filter can be expressed in Haskell. The output of the \textit{sipoSY}_k is a signal of vector. Each event is a vector of \( k + 1 \) elements. Another input of the \textit{mapSY}(ipV(h)) is the coefficients \( h_0, \ldots, h_k \) which is also a vector of size of \( k + 1 \). We implement both the coefficient vector \( h \) and every event of the output signal from \textit{sipoSY}_k as rotating shift registers. In the new \textit{ipV} process, we may reuse one adder and one multiplier, which makes the refined FIR-Filter is much more efficient in hardware area.

\begin{verbatim}
  fir h s = mapSY (ipV (lengthV h-1) h) (sipoSY (lengthV h) s)
    where k = lengthV h

  sipoSY m s = scanSY shiftV (copyV m 0.0) s

  ipV 0 hv xv = headV hv * headV xv
  ipV n hv xv | (n < lengthV hv) = headV (rotV n hv) * headV
                   (rotV n xv) + ipV (n-1) hv xv
                  | otherwise = 0

  rotV n vs | n==0 = vs
             | otherwise = rotV (rotV (n-1) vs)

  rotV :: Vector a -> Vector a
  rotV NullV = NullV
  rotV vs = tailV vs <+ headV vs

  The function rotV rotates a vector to the right.
  The function rotV right rotates a vector with \( n \) times.

  The test code for two vectors of size 3 is

\end{verbatim}
Resourcesharing > ipV 1 (1:2:3:NullV) (1:2:3:NullV)

Resourcesharing > ipV 2 (1:2:3:NullV) (1:2:3:NullV)

There are two inputs of the refined function ipV. The first one is an event of a signal which is a vector of size \( n \) and the other one is a vector of coefficient \( h \). The operands of the multiplication are the head of two input vectors. The resource shared FIR by vector rotation is shown in Figure 3.12.

![Diagram of Resource Shared FIR by Vector Rotation](image)

**Figure 3.12: Resouce Shared FIR by Vector Rotation**

In the first operation cycle, we take the heads of both input vectors and implement the multiplication operation between them. The result of the multiplication is saved into a register as the accumulator initial value. In the rest of operation cycles, the same arithmetic operations are repeated \( n = length_{hv} h_{v} - 1 \) times as following. At first, each tails of the input vectors moves to the head by the operation of vector rotation. Then, the multiplication result and the accumulator value of previous cycle are added and a new accumulator value is generated. Finally, the new value is saved in the register. At the last operation cycle, the value in the register is output as an event.
Chapter 4

Summary

The ForSyDe methodology allows design refinement inside the functional domain. It uses design transformations, semantic-preserving and design decisions, to change a specification model into an implementation model. Exploration of design space inside the functional domain will be cheaper compared with doing so in the implementation domain, because no real implementations need to be taken.

4.1 Conclusion

Based on previous work, this thesis has discussed some aspects of design refinement in ForSyDe, namely, data type transformations, clock domain refinement and resource sharing. We have developed some Haskell libraries 'FixedPoint', 'ClockDomain' and 'Resourcesharing' to realize them. Using the developed libraries, we have conducted a case study on the Audio Filter and Audio Analyzer of the digital equalizer system. The specific refinements used upon the digital equalizer is shown in Figure 4.1 where ”1” denotes Data Type Transformation, ”2” Refinement of Clock Domain, and ”3” resource sharing, respectively.

4.2 Future Work

A number of future research opportunities exist. We have discussed three kinds of refinement methods in this thesis. However, to refine a purely function specification into an implementation model covers a wide range of topics. For example, a complex communication channel may be needed for process communications. Asynchronous interfaces only cope with multiple clock domains. There are many alternatives, for instance, FIFO channel. Even a communication channel maybe a bus, or a type of network interconnection. In that case, complex transformations are required. Another point is to automate the well-defined transformations.
Figure 4.1: The Refinements on the Digital Equalizer
Chapter 5

Appendix

5.1 The Fixed Point Library

Description: The module Fixed implements the data type Fixed, that models a scalable fixed-point value which ranges in \([- (2^m - 1), 2^m - 1]\) (here \(m\) is the word length of the fixed-point number). A value is specified by three elements. The first one is the word length \(m\) with data type 'Int', the second one is the integral word length \(n\) with data type 'Int' and the last one means the binary value \(v\) with data type 'Int8/Int16/Int32', e.g. the value Fix8 8 6 64 corresponds to 2.0 with a precision of 5 bits. The library also implements the arithmetic operations of two fixed-point numbers. All the fixed-point numbers including the operators and results of the operations have the same word length.

The functions doubletoFix8, doubletoFix16, doubletoFix32 translate the double-precision floating-point number to fixed-point number with word length \(m=8/16/32\).

The functions fix8toDouble, fix16toDouble, fix16toDouble translate the fixed-point number with word length \(m=8/16/32\) to double-precision floating-point number.

The functions f8, f16, f32 implement the addition operations of the fixed-point numbers with word length \(m=8/16/32\).

The functions g8, g16, g32 implement the multiplications of the fixed-point numbers with word length \(m=8/16/32\).

module Fixed where

import Int
import Bits

data Fixed8 = Fix8 Int Int Int8

class FixedPoint a where
  fix8toDouble :: a -> Double
  doubletoFix8 :: Int -> Double -> a

instance FixedPoint Fixed8 where
  fix8toDouble x = fix8toDouble' x
doubleToFix8 m x = doubleToFix8' m x

doubleToFix8' m x = Fix8 m n (IntTo8trunc((a * 2 ∧ (m - n - 1 + b))))
  where n = toInt(ceiling(logBase 2 (fromInteger(fst (properFraction (abs x))))) + 1))
  a = significand x
  b = exponent x

fix8toDouble' (Fix8 m n x) = (fromInt(toInt x)) / (2 ∧ (m - n - 1))

instance Show Fixed8 where
  showsPrec p (Fix8 m n x) = shows (fix8toDoubleDouble ( Fix8 m n x)) .
    showString "(" . shows n . showString " Bit")"

instance Read Fixed8 where
  readsPrec _ x = readsFixed8 x
readsFixed8 s = [(doubleToFix8 m x, e)] | (x, a) <- reads s,
  ("","", b) <- lex a,
  (m, c) <- reads b,
  ("Bit","", d) <- lex c,
  (""")", e) <- lex d

instance Eq Fixed8 where
  Fix8 m n x == Fix8 m' n' y = m == m' && n == n' && x == y

instance Num Fixed8 where
  (Fix8 m n x) + (Fix8 m' n' y) = case (m == m') of
    True -> if (n >= 0) && (n < = m - 1) && (n' >= 0) && (n' <= m' - 1)
      then Fix8 m (i+1) ((shift x (n-i-1)) + (shift y (n'-i-1)))
      else error "Operation (+): IWL must be between 0 and WL."
    False -> error "we must extend the WL of the operand whose WL is smaller to the maximal value before addition"

    where i = max n n'

  (Fix8 m n x) * (Fix8 m' n' y) = case (m == m') of
    True -> if (n >= 0) && (n < = m - 1) && (n' >= 0) && (n' <= m' - 1)
      then Fix8 m (n+n'+1) (Int16To8 (shift (Int16To8 (toInt x) * (toInt y)) (-m)))
      else error "Operation (+): IWL must be between 1 and WL."
    False -> error "we must extend the WL of the operand whose WL is smaller to the maximal value before multiplication"

  data Fixed16 = Fix16 Int Int16
class FixedPoint a where
  fix16toDouble :: a -> Double
  doubletoFix16 :: Int -> Double -> a

instance FixedPoint Fix16 where
  fix16toDouble x = fix16toDouble' x
  doubletoFix16 m x = doubletoFix16' m x

doubletoFix16' m x = Fix16 m n (intToInt16(truncate(a * 2 \ (m - n - 1 + b))))
where n = toInt(ceiling(logBase 2 (fromInteger(fst
  (properFraction (abs x)))+1))
   a=significand x
   b=exponent x

fix16toDouble' (Fix16 m n x) = (fromInt(toInt x)) / (2 \ (m - n - 1))

instance Show Fix16 where
  showsPrec p (Fix16 m n x) = shows (fix16toDouble (Fix16 m n x) .
    showString"(\ . shows n. showString " Bit)"

instance Read Fix16 where
  readsPrec _ x = readsFixed16 x
readsFixed16 s = [(doubletoFix16 m x, e) | (x,a) <- reads s,
  ("\",b) <- lex a,
  (m,c) <- reads b,
  ("Bit",d) <- lex c,
  ("\")\',e) <- lex d]

instance Eq Fix16 where
  Fix16 m n x == Fix16 m' n' y = m == m' && n==n' && x == y

instance Num Fix16 where
  (Fix16 m n x) + (Fix16 m' n' y) = case (m==m') of
    True -> if (n>0) && (n < = m-1) && (n'>=0) && (n'<= m'-1)
    then
      Fix16 m (i+1) ((shift x (n-i)) + (shift y (n'-i)))
    else
      error "Operation (+): GWL must be between
        0 and WL."
    False -> error"we must extend the WL of the operand whose
      WL is smaller to the maximal value before addition"
    where i = max n n'
  (Fix16 m n x) * (Fix16 m' n' y)= case (m==m') of
    True -> if (n>0) && (n<= m-1) && (n'>= 0) && (n'<= m'-1)
    then
      Fix16 m (n+n'+1) (int32ToInt16(shift (int32ToInt32
        ((toInt x) * (toInt y)) (-m))))
    else
      error "Operation (+): GWL must be between
        1 and WL."
    False -> error"we must extend the WL of the operand whose
  ...
WL is smaller to the maximal value before multiplication

data Fixed32 = Fix32 Int Int32

class FixedPoint a where
  fix32toDouble :: a -> Double
  doubletoFix32 :: Int -> Double -> a

instance FixedPoint Fixed32 where
  fix32toDouble x = fix32toDouble' x
  doubletoFix32 m x = doubletoFix32' m x

doubletoFix32' m x = Fix32 m n (intToInteger (truncate (a * 2 ∧ (m - n - 1 + b))))
  where n = toInt (ceiling (logBase 2 (fromInteger (fst (properFraction (abs x)) + 1))))
  a = significand x
  b = exponent x

fix32toDouble' (Fix32 m n x) = (fromInt (toInt x)) / (2 ∧ (m - n - 1))

instance Show Fixed32 where
  showsPrec p (Fix32 m n x) = shows (fix32toDouble (Fix32 m n x)) .
  showString "$\text{'Bit'}$"

instance Read Fixed32 where
  readsPrec _ x = readsFixed32 x
readsFixed32 = [(doubletoFix32 m x, e) | (x,a) <- reads s,
  ("\text{'}\text{'},b) <- lex a,
  (m,c) <- reads b,
  (\text{"Bit"},d) <- lex c,
  (\text{"\text{'},e) <- lex d]

instance Eq Fixed32 where
  Fix32 m n x == Fix32 m' n' y = m == m' && n==n' && x == y

instance Num Fixed32 where
  (Fix32 m n x) + (Fix32 m' n' y) = case (m==m') of
    True -> if (n>=0) && (n' <= m-1) && (n' >0) && (n<= m'-1) then
      Fix32 m (i+1) ((shift x (n-i-1)) + (shift y (n'-i-1)))
    else
      error "Operation (+): WL must be between 0 and WL"
  False -> error "we must extend the WL of the operand whose WL is smaller to the maximal value before addition"

where i = max n n'

(Fix32 m n x) * (Fix32 m' n' y) = case (m==m') of
  True -> if (n>0) && (n' <= m-1) && (n' >0) && (n<= m'-1) then
Fix32 m (n+n’+1) (int64ToInt32(shift (intToInt64
((toInt x) * (toInt y)) (-m))))
else
   error "Operation (+): IWL must be between
1 and WL . "
False -> error"we must extend the WL of the operand whose
WL is smaller to the maximal value before multiplication"

**Test Code for Fixed-point Arithmetic in 2-order FIR**

```plaintext
FixedPoint> doubleFix16V 16 (-0.0046;>0.9954;>(-0.0046);>NullV)
<.00457764 (0 Bit),0.995392 (0 Bit),-0.00457764 (0 Bit)>

FixedPoint> doubleFix16S 16 (0.0-0.5-0.8-NullS)
{0.0 (0 Bit),0.5 (0 Bit),0.799988 (0 Bit)}

1st cycle
FixedPoint> (Fix16 16 0 (-150))*(Fix16 16 0 0 )
0.0 (1 Bit)

2nd cycle
FixedPoint> (Fix16 16 0 0 )* (Fix16 16 0 32617) + (Fix16 16 0 (-150))
* (Fix16 16 0 16384)
-0.00231934 (2 Bit)

3rd cycle
FixedPoint> (Fix16 16 0 16384)* (Fix16 16 0 32617) + (Fix16 16 0 (-150))
* (Fix16 16 0 26214)
0.433042 (2 Bit)

4th cycle
FixedPoint> ((Fix16 16 0 26214)* (Fix16 16 0 32617)+ (Fix16 16 0 16384)
* (Fix16 16 0 (-150))
0.793945 (2 Bit)

5th cycle
FixedPoint> (Fix16 16 0 26214)* (Fix16 16 0 (-150))
-0.00366211 (1 Bit)
```

### 5.2 The Clock Domain Library

**Description:** The module Clockdomain introduces clock domain interfaces for up-sampling and down-sampling which either use head(downHSY,upHSY) or tail (downTSY,upTSY) synchronization in clock domain refinement.

The sub-domain downHSY samples the input signal with a data rate \( r \) and produces an output signal with the data rate \( rn \). The \( i \)th \((i \in N_0)\) output event is the \((i * n)\)th event of the input signal.
The sub-domain downTSY samples the input signal with a data rate \( r \) and produces an output signal with the data rate \( rn \). The \( i(\forall i \in N_0) \) output event is the \(((i + 1)n - 1)\)th event of the input signal.

The sub-domain upHSY samples the input signal with a data rate \( r \) and produces an output signal with the data rate \( r/n \). Every input event is output followed by \( n \) delay at each output cycle.

The sub-domain upTSY samples the input signal with a data rate \( r \) and produces an output signal with the data rate \( r/n \). Every input event is output after \( n \) delay at each output cycle.

module Clockdomain where

import ForSyDeStdLib

downTSY :: Int -> Signal (AbstExt a) -> Signal (AbstExt a)
downTSY n NullS = NullS
downTSY n xs = (atS (n-1) xs) : (downHSY n (dropS (2*n-1) xs))

upTSY :: Int -> Signal (AbstExt a) -> Signal (AbstExt a)
upTSY n NullS = NullS
upTSY n (x:xs) = delaynSY Abst (n-1) (unitS x) ++ upTSY n xs

downHSY :: Int -> Signal (AbstExt a) -> Signal (AbstExt a)
downHSY n NullS = NullS
downHSY n xs = (headS xs) : (downHSY n (dropS n xs))

upHSY :: Int -> Signal (AbstExt a) -> Signal (AbstExt a)
upHSY n NullS = NullS
upHSY n (x:xs) = unitS x ++ takeS (n-1) (delaynSY Abst (n-1) NullS)
+++(upHSY n xs)

Test Code for Clock Domain Library

Clockdomain> downTSY 3 (Prist 1:-Prist 2:-Prist 3:-Prist 4:-Prist 5:-Prist 6:-
Prist 7:-NullS)
{3,6}

Clockdomain> downHSY 3 (Prist 1:-Prist 2:-Prist 3:-Prist 4:-Prist 5:-Prist 6:-
Prist 7:-NullS)
{1,4,7}

Clockdomain> upHSY 3 (Prist 1:-Prist 2:-Prist 3:-Prist 4:-Prist 5:-Prist 6:-
Prist 7:-NullS)
{1,...,2,...,3,...,4,...,5,...,6,...,7,...}

Clockdomain> upTSY 3 (Prist 1:-Prist 2:-Prist 3:-Prist 4:-Prist 5:-Prist 6:-
Prist 7:-NullS)
{,...,1,...,2,...,3,...,4,...,5,...,6,...,7}

Clockdomain> upTSY 2 (downTSY 2 (Abst:-Prist 1:-Abst:-Prist 2:-Abst:-Prist 3:-NullS))
5.3 The Resource Sharing Library

Description: The module Resourcesharing introduces sub-domain interfaces for parallel-serial and serial-parallel transformations to implement resource sharing refinement together with clock domain interface and FSM. We also defined the function rotV to reuse one function unit by vector rotation.

The function rotV rotates a vector to the right. 
The function rotV right rotates a vector with n times.
The function unitNS creates a signal with n inputs.
The function pFSM performs the corresponding operation (+, -, *, /) in n event cycles by a moore FSM.
The function pnS implements the parallel-serial transformation to produce a serial signal with n parallel inputs. The data rate of output is n times that of the input signals.
The function sPn implements the serial-parallel transformation to produce n parallel output signals with a serial input. The data rates of the outputs is 1/n times that of the input.

module Resourcesharing where
import ForSyDeStdLib
import Vector
import Clock

rotV n vs |n==0 = vs
           |otherwise = rotV (rotV (n-1) vs)

unitNS n x |n <= 0 = NullS
           |otherwise = x-(unitNS (n-1) x)

pnS:: [Signal a ]-> Signal (Signal a)
pnS (NullS : NullS) = NullS
pnS (x:- xs) (y:-ys) = ((mapSY headS (signal((x:-xs):xsxs))):-
                           (pnS (fromSignal(mapSY tail (signal ((x:-xs):xsxs))))))

tail NullS = NullS
tail (x:-xs) = xs

sPn:: Int -> Signal a -> [Signal a]
sPn n NullS = []
sPn n xs = fromSignal (takeS n ((downHSY n xs)):-
                           (sPn n (dropS 1 xs)))

resourceshareFSM k ls =mapSY (downTSY (lengthS (signal ls))) (mapSY
                          (pFSM k) (pnS ls))

pFSM k s = tailS(mooreSY (f k) g (0,1) ((->) s 0)) where

70
\[
\begin{align*}
f \ k \ x \ y &= \text{if } (\text{fst } x = = 0) \text{ then } \\
&\quad \begin{cases} 
(1, y) \\
\text{else if } ((\text{fst } x) < n \&\& ((\text{fst } x) > 0)) \text{ then } \\
((\text{fst } x) + 1, (\text{tran } (\text{snd } x) y)) \\
\text{else } (0, (\text{tran } (\text{snd } x) y))
\end{cases}
\end{align*}
\]

where \( n = (\text{length } s) + 1 \)

\[
\begin{align*}
\text{tran } k \ x \ y &= | k = = \'+' = x + y \\
&\quad | k = = \'-' = x - y \\
&\quad | k = = \'*' = x * y \\
&\quad | k = = \'/' = x / y
\end{align*}
\]

\[
\begin{align*}
g \ x &= \text{if } (\text{fst } x) = = n \text{ then } \\
&\quad \text{abstExt } (\text{snd } x) \\
&\quad \text{else } \\
&\quad \text{Abst}
\end{align*}
\]

where \( n = \text{length } s \)

\[
\begin{align*}
\text{parSer2SY} ::& \quad \text{Signal } a \rightarrow \text{Signal } a \rightarrow \text{Signal } a \\
\text{parSer2SY} \&\& \text{NullS NullS} &= \text{NullS} \\
\text{parSer2SY} \ (x : xs) \ (y : ys) &= x : y : \text{parSer2SY } xs \ ys
\end{align*}
\]

\[
\begin{align*}
\text{serPar2SY} ::& \quad \text{Signal } a \rightarrow (\text{Signal } a \ , \text{Signal } a) \\
\text{serPar2SY} \&\& \text{NullS} &= (\text{NullS} , \text{NullS}) \\
\text{serPar2SY} \ xs &= (\text{headS } xs: \text{fst}(\text{serPar2SY } (\text{dropS } 2 \ xs)), \\
&\quad (\text{AtS } 1 \ xs):\text{snd}(\text{serPar2SY } (\text{dropS } 2 \ xs)))
\end{align*}
\]

\[
\begin{align*}
\text{resourceshare2SY} \&\& \text{NullS NullS NullS NullS NullS} &= (\text{NullS} , \text{NullS}) \\
\text{resourceshare2SY} \ s_1 \ s_2 \ s_3 \ s_4 &= \text{serPar2SY } (\text{zipWithS } f \ (\text{parSer2SY } s_1 \ s_2) \ (\text{parSer2SY } s_3 \ s_4))
\end{align*}
\]

**Test Code for Resourceshare Library**

Resourceshare\> rotV 2 (1 : > 2 : > 3 : > 4 : > 5 : > NullV) 
<3,4,5,1,2>

(7 : - 8 : - NullS) 
(\{4,6\},\{12,14\})

Resourceshare\> pFSM '+ ' (1 : - 2 : - 3 : - 4 : - NullS) 
\{___10.0\}

Resourceshare\> pFSM '- ' (1 : - 2 : - 3 : - 4 : - NullS) 
\{___-8.0\}

Resourceshare\> pFSM '/ ' (1 : - 2 : - 3 : - 4 : - NullS) 
\{___0.0416667\}


71
{{16.0}, {20.0}}

Resourcesharing> snP 2 (1:-2:-3:-4:-NullS)
[1, 3], {2, 4}]

Resourcesharing> pnS [(1:-2:-6:-NullS), (3:-4:-1:-NullS), (5:-6:-2:-NullS)]
{{1, 3, 5}, {2, 4, 6}, {6, 1, 2}}

5.4 Case Study Code

5.4.1 Original Code

module Equalizer where
    import ForSyDeStdLib
    import AudioAnalyzer
    import AudioFilter

data AnalyzerMsg = Pass
    | Fail deriving(Show, Eq)

equalizer levels input = (output, disFlag) where
    output = audioFilter levels input
    disFlag = audioAnalyzer fitPts output

module AudioFilter where
    import EqualizerTypes
    import ForSyDeStdLib
    import FIR

audioFilter :: Signal (Double, Double) -> Signal Double -> Signal Double
audioFilter levels audioIn = audioOut
    where audioOut = zipWith3SY add3 bassPath middlePath treblePath
        bassPath = product (mapSY exp bass) ((fir lp) audioIn)
        middlePath = scale 1.0 ((fir bp) audioIn)
        treblePath = product (mapSY exp treble) ((fir hp) audioIn)
        scale k xs = mapSY (* k) xs
        product xs ys = zipWithSY (*) xs ys
        add3 a b c = a + b + c
        bass = mapSY fst levels
        treble = mapSY snd levels
lp = ((-0.0046):>0.9945:>-0.0046):>NullIV)
bp = ((-0.0046):>0.9945:>-0.0046):>NullIV)
hp = ((-0.0046):>0.9945:>-0.0046):>NullIV)

module FIR where

import ForSyDeStdLib
shiftreg k = scanlSY (shiftIV) (copyV k 0)

ipV NullV NullV = 0
ipV (x>xv) (y>yv) = x * y + (ipV xv yv)

fir h = mapSY (ipV h) . (shiftreg (lengthV h))

module AudioAnalyzer where
import FFT
import ForSyDeStdLib
import Complex

limit = 10.0

nLow = 4

audioAnalyzer :: Integer -> Signal Double -> Signal (AbstExt AnalyzerMsg)
audioAnalyzer fftLevel = mapSY (checkLimit limit) . mapSY sumVector .
mapSY logV . mapSY (selectLow nLow) . mapSY squareV . mapSY absV .
mapSY selectHalf . mapSY (fftT k) . (distributor (2^k)).mapSY fromDouble
where k = fftLevel

module FFT where
import ForSyDeStdLib
import Complex

fftT k Abst = Abst
fftT k (Prst vs) = Prst (fft k vs)

distributor n = groupSY n

selectHalf Abst = Abst
selectHalf (Prst (x:xv)) = Prst (takeV ((lengthV xs - 1) / 2) xs)

absV Abst = Abst
absV (Prst vs) = Prst (mapV magnitude vs)

squareV :: AbstExt (Vector Double) -> AbstExt (Vector Double)
squareV Abst = Abst

squareV (Prst vs) = Prst (mapV sqr vs)
where sqr x = x * x

selectLow n Abst = Abst
selectLow n (Prst xs) = Prst (takeV n xs)

logV Abst = Abst
logV (Prst vs) = Prst (mapV id vs)

sumVector Abst = Abst
sumVector (Prst vs) = Prst (foldlV (+) 0.0 vs)
checkLimit limit Abst = Abst
checkLimit limit (Prst x) | x > limit = Prst Fail
| x < limit = Prst Pass

fft k xs | n == 2 k = (bitrev . foldrV stage xs . iterateV k (* 2)) 2
where
  stage k = concatV . zipWithV (segment (takeV m twiddles) . groupV k
    where m = n 'div' k

segment twid = unduals . mapV (butterfly twid) . duals
twiddles = (bitrev . mapV (cis . negate) . halfcycle) n
where n = lengthV xs

duals :: Vector a -> Vector (a,a)
duals v = zipV (takeV k v) (dropV k v)
where k = lengthV v 'div' 2

unduals :: Vector (a,a) -> Vector a
unduals v = let
  (x, y) = unzipV v
  in
  x <+> y

butterfly :: Num a => a -> (a,a) -> (a,a)
butterfly w (x0, x1) = (x0 + t, x0 - t)
  where t = w * x1

bitrev :: Vector a -> Vector a
bitrev (v::NullV) = v::NullV
bitrev vs = bitrev (evens vs) <+>
  bitrev (odds vs)

evens NullV = NullV
evens (v1::NullV) = v1 :: NullV
evens (v1::v2::v) = v1 :: evens v

odds NullV = NullV
odds (v1::NullV) = NullV
odds (v1::v2::v) = v2 :: odds v

halfcycle :: Integer -> Vector Double
halfcycle n = halfcycle1 0 m n
  where m = fromInteger n / 2
halfcycle1 l m n
  | l == m = NullV
  | l /= m = -2*pi*l/(fromInteger n) :> halfcycle1 (l+1) m n

### 5.4.2 Refinement Code

module Refinedequalizer where
  import FFT
  import AudioFilter
import FixedPoint
import Clockdomain
import Resourcesharing

ipV 0 hv xv = headV hv * headV xv
ipV n hv xv \{(n< lengthV hv) = headV (rotV n hv) * headV (rotV n xv)
+ipV (n-1) hv xv
| otherwise=0

sipoSY m s = scanlSY shiftrV (copyV m 0) s

fir m h s = fix16DoubleSY (mapSY (ipV (lengthV h-1) (doubleFix16V m h)) (mapSY (doubleFix16V m) (sipoSY (lengthV h) xs)))
where xs= (s ++ (unitNS (lengthV h-1) 0)

doubleFix16V m v = mapV (doubletoFix16’ m) v
doubleFix16SY m s = mapSY (doubletoFix16’ m) s

fix16DoubleSY s = mapSY (fix16toDouble’) s

audioAnalyzer k s = upTSY (2k) (mapSY ((checkLimit limit) . sumVector . logV . (selectLow nLow) . squareV . absV . selectHalf . (fftT k)) (downTSY (2k) (distributor (2k) (mapSY fromDouble s))))

audioFilter :: Signal (Double, Double) -> Signal Double -> Signal Double
audioFilter levels audioIn = audioOut
where audioOut = zipWith3SY add3 bassPath middlePath treblePath
  bassPath = product (mapSY exp bass) (fir 16 lp audioIn)
middlePath = scale 1.0 (fir 16 bp audioIn)
treblePath = product (mapSY exp treble) (fir 16 hp audioIn)
  scale k xs = mapSY (* k) xs
  product xs ys = zipWithSY (*) xs ys
  add3 a b c = a + b + c
  bass = mapSY fst levels
treble = mapSY snd levels

lp = ((-0.0046):>0.9945>:(-0.0046):>NullV)
bp = ((-0.0046):>0.9945>:(-0.0046):>NullV)
hp = ((-0.0046):>0.9945>:(-0.0046):>NullV)

5.4.3 Test Code

1. Test Code of FIR
FIR> fir ((-0.0046):>0.9945>:(-0.0046):>NullV) (0:0.5:0.8:-NullS)
{0.0,-0.0023,0.49357,0.79333,-0.00368}

RefinedFIR> fir 16 ((-0.0046):>0.9945>:(-0.0046):>NullV) (0:0.5:0.8:-NullS)
{0.0,-0.00244141,0.493408,0.793213,-0.00366211}

2. Test Code of AudioFilter
AudioFilter > audioFilter ((0.0, 0.0):-(1.0, 1.0):-(2.0, 2.0):-(3.0, 3.0):-NullS) (0.0-.1.0:-2.0-.3.0:.4.0-.5.0-.6.0-.7.0-.8.0-.NullS)
{0.0-.0.029682,15.5462,81.1317}

RefinedAudioFilter > audioFilter ((0.0, 0.0):-(1.0, 1.0):-(2.0, 2.0):-(3.0, 3.0):-NullS) (0.0-.1.0-.2.0-.3.0-.4.0-.5.0-.6.0-.7.0-.8.0-.NullS)
{0.0-.0.0298571,15.5393,81.0555}

3. Test Code of AudioAnalyzer
AudioAnalyzer > audioAnalyzer 2 (0.0-.1.0-.2.0-.3.0-.4.0-.5.0-.6.0-.7.0-.8.0-.NullS)
{---Pass,---Pass}

RefinedAudioAnalyzer > audioAnalyzer 2 (0.0-.1.0-.2.0-.3.0-.4.0-.5.0-.6.0-.7.0-.8.0-.NullS)
{---Pass,---Pass}

4. Test Code of Equalizer
Equalizer > equalizer 2 ((0.0, 0.0):-(1.0, 1.0):-(2.0, 2.0):-(3.0, 3.0):-(4.0, 4.0):-(5.0, 5.0):-(6.0, 6.0):-(7.0, 7.0):-(8.0, 8.0):-(9.0, 9.0):-(10.0, 10.0):-NullS) (0.0-.5-.0.8-.0-.5-.0.8-.1.0-.2.0-.3.0-.4.0-.NullS)
{0.0-.0.0157143,7.78305,32.6574,0.672585,146.95,637.056,213.55,11739.5,47893.4,174581.0},
{---Fail,---Fail})

RefinedEqualizer > equalizer 2 ((0.0, 0.0):-(1.0, 1.0):-(2.0, 2.0):-(3.0, 3.0):-(4.0, 4.0):-(5.0, 5.0):-(6.0, 6.0):-(7.0, 7.0):-(8.0, 8.0):-(9.0, 9.0):-(10.0, 10.0):-NullS) (0-.5-.0.8-.0-.5-.0.8-.1.0-.2.0-.3.0-.4.0-.NullS)
{0.0-.0.0157143,7.78305,32.6574,0.672585,146.95,637.056,213.55,11739.5,47893.4,174581.0},
{---Fail,---Fail}
Bibliography


