System Level Specification in Lava

Satnam Singh
Xilinx Inc.
2100 Logic Drive, San Jose, CA95124, USA.
Satnam.Singh@xilinx.com

Abstract

The Lava system provides novel techniques for representing system level specifications which are supported by a design flow that maps Lava descriptions onto System-on-Chip platforms implemented on very large FPGAs. The key contribution of this paper is a type class based approach for specifying bus-based system configurations. This provides a very flexible and parameterised flow for combining predesigned IP blocks into a complete FPGA-based system.

1 Introduction

System level specification is currently performed using a variety of languages and ad hoc notations. Often these representations were not designed for the task of specification. For example, the C and C++ languages have many desirable features for generating efficient machine code but they are not amongst the most powerful and expressive languages for high level abstract descriptions. We examine how system level specifications can be improved by starting off with a language with more expressive and powerful constructs. For platform level descriptions we draw inspiration for techniques like Coral from IBM [1]. We try and achieve similar results within the framework of a language by exploiting type classes to capture system composition and interconnection at a suitably abstract level.

We concentrate on high level system descriptions which have the properties of (i) easy substitution of subblocks to perform design space exploration; (ii) abstraction of communication to allow experimentation with many different kinds of communication protocols; (iii) a link (not necessarily automatic) to a design flow that supports System-on-a-Chip style applications which are realised on platform FPGAs. We can also export our platform descriptions to external formal verification tools to help statically analyse properties about a particular system composition.

We place a great deal of emphasis on system level *specification* and *configuration*. Some system level specification techniques are geared towards automatic implementation of the specification via a technique like synthesis. This can often compromise the clarity of the original specification as one tries to cast one's thoughts in a way that can be sensibly synthesised by a particular vendor's tools. For the specification technique we present here we make no attempt to produce completely implementable specifications. However, we do provide support for taking high level partitioning information from our technique and exporting it to external system level tools to help with the task of platform tailoring e.g. how many processors and buses are required, how many arbiters, what kind of bus interface logic is required etc. The task of module level implementation is in the next part of the design flow. For modules that are to be implemented in the structural subset of Lava [2] there is a convenient link between the specification level interfaces and the implementation level interfaces.

2 Background

Many digital design projects start off with an informal natural language specification which is then used to write a C language model which acts as an executable specification. For specific application domains e.g. digital signal processing designers often use tools like MATLAB which are used to explore design alternatives. These domain specific representations have become so important that some vendors have started to offer design flows that start off with MATLAB/Simulink and automatically transform this high level specification all the way to a full digital hardware implementation.

SystemC [21] is quickly establishing itself as a executable system level specification notation which also has a path to implementation and like CONLAN [18] contains mechanisms for user extensibility. Similar approaches are exemplified by SpecC [23] and OpenJ [24] which draws on work by Odersky and Wadler [15] which combines object-oriented and functional descriptions.

3 Functional Descriptions

Many researchers have found functional notations and representations useful for describing system behaviour while abstracting system structure e.g. [4][6][7][9]. These observations are based on modelling real-world data-flow naturally using functional composition. In this paper we argue that functional notations have more to offer than that "data-flow plumbing" for system description but can also capture aspects of system architecture by exploiting of the rich lan-

1530-1591/03 $17.00 © 2003 IEEE

guage features available to the user of a modern functional language or notation e.g. type classes [22].

Keuzer et. al. [12] argue that the separation of communication from computation is essential for containing the complexity of system descriptions. We present a technique which adheres to this advice by representing system computation using higher order functions over infinite streams and system communication through a powerful type class system which encodes parameterised information about communication in the types of circuits.

One of the key features we use in Haskell for the system level Lava descriptions is called *type classes*. This feature provides a systematic way to define and resolve overloaded functions compared to the ad hoc overloading techniques used in languages like C++.

This paper focuses on the system representation of bus-based architectures expressed in the type class system of Lava. Since Lava is embedded in Haskell the examples given here are also valid Haskell programs. Although it is not possible to give a comprehensive description of programming in Haskell here we briefly outline some of the key concepts of type classes which is the main feature we use in this paper. The main use of the class system is to model system architecture rather than refine it to an implementation.

The type class descriptions given below are case sensitive and to some extent are very similar to the class concepts in languages like C++ and Java. A class is given a name and one or more type parameters and then identifies a list of overloaded functions which can use the type parameters to provide systematic overloading. The parameterised types in type classes is what makes the class system in Haskell distinctive and this is the key feature we exploit to help model aspects of bus-based system architecture.

A class is used by creating an instance by providing concrete values for the parameterised types. A class can be extended be using it as the base class for another class. This corresponds more closely to notions of *type extension* rather than inheritance although for the single inheritance case it appears very similar.

Here is an example of one of the type classes that are used extensively in the Lava system:

```
class Bitify a where
  toBit :: a -> [Bit]
  fromBit :: [Bit] -> a
```

This class defines two overloaded functions which, for a given type a can represent it as a list of bits, or take a list of bits and return the corresponding value with the type a. This class allows us to use arbitrarily sophisticated types in system level specifications with the knowledge that during later function or communication specification there is a way obtain a bit-level representation for the high level type.

For a user defined type an instance of the Bitify class is easily defined:

```
data Pixel = Intensity Int
                -- Asssumne a pixel has value 0..255

instance Bitify Pixel where
  toBit (Intensity value) = uint2bitvec value
  fromBit bits = Intensity (bitvec2uint bits)
```

The symbol "=" always means "is defined as" in Lava descriptions. This code fragment shows the declaration of a user-defined type for representing pixel intensities. The keyword **data** introduces a new type declaration. The Lava library functions uint2bitvec and bitvec2uint help to convert between bit-vectors and unsigned integers. We can define Bitify for other types and any particular use of toBit or fromBit is dispatched depending on the type of the argument. This is very similar to overloading in object-oriented languages.

4 JPEG 2000 Encoder Example

We now illustrate how system level specification and design space exploration is has been carried out for the modelling of a JPEG 2000 encoder application. The objective is to experiment with different configurations of a JPEG 2000 encoder using one or more soft processors, dedicated hardware accelerators for operations like the discrete wavelet transform, multiple SRAM interface controllers and varying amount of external memory, one or more system buses and various peripherals e.g. interfaces to video analogue to digital convertors, UART interfaces and Ethernet PHY interfaces. The target platform for the system is a Virtex™-II FPGA and the basic bus shall use is the On-Chip Peripheral Bus which is one of the buses which make up IBM's CoreConnect™ [8] bus standard. The bus is implemented in soft logic on the FPGA so we may create multiple instances of buses and connect them with bridges. The 32-bit soft processor is also realised in the regular FPGA fabric and we may also create as many instances of it as we require. The processing stages performed for JPEG 2000 encoding are shown in Figure 1. We would like to be able to describe such a data-flow in a high level representation which has the implementation of the discrete wavelet transform (DWT) suitably parameterised to allow easy design space exploration. In particular we want to explore hardware, software and mixed hardware/software implementation of the DWT function.

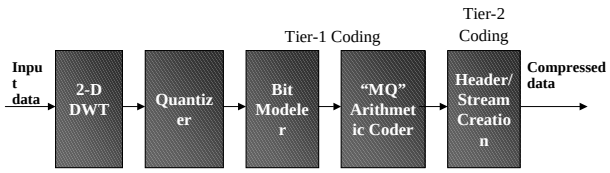

Fig. 1: Data Flow for the JPEG 2000 Encoder

The actual implementation of the JPEG 2000 system do not directly follow this data flow. The components communicate either by procedure calls and shared variables or using bus-based communication.

System level Lava helps with the description of platform and design space exploration through the use of type classes, higher order functions and polymorphism. We can even automatically extract a platform level description from our Lava specification and export it to vendor tools for elaboration and implementation onto an FPGA (assuming that implementations of all hardware blocks are available). This can give accurate information about the maximum speed of the system bus (or buses).

One of the main design space explorations we wish to make with the JPEG 2000 encoder is to decide what components should go into hardware and what components should go into software. To allow us to easily experiment with multiple implementations (hardware or software) we define type classes for each "module" that we are interested in analysing. For example, the discrete wavelet transform is one of the main sub-components of the JPEG 2000 system that we would like to analyse and it has a type class definition similar to:

```
class Bitify pixel => DWT m pixel where
  dwt :: m pixel -> m pixel
```

Here we use a special feature of type classes which allows us to use multi-parameter type classes. This affords us a very high level of abstraction. Here we can abstract two kinds of things:

1. The particular *data representation* denoted by the type variable pixels: all we require is that for whatever type is supplied we should know how to convert it into bits.
2. The *communication mechanism* for the input and output is also parameterised by the type t.

We can now refine many different kinds of implementations for the discrete wavelet transform which use different data representations and communication mechanisms. In one case we may still want to keep the exact pixel representation abstract but we may want to specify the communication mechanism as being infinite streams of values represented by lists. This is achieved by the following instance declaration:

```
instance Bitify pixel => DWT [] pixel
```

This instance declaration means that there is now a DWT function with the type:

```
dwt :: Bitify pixel =>  [pixel] -> [pixel]
```

which we are free to define. We can also fully specify the parameters to get a specific version of the DWT that operates on a concrete pixel type:

```
data Pixel = Intensity Int deriving (Eq, Show)

instance DWT [] Pixel
```

which now allows us to define a DWT operation over streams of pixel values:

```
dwt :: [Pixel] -> [Pixel]
```

By using suitable types for m in the instance declarations we can provide multiple definitions for the DWT operation at different levels of abstraction. We can mix levels of abstraction in a single expression and we can use different implementations for the communication channel.

Assuming we have available the Lava FIFO communication type. We can instantiate yet another DWT that communicates along unbounded FIFOs with the inputs and outputs appearing like infinite streams of values.

```
instance Bitify pixel => DWT FIFO pixel
```

The type of the DWT operation described above is:

```
dwt :: Bitify pixel => FIFO pixel -> FIFO pixel
```

We do not supply any special read or write operations to FIFOs. For many kinds of communication we assume a standard interface which allows us to substitute many different kinds of communication models. The model is simply taking infinite lists of value as inputs and returning infinite lists of results. This provides a simple but powerful way of abstracting from a particular communication channel realisation and is motivated by the higher order functions over finite streams view of hardware. Using the unbounded FIFOs shown above for the interfaces of all the components of the JPEG 2000 system we can implement a Kahn networks [11]. Variants of FIFO allow us to define static data-flow networks. By using *monadic* implementation for the type m we can represent state and accumulate statistics about the operation of a FIFO. Another use of a monad might be to write out values for debugging. We can easily define a dummy version of DWT that simply writes out the input intensity by instantiating it with the IO monad that is used to perform IO operations in Haskell.

```
instance DWT IO Pixel where
  dwt pixel = do p <- pixel
                 putStrLn ("Pixel = " ++ show p)
                 return p

v :: IO Pixel
v = dwt (return (Intensity 23))
```

If we now evaluate the v function in a Lava/Haskell interpreter we get the expected result:

```
Lava> v
Pixel = Intensity 23
```

Many different kinds of implementation for the DWT can be written. Some can be purely behavioral whilst others can be at the timed functional or untimed functional level or transactional level. Exactly what kind of implementation is being used for a specific application of DWT is evident from its type.

For the design space exploration we tried different kinds of implementation for the DWT operation each of which have different concrete types but all of which are instances of the general type for DWT operation. For example, for the 2D DWT case we can represent the input as a list of lists where each list represents a line of the image. For the 1D DWT case we can represent the input as a list of values which represents a continuous image signal. Then software can be used to apply the 1D DWT twice to obtain a 2D DWT.

Using type classes in this way we can compose the entire JPEG 2000 system which can be represented at mixed levels of abstraction using many different models of communication. At some point we will end up specifying the type of bus to be used and properties for various components and peripherals. This information can be automatically exported to external vendor tools to create the specified platform and implement it on a specific FPGA and then perform timing analysis to check the speed of the system buses and the hardware sub-components as well as making sure components like digital clock managers (DCMs) will function properly.

Memory mapped system components can be specified in Lava using a class that defines the properties of a simple peripheral:

```
class Peripheral periph where
  isMaster :: periph -> Bool
  baseAddress :: periph -> Int
  addressSize :: periph -> Int
```

This can be used to define the properties of a specific peripheral which can be of any time with the appropriate signals.

```
data SimpleUART = SimpleUART {rx, tx, clk :: Bool}

instance Peripheral SimpleUART where
  isMaster uart = False
  baseAddress uart = 0xffff0800
  addressSize uart = 0x100
```

A specific configuration of the JPEG 2000 encoder which uses a single 2D DWT hardware engine has been implemented on a XC2V1000 FPGA. This is produced from a collection of system components described in system level Lava although the existence of the software for the soft processor and the peripherals, buses, and hardware DWT engine is assumed. It is important to note that the constituent components for this example were not designed in RTL-level Lava (although in principle they could have been). Instead, system level Lava has been used to model the interfaces of these pre-designed components and provides the basic infrastructure for composing these components together to form a complete system.

The benefit of the system level Lava description is that the whole system configuration can be expressed in a high level exploiting the powerful features of type classes and that this is the same language that the RTL blocks are specified in so there is no need to try and maintain consistency between two different notations for representing module functionality and system configuration. Furthermore, the single system specification remains executable. The output of system level Lava is a series of configuration files that are used to drive the vendor back end tools which then perform the tasks of elaborating the soft processors, peripherals etc. and composes them together. In the case of the JPEG 2000 encoder the Lava system generated input files for Xilinx's Embedded Developer Kit in the form of MHS files and MSS files which describe the details of the hardware configuration and software device drivers. These files, along with the actual implementation netlists for the base components and run-time system are then combined by Xilinx's vendor tools to produce a final implementation bitstream.

5 Related Work

Functional languages have been used extensively for the RTL level description of hardware, especially for data-flow style descriptions. However, their use for system level description and analysis is less developed. Examples of system level executable specifications include Launchbury and Matthews work on the Hawk [5] system which has been successfully used to model the behaviour of modern pipelined microprocessors through the notion of *transactions*.

Jantsch and Sander argue for the integration of the functional and object-orientated techniques for system

level specification [10]. They argue that functional techniques should only be used for the "functional design space exploration" and that object-orientated representations should be used only for architectural exploration. Here we try and blur the distinction by stretching the capabilities of the functional type classes system to provide capabilities similar to those found in object-orientated systems. This experiment can be taken further by using a small extension to the Haskell [16] language by Hughes and Spraud called Haskell++ which provides even more direct representation of object-oriented representations. A formal link between functional and object-oriented notations has already been shown by [17] who demonstrate how an object-oriented language can be translated into a lambda calculus with higher-order functions and subtyping.

Recently the ForSeDe [20] system, which like system level Lava is implemented in Haskell, has been proposed to assist with semantic preserving transformations and for support design designs.

The underlying semantics for our descriptions of based on recurrence equations over streams as described by Kloos [13]. Reekie [19] has used Haskell to represent digital signal processing circuits using higher order functions over infinite streams. Li and Leeser [14] have developed HML which exploits powerful features of the strict functional language ML for expressing structural circuit descriptions.

6 Conclusions

The system level designer has a wide array of design notations and languages to choose from for the task of system specification. Some, like SystemC, are enjoying a high level of endorsement from vendors and propose an almost entirely automatic flow from system description or specification to system implementation. We argue that although such approaches are appropriate for many kinds of systems the system architect should have a choice of notations and sometimes these notations may be executable and perhaps not directly automatically implementable. We have presented an example of a system level specification technique which makes for lack of complete automatic implementation (i.e. no synthesis) through power abstractions for system level specification and configuration. Furthermore, these descriptions can be used to create SoC platforms comprising of high level components, buses, arbiters etc. and then the module level components can be specified and implemented using conventional techniques or advanced tools for managing the configuration os SoCs like Coral.

"Virtex" and "Virtex-II" are trademarks of Xilinx Inc.

References

[1] Reinaldo A. Bergamaschi, William R. Lee, "Designing Systems-on-Chip Using Cores", DAC 2000.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell", International Conference on Functional Programming, 1998.

[3] Per Bjuréus and Axel Jantsch, "MASCOT: A specification and cosimulation method integrating data and control flow", Proc. of the Design, Automation and Test in Europe Conference, pages 161-168, Paris, France, March 2000.

[4] J. P. Calvez, "Embedded Real-Time Systems", J. Wiley and Sons, 1993.

[5] Byron Cook, John Launchbury, and John Matthews, "Specifying superscalar microprocessors in Hawk", 1998 Workshop on Formal Techniques for Hardware, Marstrand, Sweden. 1998.

[6] T. DeMarco, "Structured Analysis and System Specification", Yourdon Inc., New York, 1978.

[7] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specification and Design of Embedded Systems", Prentice Hall, 1994.

[8] IBM, "The CoreConnect™ Bus Architecture", http://www.chips.ibm.com/product/coreconnect/docscrcon_wp.pdf, 1999.

[9] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, "Object Oriented Software Engineering: A Use Case Driven Approach", Addison Wesley, Reading, Massachusetts, 1992.

[10] Axel Jantsch and Ingo Sander, "On the roles of functions and objects in system specificatio", Proc. of the International Workshop on Hardware/Software Codesign, 2000.

[11] G. Kahn, "Coroutines and networks of parallel processes", Information Processing, North-Holland Publishing Company, 1977.

[12] K. Keuzer, S. Malik, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonolization of concerns and platform-based design", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 19(12):1523–1543, December 2000.

[13] Carlos Delgado Kloos, "Semantics of Digital Circuits:, Lecture Notes in Computer Science Vol. 285 Springer 1987,

[14] Y. Li and M. Leeser, "HML, a novel hardware description language and its translation to VHDL", IEEE Transactions on VLSI, 8(1):1–8, February 2000.

[15] M. Odersky, P.Wadler, "Pizza into Java: Translating Theory into Practice", Proceedings of 24th ACM Symposium on Principles of Programming Languages, Paris, France, January, 1997.

[16] Simon Peyton Jones et. al. "Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language", Available from http://haskell.org. February 1999.

[17] Benjamin C. Pierce and David N. Turner, “Simple type-theoretic foundations for object-oriented programming”, Journal of Functional Programming, 4(2):207, April 1994.

[18] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly, “CONLAN - A Formal Construction Method for Hardware Desceiption Languages: Basic Principles”, National Computer Conference, Vol. 49, Anaheim, 1980.

[19] H. J. Reekie, “Realtime Signal Processing”, PhD thesis, University of Technology at Sydney, Australia, 1995.

[20] Ingo Sander and Axel Jantsch, “Transformation Based Communication and Clock Domain Refinement for System Design”, DAC 2002 June 10-14, 2002.

[21] SystemC. http://www.systemc.org. 2002.

[22] P. Wadler and S. Blott, “How to make ad hoc polymorphism less ad hoc.”, In Proceedings 1989 Symposium Principles of Programming Languages, pages 60, Austin, Texas, 1989.

[23] Jianwen Zhu, Rainer Dömer, Daniel D. Gajski, “Syntax and Semantics of the SpecC Language”, Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies 1997, Osaka, Japan, December 1997.

[24] Jianwen Zhu and Daniel D. Gajski, "OpenJ: An Extensible System Level Design Language," Proceedings of Design Automation and Test Conference in Europe, Munich, Germany, March, 1999.