Reusable Component IP Design using Refinement-based Design Environment

Sanggyu Park, Sangyong Yoon, and Soo-Ik Chae

Center for SoC Design Technology and
School of Electrical Engineering and Computer Science
Seoul National University,
Seoul 151-742, KOREA
email : {sanggyu, syyoon, chae}@sdgroup.snu.ac.kr

Abstract - We propose a method of enhancing the reusability of the component IPs by separating communication and computation for a system function. In this approach, we assume that the component designers describe mainly the computation part of the component, and the system designer can construct the communication part by using our refinement-based design environment. Moreover, we introduced a concept of the Communication Architecture Template Tree (CATree), which helps IP designers to effectively separate computation and communication for a system function. We confirmed that this approach is effective by applying it to a H.264 decoder design.

I. Introduction

Reuse-centric design methodologies including IP-based design and platform-based design have been widely accepted in the SoC industry . Its effectiveness is determined mainly with the richness of the component IPs and their reusability. We can enhance the component reusability substantially by using the standard bus interfaces [1] such as AMBA and Core Connect and generic memory interfaces such as an on-chip SRAM interface.

Although integrating the component IPs with standard interfaces is easier, there are still several limitations. First, connecting components with different protocols incurs considerable overheads. Second, standard interfaces limit the internal architecture of the component. Furthermore, the standard interfaces limit the system-level communication architecture. Consequently, there have been strong attentions on using flexible communication interfaces [2,4-6].

In this paper, we propose a method to enhance the reusability of component IPs by exploiting the concept of orthogonalization. We re-defined the roles of the component and system designers. The component designers should capture mainly the computation part of a function and its test model with only essential architectural hints for the communication refinement and provide a component IP package, which is described in details in Section IV, to the system designers. The system designers should configure only the communication part of the component IP to make it best fit to the system with our refinement-based design environment where we introduced a concept of the Communication Architecture Template Tree (CATtree).

The CATtree provides communication architecture templates to the system designers so that they can refine the communication part of the component for a system function before integrating it to the system. Therefore, the component designers just model the computation part of the component IP without worrying about the refinement of its communication part.

The rest of this paper is organized as follows. In Section II we review the previous works on reusable component designs and refinement-based design methodologies. We introduce the refinement-based design environment and explain the proposed component IP package in Sections III and IV, respectively. After describing a H.264 VLD component design example in Section V. we summarize our contributions and future works in Section VI.

II. Related Works

Several methods that provide more flexibility in the communication interface have been proposed. These component design methods can be summarized into two approaches: standard interface-based and abstract interface-based.

A. Standard Interface-based Component Design Approach

There are several popular standard interfaces for on-chip buses and memories, which are main primitives for the system integration. Among the various on-chip bus interfaces, AMBA is the most popular bus interface, which defines transaction functions, protocols and RT-level signals.

The example shown in Figure 1 is a design environment for a standard interface-based component that has three functions F_A, F_B and $F_{C,}$ which communicate with peer functions A_F, B_F, and C_F, respectively, in a system with standard interfaces. Although its intended functions were initially only three, the final component IP contains eight functions including two bus interface, two buffer, one on-chip memory interface. Although integrating a component IP to the system is relatively easy, designing the component IP itself is more complicated. Especially, it is difficult and time-consuming to design and verify the bus interface logic. Moreover, in designing a component IP with a standard interface, the component designer should make several architectural decisions, for example,

- Bus interface standard: AMBA or CoreConnect
- Memory type: On-chip memory, External memory
- # of bus interfaces and on-chip memory interfaces
- Bus interface types: Bus master or Bus slave
- Existence of internal buffers and those sizes

0-7803-9451-8/06/$20.00 ©2006 IEEE.

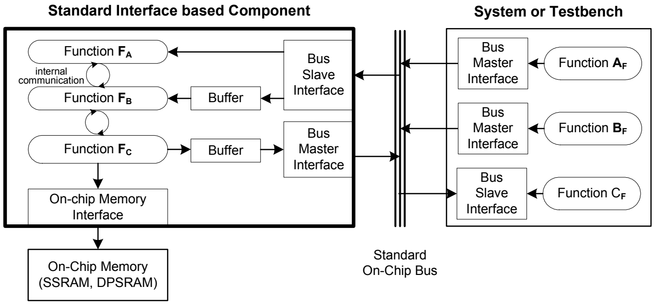


Figure 1. Standard Interface-based Component Design

To build an optimized system, these architectural decisions should be made not by the component designer, but by the system designers. Although this approach eases system-level hardware integration, it limits the system level design space.

B. Abstract Interface-based Component Design Approach

Exploiting the concept of orthogonalitzation can enhance the reusability of the component IPs. There are two types of orthogonalization: the separation of a function and its architecture and the separation of computation and communication for a function. Many researchers have made good contribution, which can be summarized in three stages: (1) finding a good abstraction of various interfaces, (2) generating wrappers efficiently and automatically, and (3) refining the architecture of system functional models

In the stage (1), VCI [4] and OCP-IP [5] defined generic protocol between the internal core part of a component IP and its bus wrapper. In figure 1, VCI and OCP-IP can be the interface between F_A and its bus slave wrapper. The bus interface controller, internal buffers and on-chip memory controllers are the wrappers in this category.

In the stage (2), methods for wrapper generation or synthesis were studied. Y. Hwang et. al. [2] proposed a method for generating communication wrappers from the timing diagram. They showed that synthesized wrappers are more efficient in terms of delay and area comparing to the generic wrappers and bridges.

In the stage (3), the refinement-based system design is an important issue in the design automation. F. Gharsalli, A. Jerraya et. al. [7,8] proposed an MPSoC design methodology. The computation part of the function is captured as a virtual component (VC), which is mapped onto architectural components: processors, memories and ASIC IP cores. Then, the architectural components are integrated with generic wrappers. S. Abdi, D. Shin, D. Gajski [9] proposed a communication synthesis tool, which is based on their own refinement-base design environment where they capture the communication function as a channel and refine it using a protocol library, which is a template set for the channel implementation.

III. Refinement-based Design Environment

The refinement-based design approach is a top-down system-level design methodology, in which the system level functions are first captured at a higher abstraction level. A system function should be divided into computation and communication functions. Hereafter, a communication function is called as a channel. After each system-level function is captured, each of its computation functions or channels is refined into a more concrete implementation step by step by making a certain architectural decision such as HW-SW partitioning, types of processors, types and sizes of on-chip memories, and the number of ASIC cores or embedded FPGAs.

Computation functions and channels of a system-level function should be captured separately with the hints from the refinement-based design environment. Because the channels such as FIFOs, arrays, and buses are commonly used in the system modeling, we provide them as a primitive library. The functional model shown in Figure 2 is for the component design example described in Section II, which includes six computation functions, three FIFO channels and a memory channel. Each computation function and channel is connected to the others through abstract interfaces regardless of their architectures or implementations.

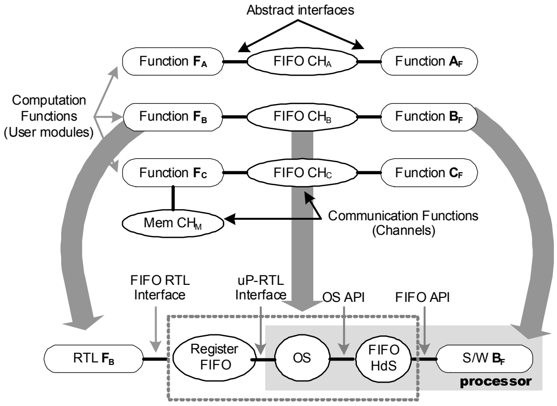


Figure 2. System Function Model and Partial Refinement

In our refinement-based design flow, a system-level function model is integration of user-defined computation functions with selected channels. A system implementation process can be seen as channel template replacement and high-level synthesis (or manual re-description) of the computation functions to a lower abstraction level. Our refinement-based design environment provides the following three types of hints to make modeling and refinement easy:

(1) hints for primitive channels that are the most frequently used for communication.

(2) hints for abstract interfaces for function modeling, RTL and S/W implementation.

(3) hints for architecture templates for refining the primitive channels.

These hints are represented with communication architecture template trees (CATtrees). Next, we explain primitive channels, abstract interfaces and channel templates before explaining the CATtrees

A. Primitive Channels

Models of computation like Kahn Process Network

(KPN) and Synchronous Dataflow Network (SDF) provide a well-defined set of abstract communication functions or channels. Although an abstract FIFO, which is the fundamental communication function of KPN and SDF, can model any point-to-point communication, it is too abstract to model an application function with the abstract FIFO. It is not easy to refine a MoC-modeled system function with it. Therefore, we replace it into the following four types of primitive channels, which are more concrete and widely used in the real application function modeling and system implementation.

(1) FIFO channels

A FIFO channel has two primitive functions: a blocking read and a blocking write, which are sufficient for capturing a high-level FIFO function. For the step-by-step architecture refinement, however, we additionally defined four additional functions such as peek, clear, more and sync. A peek function just checks the data without data retrieval, which is useful for H/W RTL implementation. A clear function initializes the FIFO channels. A more function returns true if there is more data to be written, which is useful to determine when to destroy a dynamically allocated channel. A sync function returns true if the written data is transferred to the reading counterpart, which is relevant when the refined architecture of a FIFO channel has intermediate buffers. The FIFO channel provides two abstract interfaces: abs_fifo_write interface and abs_fifo_read interface.

(2) Variable channels

A variable channel provides read and write functions, which do not have data synchronization. This channel is very useful in modeling memories updated infrequently. This channel provides abs_var_read and abs_var_write interfaces.

(3) Array channels

An array channel is a set of variable channels that is pointed by an index. Its primitive functions include index read and write functions. Although the array and variable channels are not required in the most MoCs, they are very useful in modeling and refining architecture. An array channel is an abstraction of memories including on-chip and external memories. Therefore, it is not easy to write a RTL model without using them in many cases. This channel provides abs_array_read and abs_array_write interfaces.

(4) Bus channels

A bus channel can perform point-to-point data transfers through a shared medium. In the top-down approach, a bus channel is not necessary in the function modeling. In the bottom-up approach, however, the bus channels are the most important communication pattern to connect components. In our refinement-based design environment, the bus channel is not used in function modeling but many partially refined (PR) channels, which will be explained later, have bus interfaces. For example, the PR channels and adapters related to the S/W have bus interfaces because RISC processors have a bus master interface.

B. *Abstract Interfaces*

In the system function model, an abstract interface is a boundary between a computation function and a channel. For computation of a function, its abstract interfaces mean primitive functions that it can use. For communication of a function, its abstract interfaces mean primitive functions that it must implement. Therefore, the abstract interfaces are a key to separate computation and communication for a function. An abstract interface is active for computation while it is passive for communication. An active interface of a computation model is connected to a passive interface of the channel. An abstract interface can be refined into three concrete interfaces: a TLM one for function capture and transaction level simulation and a RTL one for RT-level implementation, and a SW API one for S/W implementation.

C. *Partially Refined Channels (PR channels)*

Partially refined (PR) channels are used to refine primitive channels such as FIFOs, variables and arrays by making certain architectural decisions.

(1) Bus-FIFO Channels

A bus-FIFO is two FIFO channels that are connected using a bus channel, which can be refined to a Bus Master Write FIFO (top) and a Bus Master Read FIFO (bottom) as depicted in Figure 3(a). Although the internal architecture of a bus-FIFO is complex, its function is the same with that of the abstract FIFO channel.

(2) Cached Array Channels

A cached array channel is an array channel that contains a cache. An array channel is often refined into an external memory. However, an external memory has long latency and limited bandwidth. Therefore, an array channel can first be refined to a cached array channel that is connected to an external memory as depicted in Figure 3(b).

(3) Bus-Memory Channels

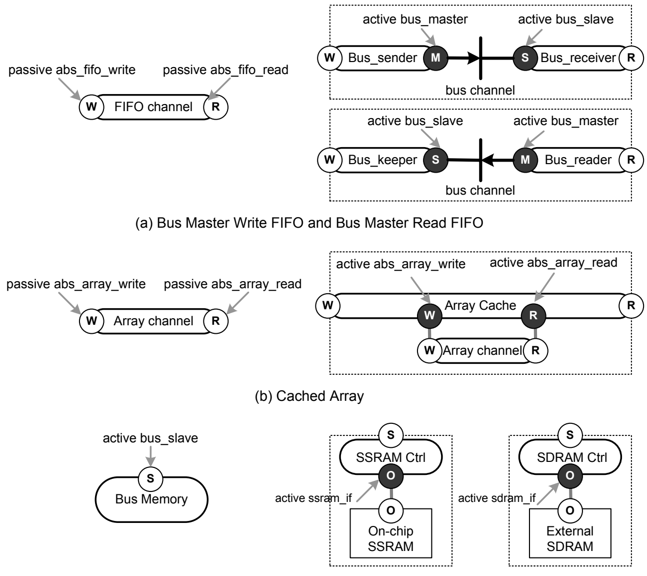
Some memories are shared with many computation functions. A bus-memory is an abstraction of shared memories that has a bus slave interface. A bus-memory channel can be refined to either an on-chip memory (left) or external SDRAM memory (right) as depicted in Figure 3(c).

(4) Channel Adapters

There are two types of channel adapters, which are interface and abstraction adapters. An interface adapter is used to connect two different types of channels. For example, a Bus Master Read FIFO channel shown in Figure 3(a) contains a bus_fifo_sender channel and a bus_fifo_receiver channel. These two PR channels are interface adapters. They adapt a FIFO channel to a bus channel. An abstraction adapter connects two channels in different abstraction levels. For example, a TLM-to-RTL adapter and a RTL-to-TLM adapter can connect a transaction-level model of computation to a RTL model of channel for a function or vice versa.

D. Channel Templates

Channel templates are configurable implementations of the primitive and PR channels, which are actually parameterized source files or generators. There are three templates for each channel: TLM, RTL, and S/W ones. A channel template has template parameters. To instantiate a channel instance, the value of the template parameters must be determined.



(a) Bus Master Write FIFO and Bus Master Read FIFO

(b) Cached Array

(c) Bus Memory

Figure 3. Partially refined channels

E. Communication Architecture Template Trees

A communication architecture template tree (CATtree) for an abstract channel is the collection of a primitive channel, an abstract interface and a set of templates for the abstract channel. We named CATtree because the architectures are expanded like a tree as shown in Figure 4.

Representing the architectural information for an abstract channel with its CATtree is an effective and integrated way of realizing the orthogonalization of function and architecture for communication. Providing a set of CATtrees can clearly expose what are communication functions and what are not for the designer. Once a communication function for a system function is captured with the CATtree for an abstract channel, refinement of that channel is guaranteed by its corresponding templates.

Computations for a system function are modeled with the TLM APIs in the CATtree and they are integrated to build its system function model. This system function model is the starting point of further implementation. Channels in the system function model are refined by replacing it with templates of the CATtree. Computations can be refined to RTL or S/W by high-level synthesis tools or by hand. Each computation and communication can be refined independently exploiting the abstract interfaces and adapters. Note that we do not cover the refinement of computation in this paper.

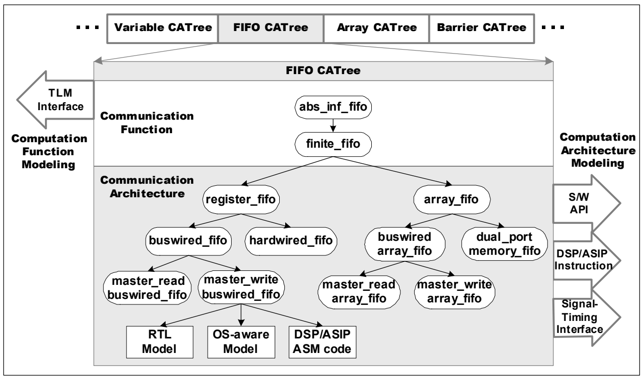


Figure 4. Communication Architecture Template Tree

IV. Reusable Component IP Design

Refinement-based design is an alternative to the reuse-centric design methodology. However, the platform-based design methodology is still attractive to the design teams who have systems that have been pre-integrated and verified. Therefore, reusable components are still important matters for them. However, conventional component IPs, which are designed to have standard interfaces, have limited reusability. Therefore, we introduce a new method of component IP design and delivery that utilizes our refinement-based design environment.

In this method, a component designer captures the component function using the TLM interfaces in the CATtrees. At the same time, he or she should model its test drivers. The test drivers generate stimulus to the captured component function and validate its response from the captured function model. By connecting the component function and its test-drivers with the TLM templates in the CATtrees, the component design obtains a testbench for validation, verification and interface refinement of the component IP. CATtree-based function capture of the component IP is relatively easier than bus-aware function modeling because he or she can concentrate on the computation and ignore communication related details.

The component designer can describe a RTL model of the computation function with the RTL interfaces. An abstraction adapter is inserted when a RTL computation function is connected with transaction-level channels and test drivers. Most of the designers have difficulty in designing a component because of the bugs related to its interfaces. In our approach, the designer can ignore the details about communication, which greatly reduces the complexity of the RTL design. For a computation function, we define the set of its transaction-level model, RTL model, and testbenches as the augmented deliverable package of its reusable component IP.

System designers can configure the communication part of the component IPs with our refinement-based design environment. The system designers replace an abstract channel with a more concrete one by using the CATtrees. Because they replace the channels instead of adapting the interfaces, the communication parts of the test drivers are also refined together. Therefore, without any manual modification of the testbenches, the system designers can execute them to estimate the performance of the component IP and to verify the refined model at each refinement step.

V. An H.264 VLD Component Design

We designed a new H.264 decoder system by reusing a system design shown in Figure 5. We decided to design and integrate a new dedicated hardware IP for the VLD operation. We followed the platform-based design methodology, utilizing the proposed method to design a more reusable VLD component IP as follows.

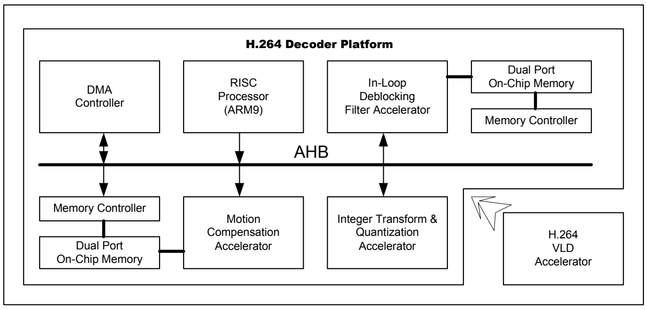


Figure 5. H.264 decoder SoC Platform

A. Reusable H.264 VLD Component IP design

A component designer designed an H.264 VLD component IP with the design flow we described. First, he captured its computation function and developed its test drivers, which took about a. week (Figure 6). The computation consists of a VLD core, an nC calculation block and a NAL decoder, which has 4 FIFO interfaces and 1 array interface. The VLD core decodes Exp-Golomb and CAVLC codes. The nC calculation block determines which mapping table is used in the VLD core to decode a CAVLC code. The NAL decoder eliminates the emulation prevention codes in the bit-stream.

After modeling the VLD function, he manually described its RTL model in HDL with the RTL interfaces of the CATtrees. The RTL model can decode an Exp-Golomb code in 1 cycle, a 2x2 chroma DC in 10 cycles (average) and 4x4 residual decoding in 30 cycles (average). We synthesized the RTL model using the Synopsys DesignCompiler^{TM} and the estimated gate count was 8277 gates at 5 ns delay in 0.18 um process technology. He finished both description and verification of the VLD RTL model in two days. It was relatively faster than its function modeling because of the two reasons. First, we described only a computation part in details. In general, the communication part is more complex and error-prone than the computation part. In our approach, however, potential communication errors are eliminated by exploiting the simple abstract interfaces of the CATtrees. Second, in the verification of RTL description we reused all the test models used for function modeling.

B. Communication Refinement and System Integration

A system designer configures the communication part of the H.264 VLD computation and integrates it to an existing platform. Here we present two implementations of the VLD IP in the H.264 decoder for QCIF (176*144) and HDV (1280*768) images, respectively.

(1) VLD decoding for QCIF images

In decoding QCIF images of 15fps, both computation and communication loads are low. The system designer decided to configure the VLD computation to have only an AHB slave interface for easier system integration. Because Flexible Macroblock Ordering (FMO) in the H.264 baseline profile, he also decided that the array channel connected to nC calculation sub-block has only 44 indices. In this configuration, the memory size is 220 bits and he finally decided to refine the array channel into a register array. Figure 7(a) shows refinement steps. The total gate count of the communication part was 12,150 gates

(2) VLD decoding for HDV images

In decoding HDV images of 30 fps, both computation and communication loads are very high. Additionally the H.264 decoder must support the FMO feature. Because the FMO feature requires all nC values of a frame, $\mathrm{ARRAY}_{\mathrm{nC}}$ must have at least 92160 indices, which is too big to be implemented with registers or on-chip memories. Thus, he decided to configure the H.264 VLD computation with the following five refinements:

- $\mathrm{FIFO}_{\mathrm{CMD}}$ to a 8 depth Bus-Master Write channel and refine $\mathrm{FIFO}_{\mathrm{VLD}}$ to a 8 depth Bus-Master Read channel
- a bus slave interface is shared by $\mathrm{FIFO}_{\mathrm{CMD}}$ and $\mathrm{FIFO}_{\mathrm{VLD}}$.
- $\mathrm{FIFO}_{\mathrm{ITQ}}$ to a dedicated register FIFO.
- $\mathrm{FIFO}_{\mathrm{STRM}}$ to a memory FIFO that have a cache with 16 registers of 8-bit width
- $\mathrm{ARRAY}_{\mathrm{nC}}$ to an external memory with cache.

Figure 7(b) shows the refinement steps according to the decisions listed above. The total gate count of the communication part was 10,559 gates.

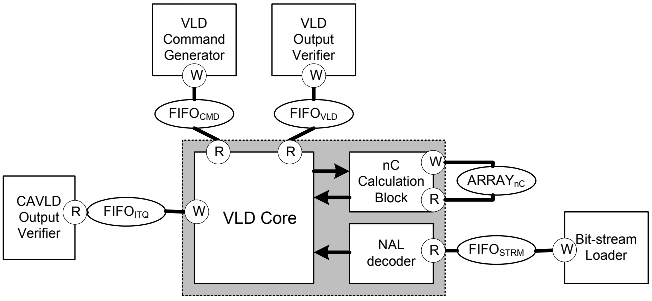
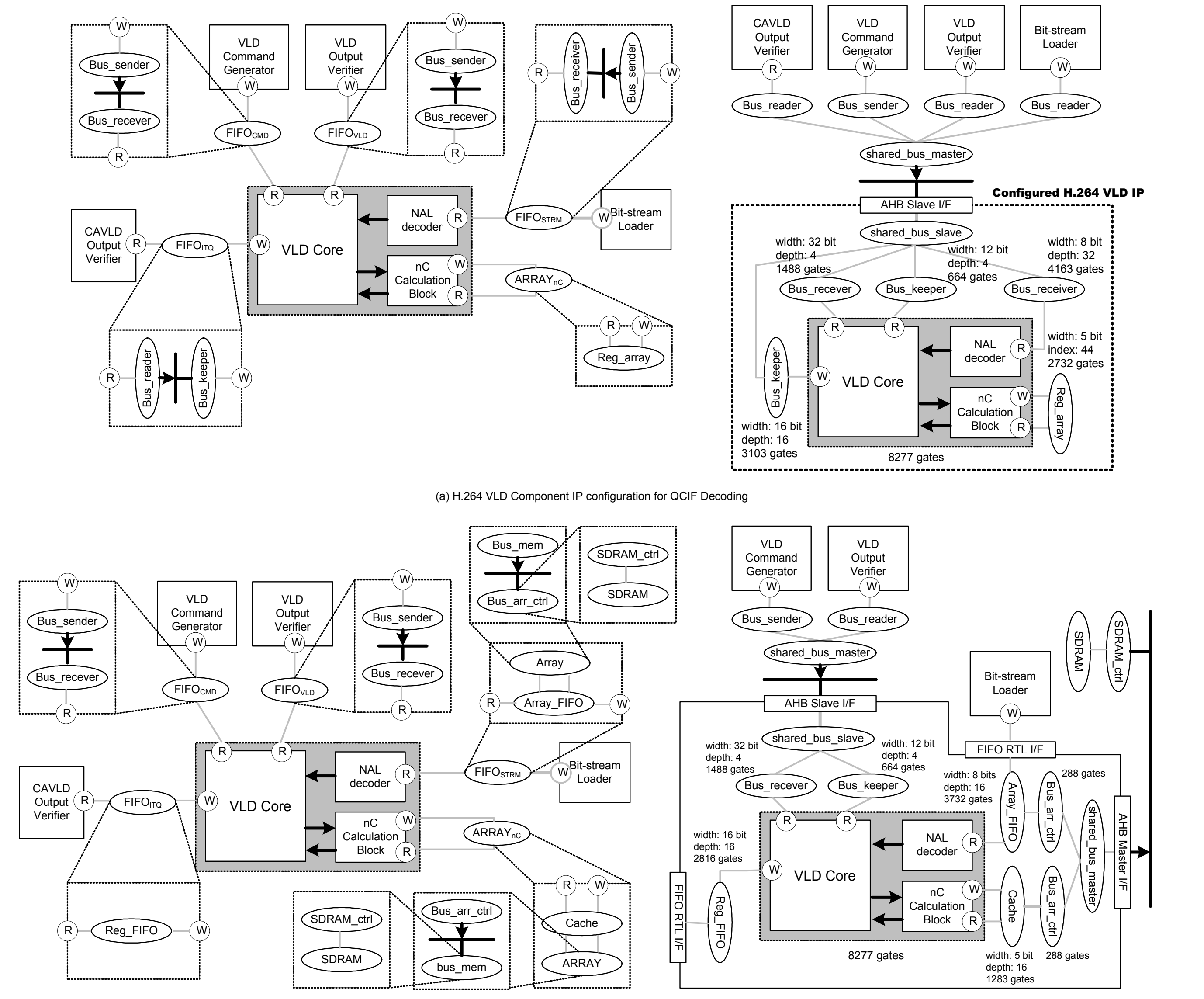


Figure 6. H.264 VLD Component IP

VI. Conclusions

We proposed an effective design approach to enhancing the reusability of component IPs. In this approach, we re-defined the roles of component and system designers. In designing a component IP, a component designer captures its computation and its communication in an abstract way while a system designer refines its communication part with our refinement-based design approach. We also proposed an augmented component IP deliverable package according to the redefined roles of the component and system designers.



(a) H.264 VLD Component IP configuration for QCIF Decoding

(b) H.264 VLD Component IP configuration for HDV Decoding

Figure 7. H.264 VLD Component IP Configuration Example

Our refinement-based design environment introduced a concept of the Communication Architecture Template Tree (CATtree), which can clearly separate computation function and communication architecture. Moreover, it provides generic interfaces to model both of them. After designing a VLD component for H.264, we found that the concept of the CATtree is effective for the communication refinement.

With this proposed approach, the component designers can design computation functions easily and the system designer can explore a larger design space with the help of the CATtree.We are now developing an H.264 decoder system with our design environment to exploit the full power of the CATtree. We will cover the refinement of computation later in another paper.

References

[1] Michael Keating, Pierre Bricaud, “Reuse methodology manual”, Kluwer academic publisher, 2002

[2] Y.-T. Hwang and S.-C. Lin, “Automatic protocol translation and template based interface synthesis for IP reuse in SoC”, *Proc of 10th ASP-DAC*, pp. 565-568, Dec. 2004

[3] K. Keutzer, S. Malik, et. al., “System-level design: Orthogonalization of concerns and platform-based design”, *Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol19, No. 12, pp. 1523-1542, Dec. 2000.

[4] VSI Alliance On-Chip Bus Development Working Group,“OCB 2.0”, Apr. 2001

[5] Sonics Inc. “Open core protocol specification 1.0”, 2000

[6] W. Cesario,D. Lyonnard,G. Nicolescu,Y. Paviot,S. Yoo,A. Jerraya, “ Multiprocessor SoC Platforms : A Component Based Design Approach", *IEEE Design & Test of Computers*, pp. 52-63, Nov. 2002,

[7] F. Gharalli, D. Lyonnard, S. Meftali, F. Rousseau, A. A. Jerraya, “Unifying memory and processor wrapper architecture in multiprocessor SoC design”, *proc of ISSS’02*, pp. 26-31, Oct. 2002.

[8] S. Abdi, D. Gajski, “Automatic generation of equivalent architecture model from functional specification”, *Proc. of42th DAC*, pp. 608-613, June, 2004