System-level design refinement using SystemC

by

Robert Dale Walstrom

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Diane T. Rover, Major Professor
Ricky Kendall
Akhilesh Tyagi

Iowa State University

Ames, Iowa

2005

Copyright © Robert Dale Walstrom, 2005. All rights reserved.

Graduate College
Iowa State University

This is to certify that the master's thesis of

Robert Dale Walstrom

has met the thesis requirements of Iowa State University

Major Professor

For the Major Program

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Embedded systems have become increasingly complex with the advent of the system-on-a-chip (SOC) era. Prior to this period, the task of designing an embedded system consisted of integrating microprocessors with other hardware components on a circuit board. The functionality of the system needed to be partitioned to either the hardware components or the software running on the system. Typically the custom hardware components were simple enough that they could be developed using a hardware design language (HDL), such as VHDL or Verilog. As advances in process technology were made in the 1990s, it became clear that both a processor core and the hardware components of an embedded system would be able to fit onto a single chip [1]. However, this advancement also introduced several problems into the traditional method of system design. Software became more closely coupled with the hardware and needed to be considered an integral component during the design of the system. Another issue was that the complexity of SOC design had made it difficult for an HDL to manage. A typical SOC may consist of one or more microprocessors, dedicated hardware processing units, peripheral devices, on-chip memories, and the logic for a sophisticated communications network to link all of these components together. In order to address these issues, designers needed new design languages and tools that would help manage the complexity of SOC designs.

1.1 Motivation

Consumer electronics has been one of the most demanding markets that utilize embedded systems. The consumer electronics industry is so highly competitive that manufacturers strive to place these products on the market as fast as possible. At the same time, consumers demand high performance products that are compact, energy-efficient, and low-cost.

With time-to-market demands requiring manufacturers to produce complex embedded systems faster and cheaper, system-level design (SLD) has become an attractive alternative to traditional design approaches. System-level design languages (SLDLs) and tools allow designers to manage the complexity by using different levels of abstraction to define and model the system. With the support of Engineering Design Automation (EDA) tools, steps involved the design process have become automated. In order to save more time in the development process, SLDLs and EDA tools also focus on the ability to manage and reuse intellectual property (IP) components that have previously been implemented and tested. These features result in the ability to produce very complex systems in a faster and cheaper manner.

Several SLDLs have been introduced in recent years, most notably SpecC [2] developed at the University of California, Irvine and SystemC developed by the Open SystemC Initiative (OSCI) [3]. In some aspects, these SLDLs share the same goals. Both support the ability to model a system at various levels of abstraction and support the reuse of IP. However, while SystemC claims the most industry support with a wide variety of SystemC-based tools available from major EDA vendors, it lacks a well-defined refinement methodology like that of SpecC. The SpecC refinement methodology leads designers from its highest level of abstraction down to its lowest level of abstraction. While each level of abstraction in SystemC is clearly defined, it is not clear what changes need to be made to convert a design from one level of abstraction to another.

1.2 Thesis Statement

In order to facilitate an SLD approach, SystemC needs a well-defined methodology for bringing a model defined at the highest level of abstraction down to the lowest level of abstraction. This thesis presents a top-down refinement methodology for systems modeled in SystemC. Since SystemC has gained widespread industry support, such a methodology would make it easier for designers who use SystemC to refine their design and use the SystemC language as intended by a SLD approach.

1.3 Approach

In this thesis, an SLD refinement methodology is defined for SystemC. Since SpecC already has a well-defined refinement methodology for each of its supported abstraction levels, the SpecC methodology was used as a basis for the proposed SystemC methodology. For this reason, a strong understanding of both languages was necessary. In order to define the refinement rules for the proposed SystemC methodology, the similarities and differences between both languages needed to be considered. To demonstrate the application of the refinement rules for both languages, functionally equivalent SpecC and SystemC models of a digital camera were implemented.

1.4 Contributions

The following is a summary of the contributions of this research project:

- Analysis of the similarities and differences of the SpecC and SystemC SLD languages and modeling capabilities.
- A top-down refinement methodology for SystemC models.
- Demonstration of the SystemC refinement methodology on the digital camera example.
- SpecC and SystemC implementations of a digital camera as a case study.

1.5 Thesis Organization

The remainder of this thesis is organized as follows: Background material is presented in Chapter 2, where overviews of the language features of both SpecC and SystemC are presented. A comparison of the models of computation and a definition of a universal set of models of computation are presented in Chapter 3. The proposed SystemC refinement methodology is presented in Chapter 4. The digital camera system which the case study was based on is presented in Chapter 5. Chapter 6 summarizes related work to SLD methodologies and refinement and is followed by a conclusion and recommendations for future work in Chapter 7.

CHAPTER 2 BACKGROUND

A brief overview of the SpecC and SystemC modeling languages is presented in sections 2.1 and 2.2 respectively.

2.1 SpecC Language Overview

SpecC is an ANSI C-based SLDL developed at the University of California, Irvine. It was introduced in 1997 as a specification language to address the problem of an increasing gap in productivity due to the increasing chip complexity of SOC designs [4]. The codesign methodology for the SpecC language was introduced in 1998, providing the necessary steps to refine the model through each layer of abstraction [5]. This section presents an overview of the main components and features used to build a SpecC model.

2.1.1 Behaviors

Behaviors are the basic unit of functionality in a SpecC model. They represent the computation of the system. A typical behavior consists of ports, local variables, functions, and a main function. There are two types of behaviors: composite behaviors and leaf behaviors [6]. A composite behavior is a behavior that contains instances of child behaviors. A leaf behavior is a behavior that contains no instances of other behaviors.

An example of a leaf behavior is shown in Figure 2.1. In this example, the first line defines the behavior `A` and the two ports of integer type associated with it. Ports allow for communication between behaviors using channels or interfaces. Behavior A has one input port, p1, and one output port, p2. It also has one private local variable, x, which can be accessed only from within the behavior itself. The main function defines the actual functionality of the behavior and is a public function because it must be called by a parent behavior in order to execute. The functionality of the behavior is to read the input data from port p1, increment it by the value of x, and send the result through output port p2.

```
behavior a( in int p1, out int p2 )
{
      int x;

      void main( void );
      {
            x = 1;
            p2 = p1 + x;
      }
};
```

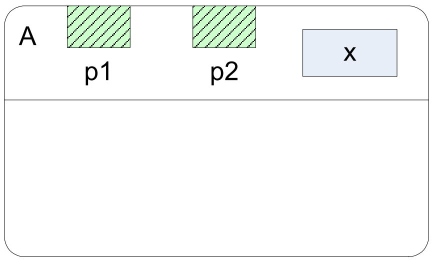


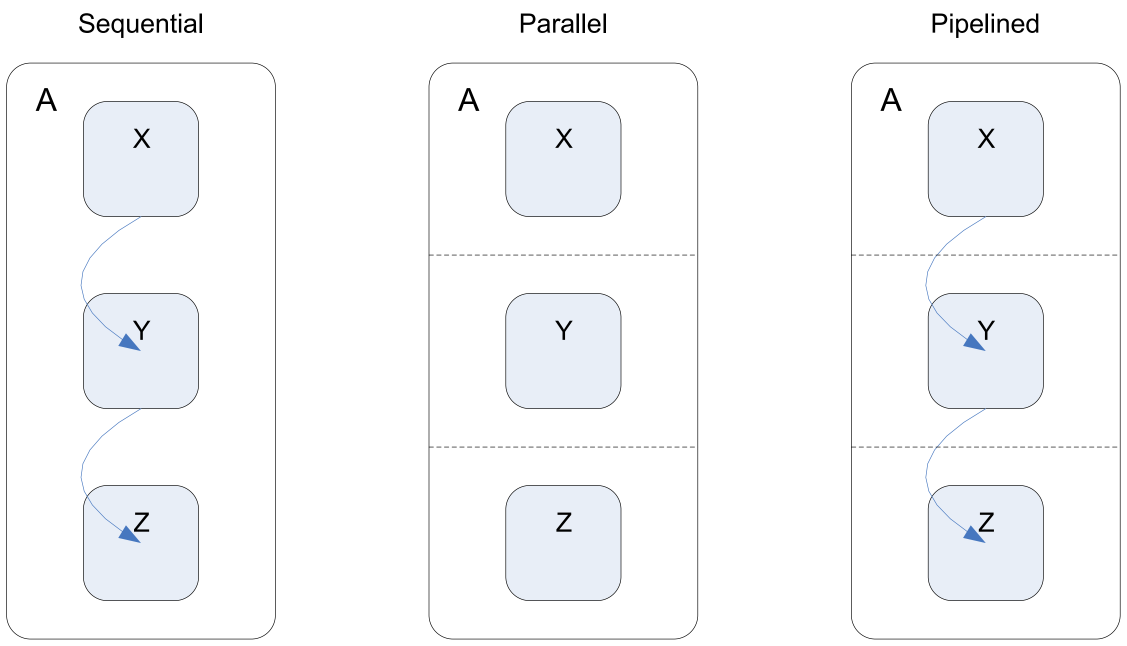
Figure 2.1 SpecC leaf behavior code segment and block diagram

Composite behaviors can have instances of child behaviors, which may or may not be leaf behaviors. This introduces a hierarchy that allows a composite behavior to control its child behaviors. In the `main` function of the composite behavior, the execution of a child behavior is initiated by making a function call to the child's main function. There are three types of execution sequences supported by SpecC: sequential, parallel, and pipelined. The default execution sequence is sequential so that once one behavior finishes execution, the next behavior begins. SpecC also provides the ability to execute behaviors concurrently using the par statement or in a pipelined fashion using the pipe statement. Examples of the different execution types and how they are represented in block diagrams are shown in Figure 2.2.

Composite behaviors also allow for functionality to be abstracted. For instance, a behavior that communicates with a composite behavior does not have direct knowledge of the composite behavior's child behaviors. The composite behavior is seen as a black box from that perspective. Meanwhile the details of the computation are handled by the child behaviors, hidden at a lower level of abstraction.

2.1.2 Channels and Interfaces

Channels are used to represent the communication of the system. The variables and methods found in a channel represent the communication protocol of a communication bus. In some ways, a channel is the same as a behavior. Methods define the functionality and behavior of the communication in a system. The channel is accessed by calling these methods instead of assigning values to or attempting to read from the signals inside the channel. An interface defines the connection between a behavior and a channel. It serves as a prototype of the methods provided by a channel. All that a behavior needs is knowledge of the channel's interface so the behavior knows what methods to call in order to access the channel.



```
x.main();
y.main();
z.main();
```

```
par {
   x.main();
   y.main();
   z.main();
}
```

```
pipe {
   x.main();
   y.main();
   z.main();
}
```

Figure 2.2 Behavior execution styles in SpecC

An example depicting the use of a communication channel between two concurrent behaviors is shown in Figure 2.3. Note that behavior A is the composite behavior and behaviors X and Y are its child behaviors. The dashed line separating the behaviors indicates that the behaviors are executing concurrently, as depicted in Figure 2.2. The channel is noted as c1 and there are two interfaces, L and R. A code listing of this example is shown in Figure

2.4. This code listing demonstrates several important syntactical features of channels and busses. Earlier we mentioned that composite behaviors contain instances of child behaviors. Note that channel c1 and behaviors X and Y are instantiated in behavior A. When defining the behaviors, the port types were defined. If a port on a behavior is supposed to connect to a channel, an interface is specified as the port type in the behavior definition. When the channel c1 and child behaviors X and Y are instantiated in behavior A, the ports are bound using the variables or channels. Also, the behaviors X and Y are to execute concurrently, so the par statement is placed around the main function calls for behaviors X and Y.

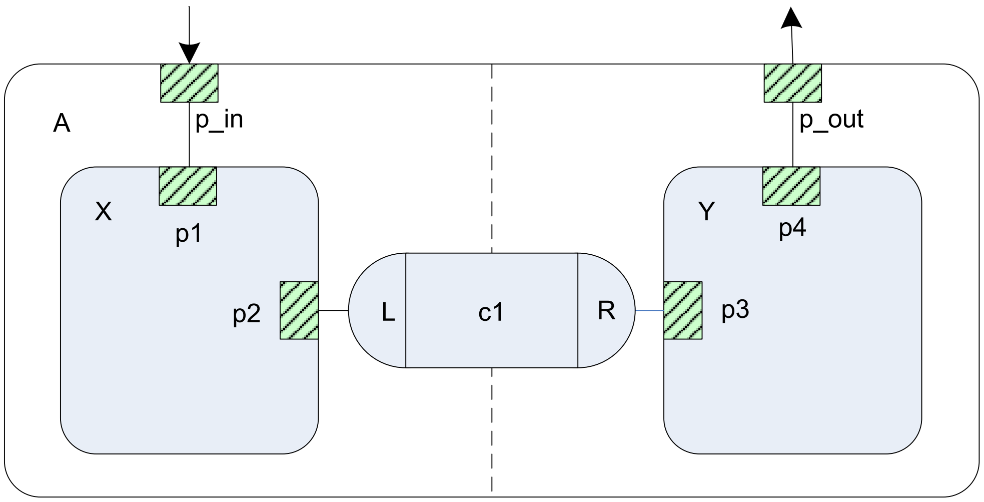


Figure 2.3 Example of a message-passing communication channel

2.1.3 Synchronization

Synchronization in SpecC is done using the built-in event data type. Events can be instantiated inside behaviors or channels and bound to ports like any other data type. However, an event can only be manipulated as arguments for wait and notify statements. When a wait statement is called on an event by a behavior, the execution of that behavior is suspended until that event is notified by another behavior. When a behavior calls a notify statement on an event, all of the behaviors that are waiting on that event will resume execution.

```
interface L { void write( int data_in ) };
interface R ( int read( void ) };

channel C implements L, R
{
      int   data;
      bool  valid;

      void write( int v ) {
            data = data_in;
            valid = true;
      }

      int read( void ) {
            while( valid == false ) {}
            return( data );
      }
};

behavior X ( in int p1, L p2 )
{
      void main ( void )
      {
            if ( p1 > 5 )
                  p2.write( p1 );
            else
                  p2.write( 0 );
      }
};

behavior Y ( R p3, out int p4 )
{
      void main ( void )
      {
            p4 = p3.read();
      }
};

behavior A ( in int p_in, out int p_out )
{
      C     c1;
      X     x( p_in, c1 );
      Y     y( c1, p_out );

      void main ( void )
      {
            par { x.main();
                  y.main();
            }
      }
}
```

Figure 2.4 SpecC code for message-passing communication channel example

2.1.4 Timing

In order to simulate time in models of computation which are concerned with timing, SpecC provides waitfor statements. All other statements in a SpecC program are executed in zero time, so waitfor statements allow for exact delays to occur in particular parts of the code. The waitfor statement accepts a single integer argument, which is the number of time units (nanoseconds) that a behavior is supposed to suspend execution.

2.1.5 Summary

The previous sections covered many of the basic components and features used to create a SpecC program. Although more details about the SpecC language can be found in [6][7][8], only the modeling components used in this thesis have been covered.

2.2 SystemC Language Overview

SystemC is a C++ library-based language developed by the OSCI Language Working Group [3]. Like SpecC, the language was introduced in response to the problem in that it was no longer sufficient for designers to use HDLs such as Verilog or VHDL for SOC designs. The SystemC language provides a number of specialized classes, types, and macros used for the various modeling components and features. In this section an overview of the modeling components used in SystemC designs is presented.

2.2.1 Modules and Processes

The basic building blocks of a SystemC design are modules and processes. Modules are like SpecC behaviors, in that they are used to partition the design of a complex system and can use hierarchy to hide some of the internal details of a module. Modules also use ports for communication and can have internal variables. Modules are defined using the SC_MODULE macro.

The functionality of a module is implemented using processes. Processes are basically member functions of a module that are executed concurrently. There are two different types of SystemC processes, method and thread. The main difference between the two types of processes is that a method process will always execute its code from start to finish without interruption while a thread process has the ability to suspend and resume its execution. Member functions are mapped to processes using SC_METHOD or SC_THREAD statements. Each module also has a constructor which is where mapping of member functions to processes takes place. Modules typically use the default constructor, SC_CTOR. If a module contains instances of other modules, then the ports of the child modules are mapped to signals in the constructor as well.

The SystemC version of the SpecC leaf behavior originally shown in Figure 2.1 can be seen in Figure 2.5. The sc_in and sc_out types are used to define the input and output ports. The sc_int type is used to define the integer variable x. The line, sensitive << p1, is called to add the port p1 to the sensitivity list. This is done so that each time the value for p1 changes, the member function func1 will be executed again to derive the new result and send it out of the module through the p2 port.

```
SC_MODULE( A )
{
      sc_in<int>  p1;
      sc_out<int> p2;
      sc_int<8>   x;

      void main() {
            x = 1;
            p2 = p1 + x;
      }
      SC_CTOR( A ) {
            SC_METHOD( main );
            sensitive << p1;
      }
}
```

Figure 2.5 SystemC version of the SpecC leaf behavior

2.2.2 Channels and Interfaces

Like SpecC, SystemC supports the use of hierarchical channels and interfaces for modeling communication. All interfaces in a SystemC program are derived from the class sc_interface. Each interface is used to specify operations that are able to be performed over that particular channel. Channels are used to implement interfaces in much of the same way as SystemC.

2.2.3 Synchronization

Synchronization can be done through the use of events, supported in SystemC by the sc_event class. Events in SystemC are used to determine if and when the execution of a process should be triggered or resumed, depending on whether the process is a method or thread. An event object keeps track of all the processes that are sensitive to it, so when it is asserted it tells the scheduler which processes to trigger or resume execution. There are two ways a process may be sensitive to an event, static sensitivity or dynamic sensitivity. If an event is on the static sensitivity list, as shown in Figure 2.5, then the processes in that module will always be sensitive to that event. If the process is a thread, then the process may use the wait function to wait on an event. Dynamic sensitivity in SystemC is the same method in which sensitivity to events is handled in SpecC. It should be noted that when a SystemC process uses a wait statement, its static sensitivity list is ignored.

Another SystemC class that is useful for synchronization is the sc_fifo channel. These are particularly useful in writing functional models where communication and synchronization can be simplified. Reads done on a sc_fifo channel are blocking, so execution of the particular process that makes the read call will halt until there is something written to the FIFO. Likewise, writes done on a sc_fifo channel will be blocked if the channel is full. When a sc_fifo channel is instantiated, the size may be specified as either finite or dynamic in terms of the number of tokens, as seen in Figure 2.6. Due to the nature of FIFOs, where blocking is predictable, the execution of a model can be deterministic.

```
sc_fifo<int>  fifo1("fifo1");       // Dynamic FIFO
sc_fifo<int>  fifo2("fifo2", 1);    // FIFO of size 1
sc_fifo<bool> fifo3("fifo3", 10);   // FIFO of size 10
```

Figure 2.6 FIFO instantiations of different sizes

2.2.4 Timing

The wait statement that is used for events may also be used for waiting a specified amount of time. Used in this form, the SystemC wait statement is similar to the waitfor statement in SpecC. The main difference is that the wait statement in SystemC can be instructed to wait in different units of magnitude, like nanoseconds or picoseconds. The wait statement may also be used to wait either for a specified amount of time or for an event to occur. This is useful in case an event the process is waiting for does not occur. After the specified amount of time has elapsed, the process will resume execution.

2.2.5 Summary

The previous sections covered many of the basic modeling components and features of SystemC. Although more in depth information about these and other SystemC features can be found in [9], only the elements used in this thesis have been covered.

CHAPTER 3 MODELS OF COMPUTATION

The primary goal of SLD is to make it easier for designers to manage highly complex systems. One essential aspect of a SLDL is the ability to support modeling of the system at multiple layers of abstraction. The problem with designing complex systems with traditional HDLs is that they often support only one layer of abstraction, so detailed design decisions must be made early on in the design process. In the SLD approach, designers are able to test functionality of their systems at each layer of abstraction before getting into intricate details of the final implementation.

In a top-down SLD methodology, a designer begins by modeling the functional specification of a system. At this point the details of the system are highly abstracted. The designer then refines the model to gradually define more details about the system, each transformation producing a distinct model at a particular layer of abstraction. In SLDLs such as SpecC and SystemC, a model that is defined at any layer of abstraction may be simulated, verified, and debugged. Such a model is referred to as a model of computation. A model of computation is distinguished by how accurately it represents the target implementation of the system.

The models of computation supported in SpecC and SystemC are described in sections 3.1 and 3.2 respectively. Then the accuracy of each model of computation is quantified using metrics presented in section 3.3. Lastly, a set of universal models of computation are presented in section 3.4.

3.1 Models of Computation in SpecC

In [7], the models of computation in the SpecC methodology are presented. The four models supported by SpecC are a specification model, an architecture model, a communication model, and an implementation model. The SpecC methodology also specifies three refinement tasks necessary to transform one model of computation to the next. These three refinement tasks are architecture exploration, communication synthesis, and backend. The hierarchy of the models of computation and the refinement tasks for of the SpecC methodology are shown in Figure 3.1.

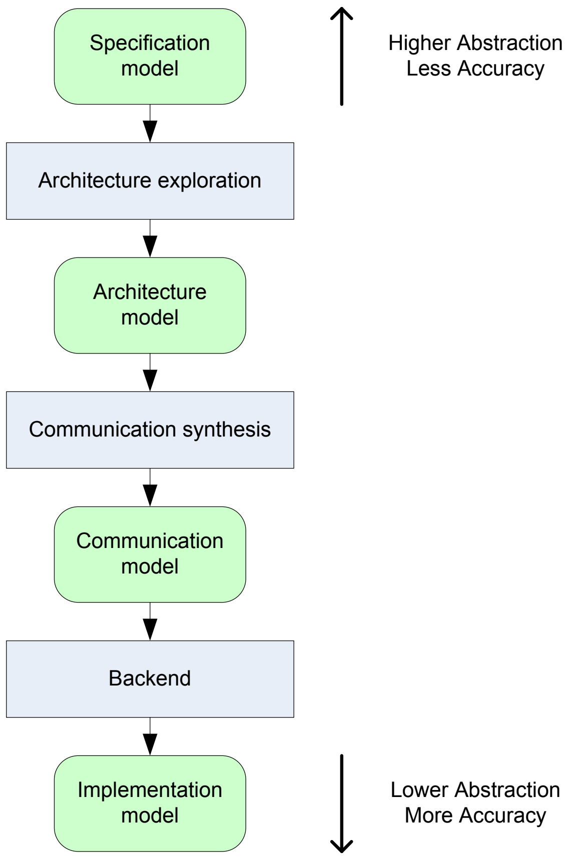


Figure 3.1 The SpecC design methodology

Each model of computation represents a stage in the design process. As each refinement task is performed, a new model of computation is derived that reflects design decisions made at that stage in the methodology. In this top-down methodology, a design starts as an abstract definition of the system in terms of functionality that is transformed into a detailed implementation of the system.

3.1.1 Specification Model

The specification model is the top-most model of computation in the SpecC methodology. Since this is the first model of the design process, the purpose of the specification model is to define how the system is supposed to behave. The overall functionality of the system is broken down into computational behaviors. The specification model does not attempt to define any implementation details at this point and has no notion of timing. Communication is only done directly between modules using events for synchronization. Therefore the specification model is only intended to reflect the functionality of the target system.

3.1.2 Architecture Model

The architecture model is the next model of computation in the SpecC methodology. This model is derived from the specification model after performing the architecture exploration refinement task. The architecture model defines the structure of the system in terms of system components (such as processors, busses, and memories). The purpose of the architecture exploration refinement task is to use these components as building blocks to determine the architecture of the system. Behaviors from the specification model are mapped to specific processing elements, which are used to represent standard or custom processors. The architecture model also introduces the notion of timing to the model, so the execution times for these processing elements are annotated in the form of estimated execution delays. The synchronization events between behaviors that execute on different processing elements are now separated into communication behaviors, called virtual busses. These virtual busses are highly abstract behaviors of the communication channels and their respective protocols. These abstracted communication channels consist of methods that encapsulate the protocols and details of the transactions so that they are separated from the processing elements. Any estimated delays associated with the communication channels are annotated in the architecture model as well. Thus the architecture model represents the target architecture of the system in terms of hardware-software partitioning as well as representing an abstract form of communication between the components.

3.1.3 Communication Model

After performing the communication synthesis refinement task on the architecture model, the communication model is formed. The abstract communication behaviors in the virtual busses of the architecture model are transformed into implementations of the actual wires of a communication bus and the protocols are integrated into the processing elements. The interfaces to channels are changed to pins in order to connect to the wires of the transformed communication channels. The estimated timing delays associated with communication are replaced with cycle-accurate delays associated with the protocol. The purpose of the communication model is to define all of the communication aspects of the target system, but the architecture of the system remains unchanged from the architecture model.

3.1.4 Implementation Model

The final model in the SpecC methodology is the implementation model. This model represents the lowest layer of abstraction of the system, where all aspects are defined explicitly. The backend refinement task separates the communication model into hardware and software components. The processing elements of the communication model are replaced with cycle-accurate representations of the target processor, such as an instruction set simulator. The software portion of the model is compiled into assembly code to execute on the target processor. A high-level synthesis tool is used to synthesize the custom hardware and communication channels into a register-transfer level model of the hardware in the system. Using a cosimulation tool to simulate both the software and hardware aspects of the system simultaneously, the implementation model provides a cycle-accurate representation of the system.

3.2 Models of Computation in SystemC

In [9], the five models of computation supported by SystemC are presented. These five models of computation are an untimed functional model, a timed functional model, a transaction-level model, a behavioral hardware model, and a register-transfer model. In terms of abstraction and accuracy, the hierarchy of the models of computation supported by SystemC can be seen in Figure 3.2.

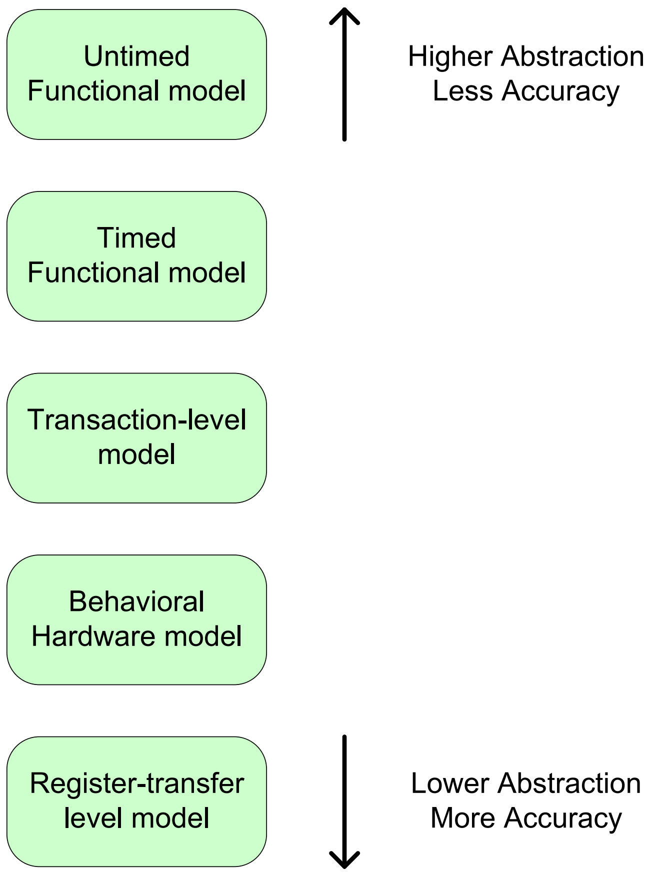

Untimed Functional model
Higher Abstraction Less Accuracy
Timed Functional model
Transaction-level model
Behavioral Hardware model
Register-transfer level model
Lower Abstraction More Accuracy

Figure 3.2 Models of computation in SystemC

3.2.1 Untimed Functional Model

The untimed functional model is a functional specification of the target system. All of the functionality of the system is implemented in this model, but there is no reference to any architectural details of the system. As the name implies, the model has no notion of timing either. When the untimed functional model is simulated, only the functional results may be verified. Functionality may be broken down into modules if it assists in making the modeling process easier, but is unnecessary. Communication between modules is done implicitly as there are no communication links or busses being modeled in this layer of abstraction. This model of computation is also called an executable specification.

3.2.2 Timed Functional Model

The timed functional model in SystemC is functionally the same as the untimed functional model, but includes the notion of timing during simulation. Approximate timing constraints are annotated so that the computation delays associated with the target implementation can be estimated. No details regarding the communication between modules are defined at this level since it is still done implicitly. All other aspects in comparison with the untimed functional model remain the same.

3.2.3 Transaction-level Model

The transaction-level model defines the communication between modules by using function calls. This models accurate functionality of the communication protocol and isolates the communication details from the computational functionality. The transaction-level model has approximated timing annotations in both communication functions and computational modules to indicate a rough estimate of the timing characteristics of the system. In the transaction-level model, the modules represent computational components or processing elements and the function calls related to communication represent the communication busses of the target implementation.

3.2.4 Behavior Hardware Model

The behavior hardware model has detailed implementations of the communication busses of the target system. The communication protocols of the target implementation are inserted into the processing elements. Instead of the abstract communication interfaces used in the transaction-level model, wires represent the communication busses and pins are added to the processing elements so they may be connected to the wires. The computational timing is approximate-timed, so the key difference between the transaction-level model and the behavior-hardware model is whether the communication aspects are abstract or accurate.

3.2.5 Register-Transfer Level Model

The register-transfer level model is the most accurate model supported by SystemC. All of the communication, computation, and architectural aspects of the target system are defined explicitly. Timing characteristics of both the computational and communication elements are clock-cycle accurate. At this layer of abstraction the SystemC code representing the hardware components is translated to a HDL that can be synthesized and the SystemC code representing software is translated into the desired software programming language.

3.3 Quantifying Accuracy of Models of Computation

In order to compare the models of computation supported by the SystemC language with the models of computation supported by the SpecC methodology, a set of independent metrics was used to determine the accuracy of a model of computation. Based on [9] and inspired by [10], the following metrics were used for determining the accuracy of each model of computation:

- **Functional accuracy:** A model is said to be functionally accurate if it reflects the functionality of the target implementation of the system.
- **Computational timing accuracy:** The magnitude to which the model reflects the computational delays of the system. Computational delays typically refer to processing delays, memory accesses, and delays due to resource constraints. Using this metric, a model of computation can be cycle-accurate, approximate-timed, or untimed.
- **Communication timing accuracy:** The magnitude to which the model reflects the communication delays of the system. Using this metric, a model of computation can be cycle-accurate, approximate-timed, or untimed.
- **Communication protocol accuracy:** The magnitude to which the actual communication protocols of the target implementation is modeled. Using this metric, a model may be considered to be abstract if it reflects the functionality of the communications protocols. If the structure of the communication protocol is modeled in the same fashion as the target implementation, the model is considered to be exact in terms of communication protocol accuracy.
- **Structural accuracy:** The magnitude to which the model reflects the true structure of the target implementation. The structural accuracy is said to be approximate if the partitioning of functionality into hardware and software is same as that of the target implementation. The structural accuracy is said to be exact if the model's structure accurately reflects the internal structure of the components in the target implementation.
- **Pin accuracy:** A model is said to be pin accurate if the interfaces between components are defined at the pin level.

Using these metrics, the accuracy of a model of computation in terms of a range of criteria may be determined. In a top-down design methodology, the implementation details of the system are defined as the refinement rules transform the model of computation from one layer of abstraction to the next.

3.4 Universal Models of Computation

As presented in sections 3.1 and 3.2, there are four different models of computation in the SpecC methodology and five different models of computation supported by SystemC. In order to simplify further discussion of the models of computation between both SpecC and SystemC, a universal set of models of computation were derived. This universal set of models of computation describes the system at the same levels of accuracy regardless of which SLDL the model is coded in. To determine these universal models of computation, each model of computation for both SpecC and SystemC was analyzed using the metrics defined in section 3.3. The results of these analyses are shown in Table 3.1.

Table 3.1 Accuracy comparison of models of computation in SpecC and SystemC

Accuracy	SpecC				SystemC				
Metrics	Spec	Arch	Comm	Impl	UFM	TFM	TLM	BHM	RTLM
Functional	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Comp. Timing	No	Approx	Approx	Cycle	No	Approx	Approx	Approx	Cycle
Comm. Timing	No	Approx	Cycle	Cycle	No	Approx	Approx	Cycle	Cycle
Comm. Protocol	No	Approx	Exact	Exact	No	No	Approx	Exact	Exact
Structural	No	Approx	Approx	Exact	No	No	Approx	Approx	Exact
Pin	No	No	Yes	Yes	No	No	No	Yes	Yes

Based on these results, the equivalent models of computation between SpecC and SystemC can be identified. In most cases the equivalent models in both SpecC and SystemC are easy to determine. However, there is no equivalent model of computation in SpecC for the timed functional model in SystemC. This discrepancy is considered to be minor since the only refinement step between the untimed functional model and timed functional model is to annotate timing delays in the system. Therefore, the timed functional model is not included in the set of universal models of computation. The notion of timing will be introduced in the transaction-level model. The universal set of models of computation and their equivalent models in SpecC and SystemC are shown in Figure 3.3.

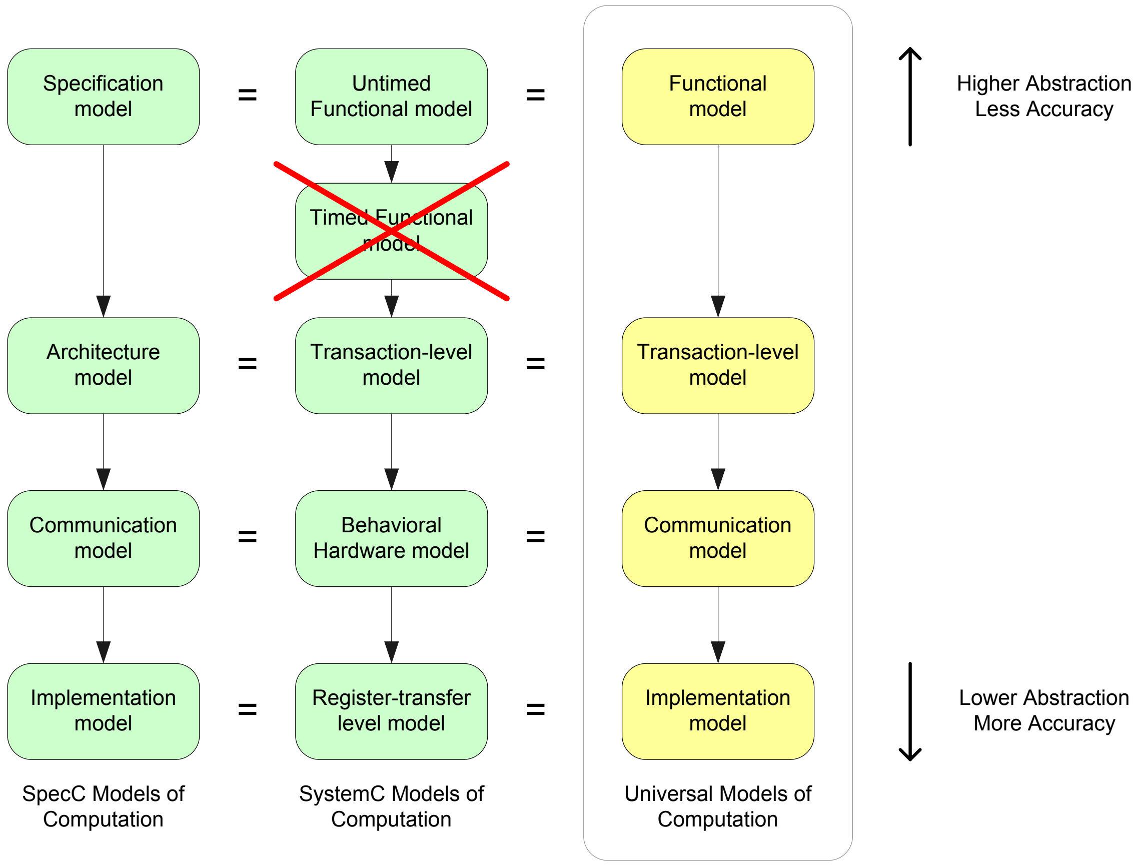


Figure 3.3 Universal models of computation

For the remainder of this thesis, the names of the universal models of computation will be used as opposed to the language-specific models presented in sections 3.1 and 3.2. In review, the general distinctions of the four models of computation are as follows. The functional model is purely representative of the functionality of the target system model, with no notion of timing or architecture. The transaction-level model reflects the target architecture of the system and the computational units and communication channels are approximate-timed. The communication model contains the same approximate-timed computation units of the architecture model, but also includes accurate implementations of the communication busses in terms of timing and protocol. The implementation model of the system is accurate in terms of all metrics and represents the lowest level of abstraction supported by a SLDL.

CHAPTER 4 SYSTEMC REFINEMENT METHODOLOGY

This chapter provides a set of rules for refining SystemC models of computation in a top-down design methodology. These rules have been derived through the modeling and refinement of the digital camera example presented in Chapter 5.

4.1 Developing a Functional Model

The functional model defines the functionality or behavior of the system, without concern for the target architecture. Therefore, the functional model must only consider the functionality in terms of components, with no regard to whether those components are implemented in hardware or software. The objective is to capture the specification of the system in terms of design behavior with the least amount of design work.

4.1.1 Guidelines for Functional Models

The first step in developing a functional model is to determine how the functionality of the system should be partitioned into processes. The granularity of the processes can vary, but each process should be fairly independent. Since functionality of the system is represented by processes that run in leaf modules, the smallest indivisible units of functionality are leaf modules. The computational details of a module are specified in one or more processes. Behavioral hierarchy, where modules may contain instances of other modules, may be used in order to mask some of the details of the functionality into modules at lower levels. The processes should be placed into independent modules in an effort to maximize concurrency. Independent modules allow more flexibility when architectural considerations are made later on in the design methodology.

Although the simulation of a functional model occurs in zero simulation time, the code inside of a process is executed in a sequential fashion. Recall from section 2.2.1 that there are two main types of processes: sc_thread and sc_method. A sc_method process will execute from start to finish without halting its execution while a sc_thread process has the ability to halt its execution using wait statements or other forms of blocking. For functional models, all processes should be instantiated as sc_thread processes.

However, the use of multiple thread processes in a single module with the intent that they will execute concurrently is discouraged. The concurrency of multiple thread processes instantiated inside a module should not be considered the same type of concurrency as that found in threaded software applications. This is due to the simulation scheduler, which does not function in the same way as an operating system scheduler. A context switch will only occur when the execution of the current process is halted, otherwise the current process will continue to execute indefinitely. For this reason it is usually safest to use only one process per module when developing a functional model. In general, move processes that are supposed to be independently concurrent into separate modules, as shown in Figure 4.1.

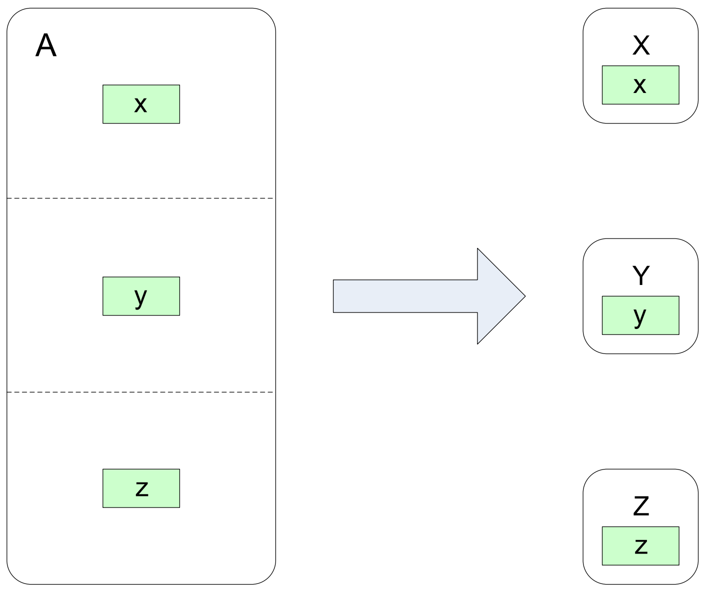


Figure 4.1 Binding of concurrent processes to modules

Due to the nondeterministic nature of how the simulator selects the next process to run, a common pitfall of using events to synchronize processes is to have the notifying process send out the notification before the process that waits for that notification has been executed. For example, the module on the left in Figure 4.2 has three processes that are supposed to execute sequentially. If process x executes first and has no wait statements used to halt its execution, it will send out the event to notify process y before process y can execute the wait statement for that event. When process y finally executes, it will call a wait statement on that event. However, process x had already sent the notification, so the system becomes deadlocked. In this case, the intent is to have each process execute sequentially. Therefore it may be more helpful to merge the functionality of the processes into a single process in order to guarantee correct execution. This merging of processes into a single process is illustrated in Figure 4.2

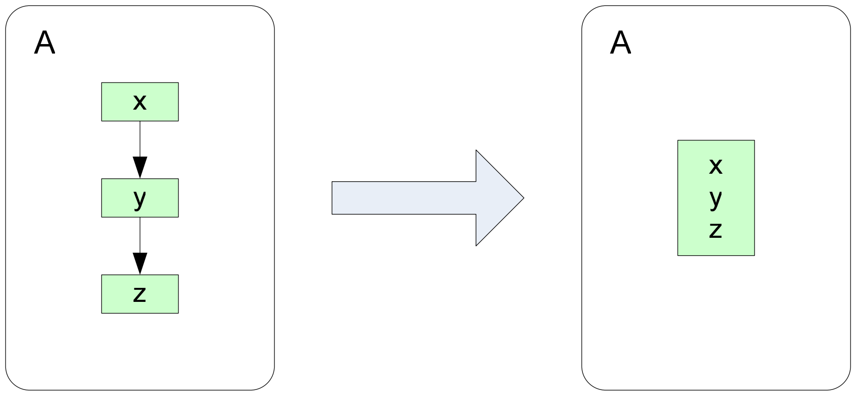


Figure 4.2 Merging of sequential processes into a single process

All communication and synchronization between modules should be implemented using the sc_fifo primitive channel type. These primitive channels can be accessed using read and write methods which are blocking calls. When a read method is called, the execution of a process will be blocked if no data is available to be read. Once data has been written to the FIFO, the process may finish the pending read and resume execution. Similarly, if a process attempts to write to a FIFO that is full, its execution is halted until space is available in the FIFO for the process to complete the write operation. Using FIFOs for communication allows the simplification of a both synchronization and data exchange. Although the use of FIFOs may not accurately reflect the target implementation, the functional model is not concerned with those details. FIFOs make the execution of a functional model deterministic and predictable due to the blocking nature of the read and write functions. In some cases, a FIFO channel may need to be initialized with a value before the simulation starts. For example, a process that must read from a FIFO channel before another process is able to write to it may cause the simulation to fail. Initial values of FIFOs may be specified by calling the write method to push an initial value onto the FIFO.

4.1.2 Summary of Guidelines for Functional Models

The following is a summary of the guidelines for developing functional models:

- Divide functionality into individual processes
- Use sc_thread processes, not sc_method processes
- Separate concurrent processes into individual modules
- Merge sequential processes into a single process to guarantee execution order
- Use the sc_fifo primitive for all communication and synchronization between modules

4.2 Deriving a Transaction-level Model

The transaction-level model takes the executable specification of the functional model and separates the architectural and communication behavior of the system into isolated entities. The task of deriving a transaction-level model from a functional model consists of a set of refinement rules which are divided into two distinct phases. The architecture partitioning phase introduces the notion of timing and the mapping of behaviors to processing elements. The communication partitioning phase separates the behavior associated with communication from the behavior associated with computation.

4.2.1 Refinement Rules for Transaction-level Models

The first step in transforming a functional model into a transaction-level model is to insert delays associated with the computation of the system. Annotating computation delays in a functional model is done by the insertion of wait statements into portions of the processes. The frequency at which timing annotations are inserted is determined by the designer. If a rough timing estimate of a portion of code is known, the timing may be annotated at the end of the block. If a more exact estimate is known, the designer may wish to annotate delays after each statement in a process. It does not matter to the simulator if the delays are annotated line by line in the code or as a single delay at the end of a block of code. However, being more specific may yield more accurate simulation results which is important when trying to make informed design decisions early on.

The wait statement takes an argument of a SystemC data type, sc_time. When a variable of type sc_time is declared, the number of units and the unit magnitude need to be specified. Any positive number may be specified for the number of units and SystemC provides several different types of unit magnitude, which is an enumerated type called time_unit. All of the supported values and their meanings [9] are shown in Table 4.1.

Table 4.1 Values and meanings of time_unit

SC_FS	Femtosecond
SC_PS	Picosecond
SC_NS	Nanosecond
SC_US	Microsecond
SC_MS	Millisecond
SC_SEC	Second

When inserting delays using the wait function, the amount of delay may be declared as a sc_time variable and passed to the wait function or the number and time_unit may be passed directly. The equivalence of both methods is shown as an example in Figure 4.3.

After the insertion of computational delays into the model, the next step is to specify the architectural structure of the system. This is done by allocating modules intended to represent processing elements that will be used in the target implementation. Examples of processing elements include processors, microcontrollers, single-purpose processors, or custom ASICs. Once these processing elements have been allocated, the next step is to map the modules of the functional model to these processing elements. When mapping modules to processing elements, it is recommended that concurrent processes are mapped to separate processing elements because eventually the modules that are on the same processing element will be executed in a serial fashion. Once the modules have been mapped to their respective processing elements, care must be taken to ensure that they preserve the same execution sequence as they did in the functional model. This may involve inserting additional synchronization between processing elements.

```
sc_time  comp_delay(100, SC_NS);
.
.
.
wait(comp_delay); // Both of these statements
wait(100, SC_NS); // are the same.
```

Figure 4.3 Examples of wait function calls

The next step is to map any global variables of the functional model into either local memories of the processing elements that use them or into a shared memory. Finally, global channels are allocated to replace the instances of sc_fifo used in the functional model. This concludes the architecture partitioning phase of refinement for deriving a transaction-level model.

An example illustrating the architecture partitioning phase of the transaction-level model refinement process is shown in Figure 4.4. As shown in the example, there are two functional modules, X and Y, which perform a computation before each passes a resulting value to module Z. Module Z is dependent upon both X and Y, while X and Y are not dependent upon each other. When partitioning the modules onto processing elements, modules X and Y are mapped to separate processing elements in order to maintain concurrency. Since module Z is dependent on both X and Y, it could have been placed on either processing element. The variables v1 and v2 are placed into local memories of their respective processing elements. When module Y is finished executing, it uses synchronization to tell Z it has finished and it passes the computed value of v2 over the global channel. The synchronization is performed using a global sc_event called sync.

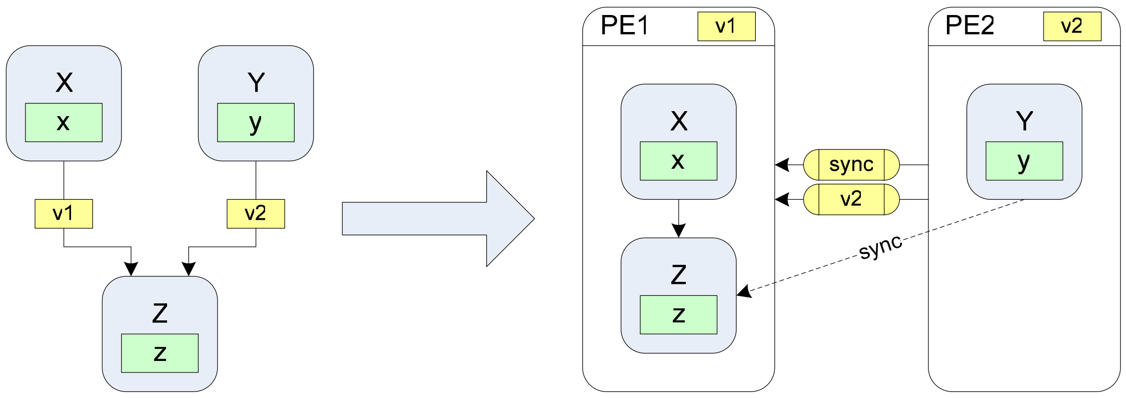


Figure 4.4 The architecture partitioning phase of transaction-level model refinement

The next step is to group channels between processing elements into one or more hierarchical channels. This step is called the communication partitioning phase of transaction-level model refinement. Hierarchical channels are used to form busses from the global channels defined during the architecture partitioning phase. The advantage of using a hierarchical channel is that the details of the communication protocol implementation are abstracted and the processing elements are connected through a single port. Access to the bus is allowed only through the interface, which is done in the form of function calls. The hierarchical channel forms an approximate representation of the bus that will be later refined into the target implementation. Timing delays for communication are also annotated in the functions of a hierarchical channel to model delays associated with transfers over the bus.

After busses have been formed using hierarchical channels and the interfaces to the channels have been defined, the ports on the modules need to be updated so they are able to connect to their respective interface. The processes also need to be updated, replacing the old port accesses with function calls through the interface. Based on the same example introduced in Figure 4.4, the transition from the architecture partitioning phase to the communication partitioning phase of the transaction-level model refinement is shown in Figure 4.5.

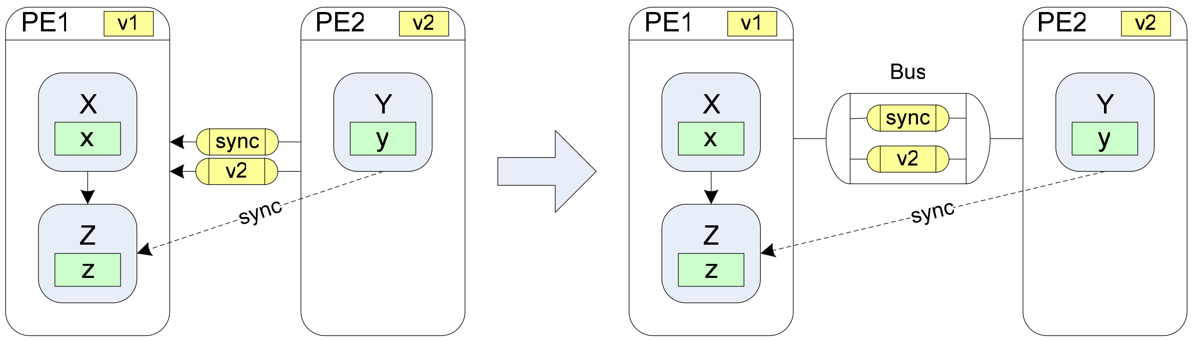


Figure 4.5 The communication partitioning phase of transaction-level model refinement

The global sync event and global channel for transmitting the value of v2 are now encapsulated by a hierarchical channel. The methods for accessing the hierarchical channel are defined within the interfaces and the processing elements no longer access the signals in the channel directly. The processing elements are now able to send data over the hierarchical channel without knowledge of how the communication protocol is able to communicate. The communication is now modeled using a simplified transaction-based approach.

4.2.2 Summary of Refinement Rules for Transaction-level Models

The following is a summary of the refinement rules for deriving transaction-level models from functional models. The refinement rules for the architecture partitioning phase are:

- Annotate delays of the computation and communication aspects of the system using wait statements
- Allocate modules to represent processing elements of the target system
- Map functional modules to designated processing elements
- Replace sc_fifo instances with global channels that use variables for data transmission and the sc_event type for synchronization
- Add any necessary synchronization to preserve the original execution sequence
- Move global variables into local or shared memories

The refinement rules for the communication partitioning phase of transaction-level model refinement are:

- Group global channels into hierarchical channels to form busses
- Define interfaces that provide member functions to access hierarchical channels and implement the communication protocol

4.3 Deriving a Communication Model

When refining the functional model to the transaction-level model, busses were formed using hierarchical channels. The advantage of hierarchical channels is that they allow the communication protocol to be abstracted, hidden from the processing elements. The process of deriving a communication model is focused on converting the abstracted implementation of the communication behavior into a pin-level implementation consisting of wires that make up the busses of the target implementation. Similar to transaction-level model refinement, the refinement rules are divided among two distinct phases that represent the process of deriving the communication model. These two phases are adapter synthesis and protocol insertion.

4.3.1 Refinement Rules for Communication Models

The first step in communication model refinement is to analyze the current implementation of each hierarchical channel in order to determine the desired communication method of the final implementation. When deriving the transaction-level model, the hierarchical channels implied a certain protocol. At the adapter synthesis phase in the communication model refinement, the cycle-accurate implementation of the protocol must be either built from scratch or implemented based on an existing protocol specification.

Next each bus must be defined at the pin-level. This is done using signals to represent the actual wires of the bus for the target system. The abstracted timing constraints associated with the communication delays will be replaced by actual clock cycles that are introduced in the implementation of the communication protocol used to implement functionality for bus transfers.

The final step in the adapter synthesis phase is to define the adapters that will be used to connect the approximate-timed processing elements with hierarchical channel interfaces to the pin-level implementations of the busses. Adapters are modules that contain pin-level interfaces to access the wires of the bus and clock driven, cycle-accurate implementations of the interface functions that were previously defined for the transaction-level model. The purpose of the adapter is to allow the processing element to use the pin-level bus implementation through the high-level interface functions defined in the transaction-level model. The processing element has no knowledge of how the communication protocol is actually implemented because it continues to use an interface function calls first defined in the transaction-level model. The interface functions within the adapters are modified to implement the protocol by manipulating the wires and following the timing rules of the protocol specification. At this point, the system may be simulated to verify the correctness of the protocol. An example illustrating the insertion of adapters around a pin-level implementation of a bus is shown in Figure 4.6.

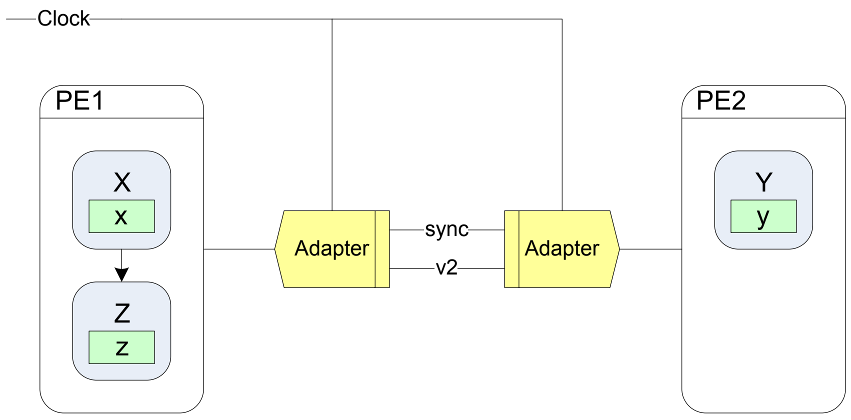


Figure 4.6 Result of the adapter synthesis phase of communication model refinement

The goal of the protocol insertion phase of communication model refinement is to merge the protocol contained in the interface functions of the adapters into their respective processing elements. The port definitions used to access the wires of the bus are moved from the adapter to the processing element. Next, the interface functions are inserted into their respective processing elements. Finally the function calls made by the processing element are changed to local function calls for the newly inserted interface functions. The processing elements receive a clock input, but the clock is only used for the communication protocol implementation. The computational portions of the processing elements remain approximate-timed in their implementation. The previous example following the protocol insertion phase is shown in Figure 4.7.

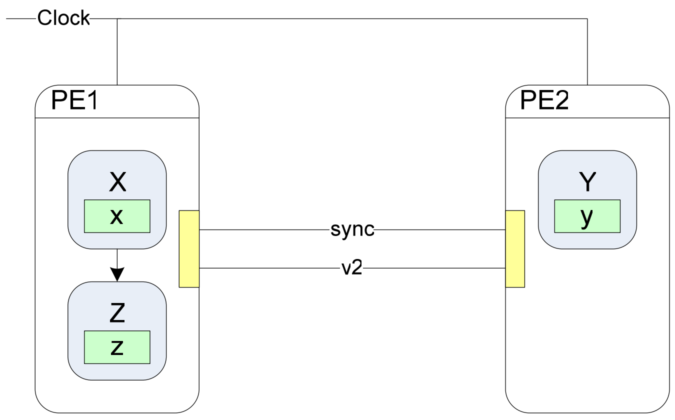


Figure 4.7 Result of the protocol insertion phase of communication model refinement

4.3.2 Summary of Refinement Rules for Communication Models

The following is a summary of the refinement rules for deriving communication models from transaction-level models. The refinement rules for the adapter synthesis phase are:

- Replace hierarchical channels with pin-level implementations using signals to represent wires
- Create adapter modules that implement the interface functions of the deprecated hierarchical channels and connect to the signals of the busses using pin-level interfaces

The refinement rules for the protocol insertion phase of communication model refinement are:

- Merge pin-level interface definitions of the adapters into processing elements
- Merge interface functions from the adapters into processing elements
- Change the interface function calls in the processing element threads into local function calls

4.4 Deriving an Implementation Model

The implementation model is the lowest level of abstraction that SystemC supports. The implementation model is derived from the communication model, which is still approximate-timed in terms of computation and remains abstract in terms of internal structure of the computation units. The hardware components of an implementation model look very similar to HDL implementations. The refinement rules for determining implementation model are the most general due to the multitude of implementations that can be derived from a communication model. Also, the final result of implementation model refinement is the actual implementation of the system in terms of software and synthesizable hardware. In order to simulate the model, cosimulation tools that support simultaneous simulation of hardware and software are needed.

4.4.1 Refinement Rules for Implementation Models

The communication model already contains cycle-accurate, pin-level implementations of the communication busses, so the purpose of the implementation model refinement is to derive cycle-accurate implementations of the processing elements that are targeted for hardware implementations. The first step is to identify distinct states that are contained in the code of a module's one or more processes. The sc_thread processes are to be changed into sc_method processes, so they are no longer able to halt execution using wait statements. Thus a single sc_thread process may need to be split up into several sc_method processes in order to perform the same functionality. Once the number of sc_method processes has been determined, the sensitivity list needs to be updated to include all of the signals that a particular method should evaluate when they change. The next step is to replace all instances of variables and other abstract types into signals. The only exception to this case is if a variable is used in only one process, in which case it cannot be read from and written to by different processes at the same time. Signals feature semantics that handle simultaneous reads and writes while variables do not. All initial values of signals and variables should be defined in the module's constructor.

Once the low-level implementations of the hardware processing elements have been derived, the system is ready for export into languages used to define the implementation of a system. Hardware components are converted to HDLs such as VHDL or Verilog, which can be synthesized. Low-level hardware implementations follow many of the same conventions of HDLs and the conversion is typically straightforward. Processing elements that are used to run software are typically transformed into both a HDL representation of the microprocessor and the software that it executes. In most cases an IP model of the microprocessor will be used and the software code is derived from the C-based SystemC implementation of the module. After the SystemC code has been exported, the communication model has then been refined into the target implementation of the system.

4.4.2 Summary of Refinement Rules for Implementation Models

The following is a summary of the refinement rules for deriving implementation models from communication models:

- Replace abstracted processing elements that are targeted for hardware into low-level implementations
- Export hardware components into HDL implementations
- Export software components into the language supported by target microprocessor(s)

CHAPTER 5 CASE STUDY: A DIGITAL CAMERA

In Chapter 4, a set of refinement rules for a top-down design methodology in SystemC was presented. In order to demonstrate the effectiveness of these refinement rules, a case study of a digital camera as a target system was performed. In this chapter an overview of the digital camera system is presented followed by some of the details on the implementations and refinements of the SystemC models.

5.1 Digital Camera

The design used in this case study is based on the digital camera example presented in detail by Vahid and Givargis in Chapter 7 of [11]. The functionality of the digital camera system can be divided into two main tasks. First, the digital camera must capture, process, and store images into an internal memory. This task is initiated when the user presses the shutter button to take a picture. The image is captured in a digital form by a charge-coupled device (CCD). The image is then compressed using the Huffman compression algorithm before being stored into the internal memory of the digital camera. Second, the digital camera must be able to upload the stored images to a personal computer. A command is sent from the personal computer to the digital camera which instructs the camera to upload an image through a serial connection.

Based on these two tasks, an informal functional specification of the digital camera is shown in Figure 5.1 [11]. The functions associated with the task of image capture are shown on the left and the functions associated with serial transmission are shown on the right. After the image is captured into a digital form from the CDD, zero bias adjustment is performed to mathematically correct any errors associated with the image. The corrected image is then compressed, which consists of two steps: the application of the discrete cosine transform (DCT) and quantization. Finally the image is stored into the internal memory of the camera.

For the serial transmission task, the image is transferred one bit at a time through the serial connection.

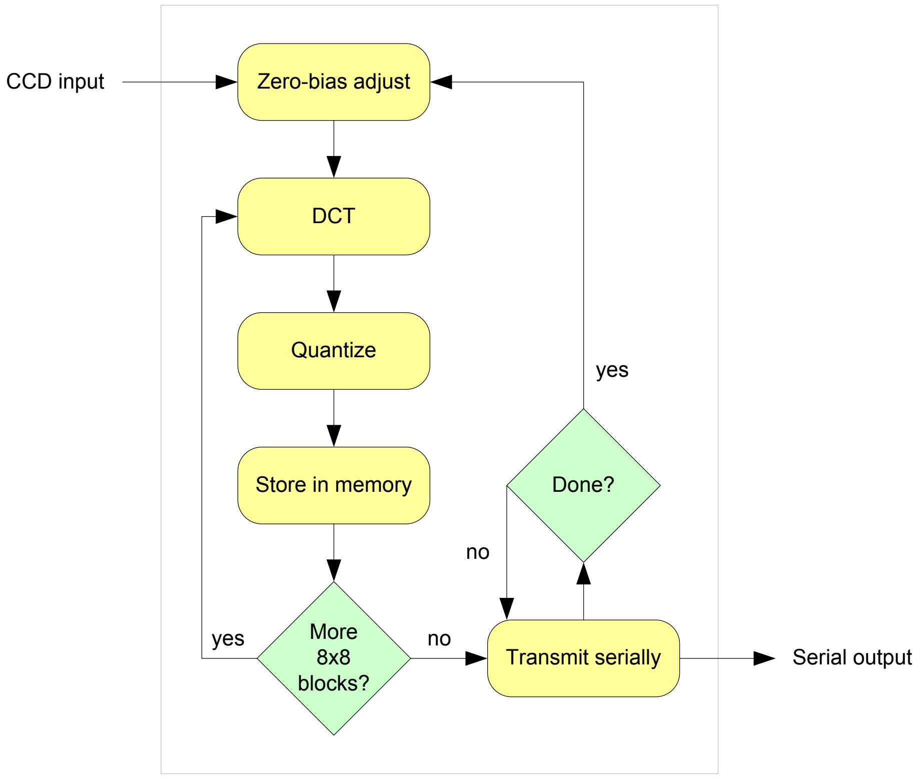


Figure 5.1 Functional block diagram of a digital camera

To obtain an executable functional specification, the informal functional specification shown in Figure 5.1 [11] is partitioned into five discrete modules: CCD, CCDPP, CODEC, UART, and CNTRL. These modules are separate executable parts of the system that could be modeled using the highest level of abstraction in a SLDL.

The CCD module is used to simulate the actions that an actual CCD would perform. Most notably it simulates the capture of an image and the transmission of pixels from the CCD. The CCDPP module is responsible for performing the zero-bias adjustment on each pixel as they are being sent by the CCD module. The CODEC module applies the Huffman encoding algorithm to the image by performing the DCT and quantization functions. The CNTRL module serves as the controller of the system, instructing each module as to what function to perform next. The UART models the serial transfer capability by sending the image byte by byte to an output file. A block diagram representing the executable model is shown in Figure 5.2 [11].

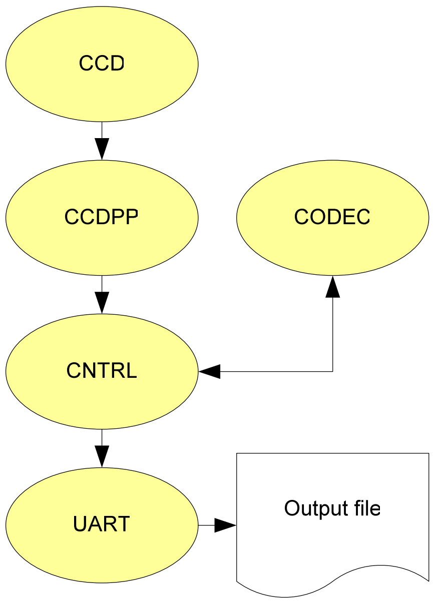


Figure 5.2 Block diagram of the executable model of the digital camera

Based on this diagram, the execution flow of the digital camera becomes apparent. The CCD module captures and sends the pixels of the image to the CCDPP module first. The CCDPP module then processes the pixels before they are sent to the CNTRL module. The CNTRL module uploads the pixels to the CODEC module to be processed before they are sent back to the CNTRL module. The CNTRL module then sends the pixels to the UART module where they are sent to an output file.

5.2 Software Prototype of the Digital Camera

The purpose of the digital camera example in [11] was to demonstrate the usefulness of codesign for embedded systems, where experimenting with different implementations of a system can result in variations in performance, power consumption, design complexity, and cost of the final implementation. In addition to an informal specification of the system, a software prototype was developed [12]. In this prototype, the implementation of the digital camera has been modified so that it operates on 16-bit images that are 64 x 64 pixels in size. Although this would result in an extremely low quality image, the prototype is only meant to demonstrate functionality and could be expanded to operate on images that are larger in size. In this section each module implemented in the software prototype is discussed. Each module has an initialize function which sets up any necessary conditions or variables prior to execution. Although these functions are not discussed in detail, they can be found in the reference code [12].

5.2.1 Prototype CCD Module

The CCD module simulates the behavior of a real CCD. This is accomplished by two functions, CcdCapture and CcdPopPixel. When CcdCapture is called, the pixels of an image are read from an input file and loaded into memory. When a real CCD is instructed to capture an image, it would read the value of each pixel and load it into a local memory. When the capture is finished, CcdPopPixel is called to transfer each pixel to the CCDPP module, one byte at a time. A functional block diagram of the CCD module is shown in Figure 5.3.

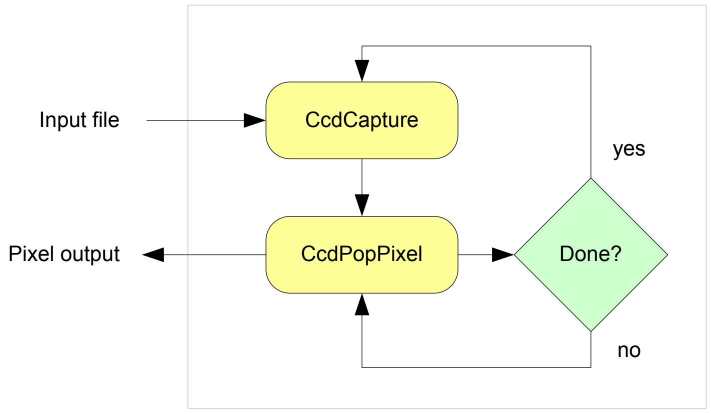


Figure 5.3 Functional block diagram specification of the CCD module

5.2.2 Prototype CCDPP Module

The CCDPP module is responsible for getting image data from the CCD and applying the zero-bias adjustments on each pixel. The image is then sent to the CNTRL module byte by byte. Two functions make up the CCDPP module: CcdppCapture and CcdppPopPixel. CcdppCapture calls the CcdCapture function first and then starts to collect each pixel from the CcdModule by calling the CcdPixelPop function. As each pixel is read from the CCD module, the zero-bias adjustment is applied and the pixel is stored. When CcdppPopPixel is called by the CNTRL module, each adjusted pixel is sent to the CNTRL module. The high-level functionality of the CCDPP module is shown in Figure 5.4.

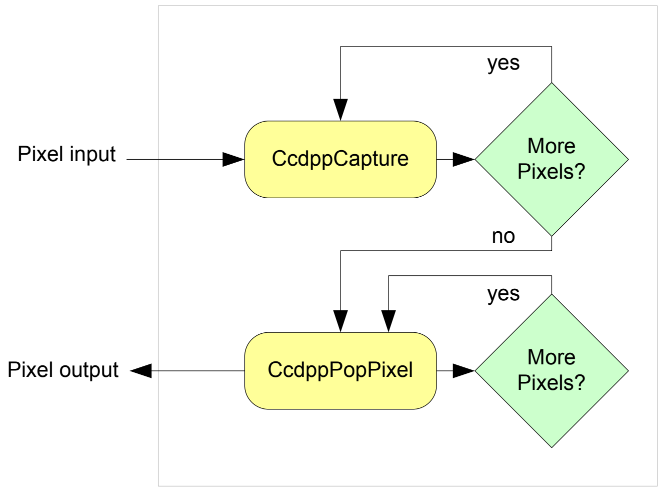


Figure 5.4 Functional block diagram specification of the CCDPP module

5.2.3 Prototype CODEC Module

The next module presented is the CODEC module. The CODEC module applies the DCT algorithm, the first half of the compression process, on 8 x 8 pixel blocks of the image. The CODEC module consists of three functions: CodecPushPixel, CodecDoFdct, and CodecPopPixel. CodecPushPixel collects the pixels from the CNTRL module until an 8 x 8 block of pixels has been read. The CodecDoFdct function then applies the DCT algorithm to the 8 x 8 block. Finally the data is sent back to the CNTRL module using the CodecPopPixel function. The high-level functionality of the CODEC module is shown in Figure 5.5.

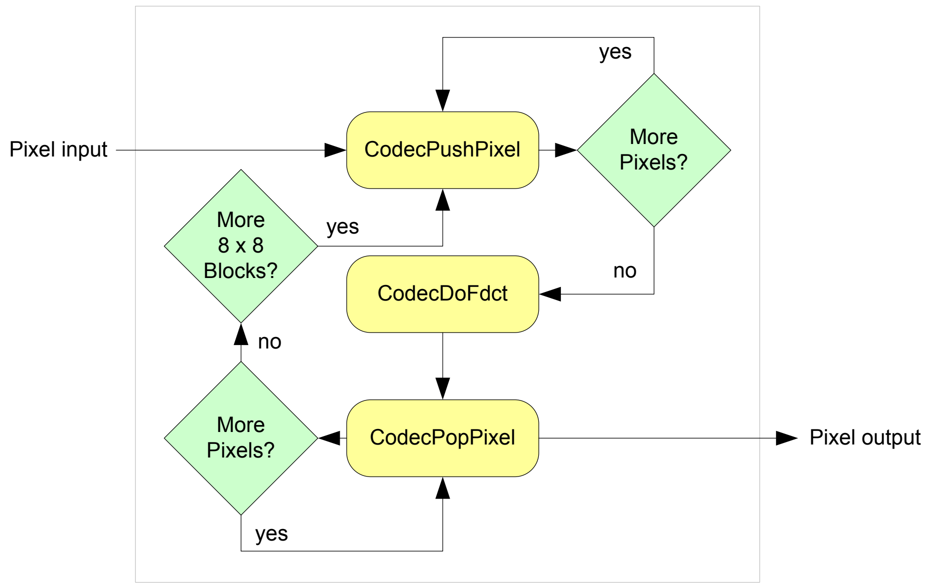


Figure 5.5 Functional block diagram specification of the CODEC module

5.2.4 Prototype UART Module

The UART module is used to replicate the functionality of a serial connection. On a real digital camera, the UART would be used to download the image from the camera to a PC. This is handled by the UartSend procedure, which receives data from the CNTRL module and writes it to an output file for verification purposes. The high-level functionality of the UART module is shown in Figure 5.6.

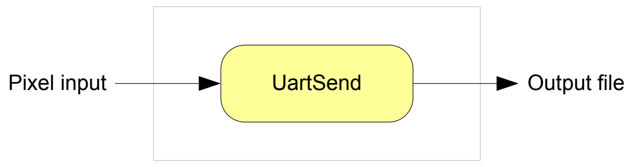


Figure 5.6 Functional block diagram specification of the UART module

5.2.5 Prototype CNTRL Module

The final module in the software prototype is the CNTRL module. This module controls all of the other modules functions by calling them in the correct sequence. The first function, CntrlCaptureImage, calls the CcdppCapture function to initiate the CCD capture and then collects the image byte by byte through the CcdppPopPixel function. The CntrlCompressImage sends the image in 8 x 8 pixel blocks to the CODEC module so that the FDCT algorithm is applied to the 8 x 8 block. When the block is returned to the CNTRL module, quantization is performed on the block before the next block is sent. Once all of the blocks have been compressed, the CntrlSendImage function is called to send the image to the UART module. The high-level functionality of the CNTRL module is shown in Figure 5.7.

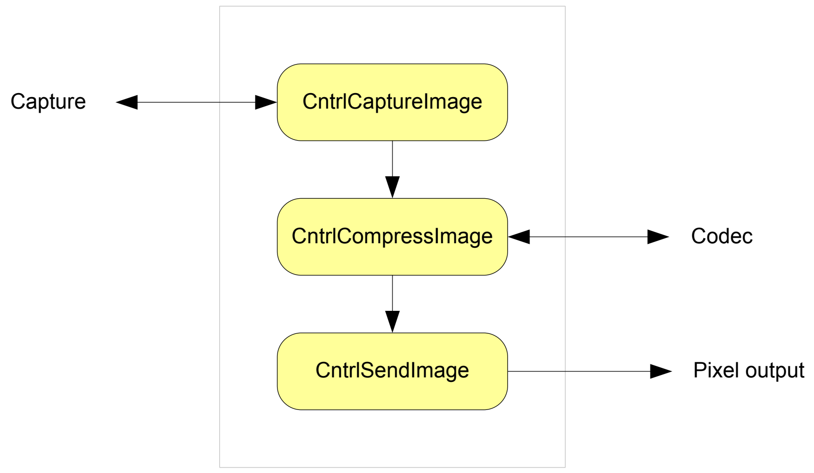


Figure 5.7 Functional block diagram specification of the CNTRL module

5.2.6 System-level Models of the Digital Camera

The software prototype is an executable specification of the digital camera coded in C. In [11], Vahid and Givargis discuss four different prototype implementations of the digital camera to illustrate how decisions made during hardware-software codesign can lead to an optimal solution. Based on their nonfunctional constraints, none of the final designs was clearly the best choice. In the third implementation, the CCDPP and UART modules run on independent custom processors while the CNTRL and CODEC modules run on a single processor core. One performance optimization for this implementation was the change of the code for the CODEC to use fixed-point arithmetic as opposed to expensive floating point operations in software. In the SpecC and SystemC implementations of this case study, the models and design decisions followed this version of the digital camera system.

5.3 SpecC Models of the Digital Camera System

In order to prove the effectiveness of the SystemC refinement rules presented in Chapter 4, the digital camera system was also modeled in SpecC to serve as a comparison. The functional model was implemented based on the software prototype described in section 5.2 and the other models were derived using the refinement rules from the SpecC methodology [7]. Although the details regarding the creation and refinement of the SpecC models will not be discussed, the source code for the functional, transaction-level, and communication models are included in Appendix A for reference.

5.4 Modeling and Refinement in SystemC

In this section the details of the implementation and refinement of the models of computation supported by SystemC are discussed. The functional model was developed based on the software prototype described in section 5.2 and the transaction-level and communication models were derived using the refinement rules presented in Chapter 4.

5.4.1 The Digital Camera System as a Functional Model in SystemC

Inspired by the software prototype [12], a functional model of the digital camera system was developed. Much of the work in partitioning the functionality into blocks was based off of the division of functionality in the software prototype discussed in section 5.2. This model distributes the behavior of the digital camera into five separate modules: CCD, CCDPP, CNTRL, CODEC, and UART. The original functions of the software prototype for the CCD, CCDPP, and UART modules were mapped directly to their respective processes. The three functions of the CODEC prototype were merged into a single process. The four functions of the CNTRL prototype were also merged into a single process. The code for the fdct process of the CODEC module was also converted from floating point to a fixed-point implementation of the DCT algorithm. A high-level representation of the functional model is shown in Figure 5.8. The modules are represented by rounded boxes while the processes are represented by the rectangles. Triggers for synchronization are represented by the arrows with dashed lines while transfers between modules involving data are represented by the arrows with the solid lines.

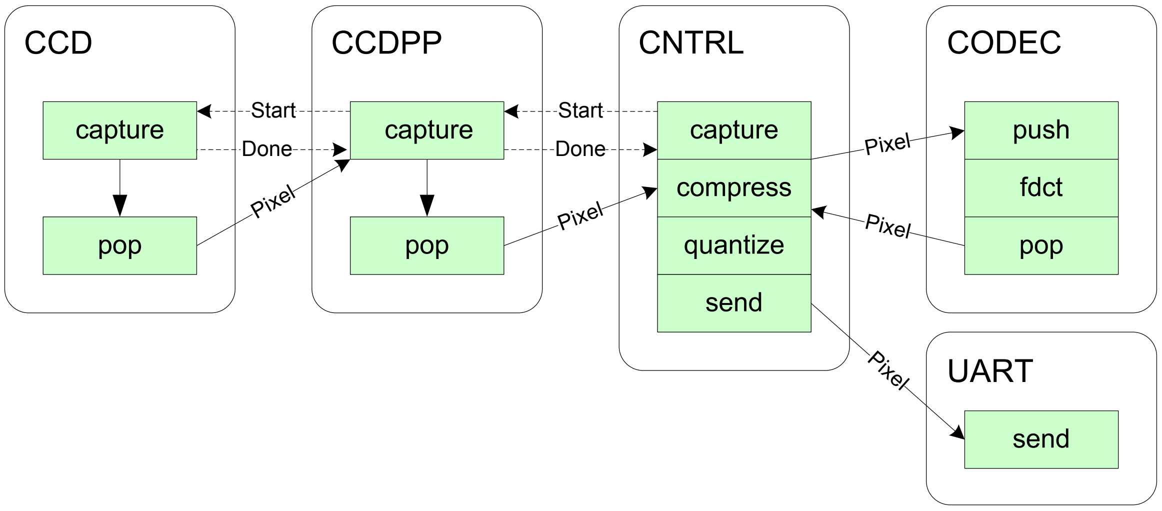


Figure 5.8 Functional model of the digital camera system

The initialization functions used in the software prototype are now handled by the module constructors. An example of the use of SC_CTOR in place of an initialization function is shown in Figure 5.9. The UART module opens the output file in the constructor before instantiating the thread process which writes to that file. When the simulation is finished, each module calls a destructor to clean up any remnants of the module. In this case, a destructor is specified as ~uart() and is responsible for closing the output file at the end of the simulation. Usually it is sufficient to use the default destructor for a module and in that case it does not need to be specified.

```
SC_MODULE( Uart )
{
   // UART Ports
   sc_fifo_in<char>  DataIn;

   // UART Vars
   FILE  *outputFileHandle;
   char   data;

   // UART Processes
   void uartSend(void)
   {
      while(1) {
         data = DataIn.read();
         fprintf(outputFileHandle, "%i\n", (int)data);
      }
   }

   // Module Constructor
   SC_CTOR( Uart )
   {
      outputFileHandle = fopen("uart_out.txt", "w");
      SC_THREAD( uartSend );
   }

   // Module Destructor
   ~Uart(void)
   {
      fclose(outputFileHandle);
   }
};
```

Figure 5.9 SystemC code for a functional model of the UART module

All of the synchronization and communication was done using the primitive channel, sc_fifo. As shown in Figure 5.9, the UART module has one input port that accepts characters. When using sc_fifo to transfer data, any supported data type can be specified and used. Each time the send process executes the read method it first checks to see if there are any characters waiting in the FIFO. If there is a character waiting, the character is read and the process continues executing the next line of code. If there is no character waiting, then the process halts execution until a character has been written to the FIFO. In this case the

FIFO acts as both a synchronization signal and data transfer channel. For synchronization without data transfer, boolean data types were used. The code excerpt from the CCDPP module shown in Figure 5.10 demonstrates the use of boolean data types for synchronization.

```
SC_MODULE( ccdpp )
{
   // ports
   sc_fifo_in<bool>     startCcdppCapture, doneCcdCapture;
   sc_fifo_in<char>     ccdPixel;
   sc_fifo_out<bool>    doneCcdppCapture, startCcdCapture;
   sc_fifo_out<char>    ccdppPixel;
...
   void capture(void)
   {
      StartCcdppCapture.read();
      StartCcdCapture.write( true );
      DoneCcdCapture.read();

      for(row=0; row<ROW_SIZE; row++) {
         for(col=0; col<COL_SIZE; col++) {
            buffer[row][col] = CcdPixel.read();
         }

         bias = CcdPixel.read();
         bias = ( bias + CcdPixel.read() ) / 2;
         for(col=0; col<COL_SIZE; col++) {
            buffer[row][col] -= bias;
         }
      }

      DoneCcdppCapture.write( true );
      notify(StartPopEvt);
   }
...
```

Figure 5.10 Capture process of the functional model CCDPP module

The simulation is started by calling the statement sc_start(-1). In this case, the argument specifies the length of simulation time to be a value of -1, which indicates to the simulator that it should execute the model in zero simulation time. There are two possibilities for stopping the simulation. One option is to call sc_stop() when a termination condition has been reached. The other option for is to stall the simulation when finished. This condition could occur intentionally or unintentionally. Once the simulation has stalled, events stop happening and the simulator automatically stops because it assumes the system is either deadlocked or has finished executing.

In the functional model of the digital camera, an exit condition was defined so the simulation could terminate normally. The CNTRL module is responsible for controlling the rest of the modules so it was a natural choice to place the exit condition in the CNTRL module. Notice that the send process in the UART module in Figure 5.9 is placed in an infinite loop. The UART module will continue to run infinitely throughout the simulation and will always be ready to receive more data. All of the modules except the CNTRL module have their final processes run in an infinite loop. They are all dependent on the CNTRL module to instruct them when to execute. After the image has been sent to the UART module, the CNTRL module has no other work to do. Calling sc_stop() at this point would stop the execution correctly.

5.4.2 The Digital Camera System as a Transaction-level Model in SystemC

The first step in refining the functional model of the digital camera into the transaction-level model is to convert it from the untimed domain to the timed domain. In order to add time delays to the functional model of the digital camera system, some estimation had to be done. The purpose of annotating delays is to make the functional model approximate-timed in terms of computation and communication time. Based on each piece of computation, a rough number of clock cycles were estimated and entered in a wait function call at the end of each annotated computation. In order to make this as simple as possible, the clock cycle time was defined as a constant value and each delay was computed as the number of estimated clock cycles multiplied by the clock cycle time, as shown in Figure 5.11.

The inserted wait function calls after computations and communication reads and writes have been highlighted. For the transaction-level model, both the approximated computation and communication delays are specified. This process also calls the sc_simulation_time function, which returns the current simulation time at the point in the code in which the function is called. This function is useful in determining how much time it took for a process to perform a task which allows for early evaluation of the performance of the system.

```
#define  clk_cycle  1
.
.
.
   void capture(void)
   {
      MainBus->ccdpp_ready();

      simTime = sc_simulation_time();

      CcdBus->start();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] = CcdBus->read();
         }

         // Perform Zero Bias Adjustment
         bias = CcdBus->read();
         bias = ( bias + CcdBus->read() ) / 2;
         wait(12*CLK_CYCLE, SC_NS);
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] -= bias;
            wait(4*CLK_CYCLE, SC_NS);
            MainBus->write(row*COL_SIZE+col, buffer[col]);
         }
      }

      MainBus->ccdpp_done();

      cout << "CCDPP\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }
.
.
.
```

Figure 5.11 Time-annotated capture process of the CCDPP module

The next step was to allocate processing elements for the system. In order to determine the number of processing elements needed, the details of the third implementation of the digital camera example [11] were followed. In this implementation, the CCDPP and UART modules are implemented as custom processors, so each require their own processing elements. The CNTRL and CODEC modules are both software components executed on a microprocessor, so they will coexist on the same processing element. Finally, the CCD module is a device that is independent of the rest of the system, so it will be mapped to its own processing element.

The only processing element with multiple modules is the CNTRL/CODEC processing element. These modules are dependent upon one another to function properly and since they share the same processing element they will be executed sequentially on the target system. Due to the sequential execution of these two modules, their processes can be merged to form a single process and eliminate unnecessary scheduling. The merging of the CNTRL and CODEC processes is shown in Figure 5.12.

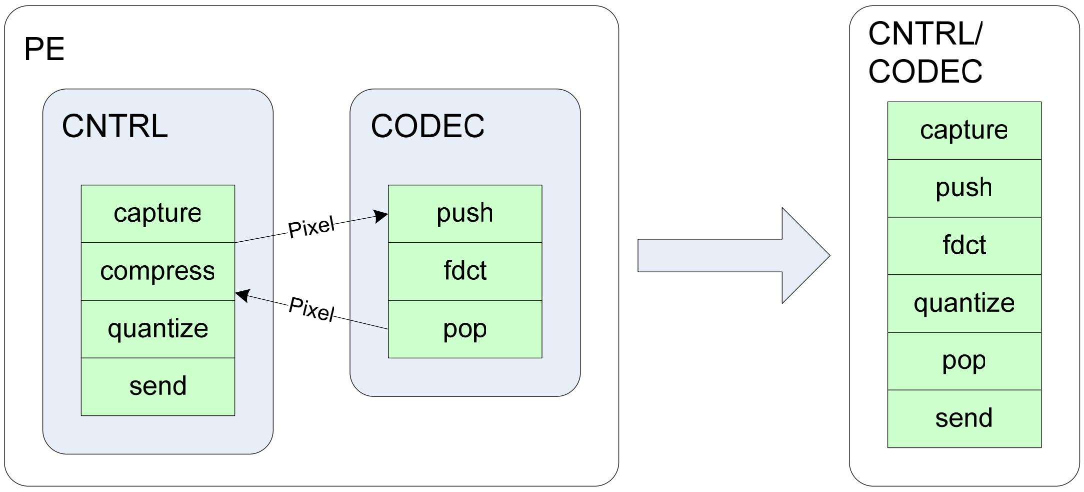


Figure 5.12 Merging of CNTRL and CODEC modules

Since the image data is fairly large in size, a shared memory module was also added to the system. The CCDPP, CNTRL, and UART processing elements all have access to the shared memory module. To prevent contention caused by two or more modules accessing the memory at the same time, some additional synchronization events were added to indicate if a particular module is busy accessing memory. Finally, data and address channels are created so that the CCDPP, CNTRL, and UART modules can access the shared memory. The digital camera model after the architecture partitioning phase of transaction-level model refinement is shown in Figure 5.13.

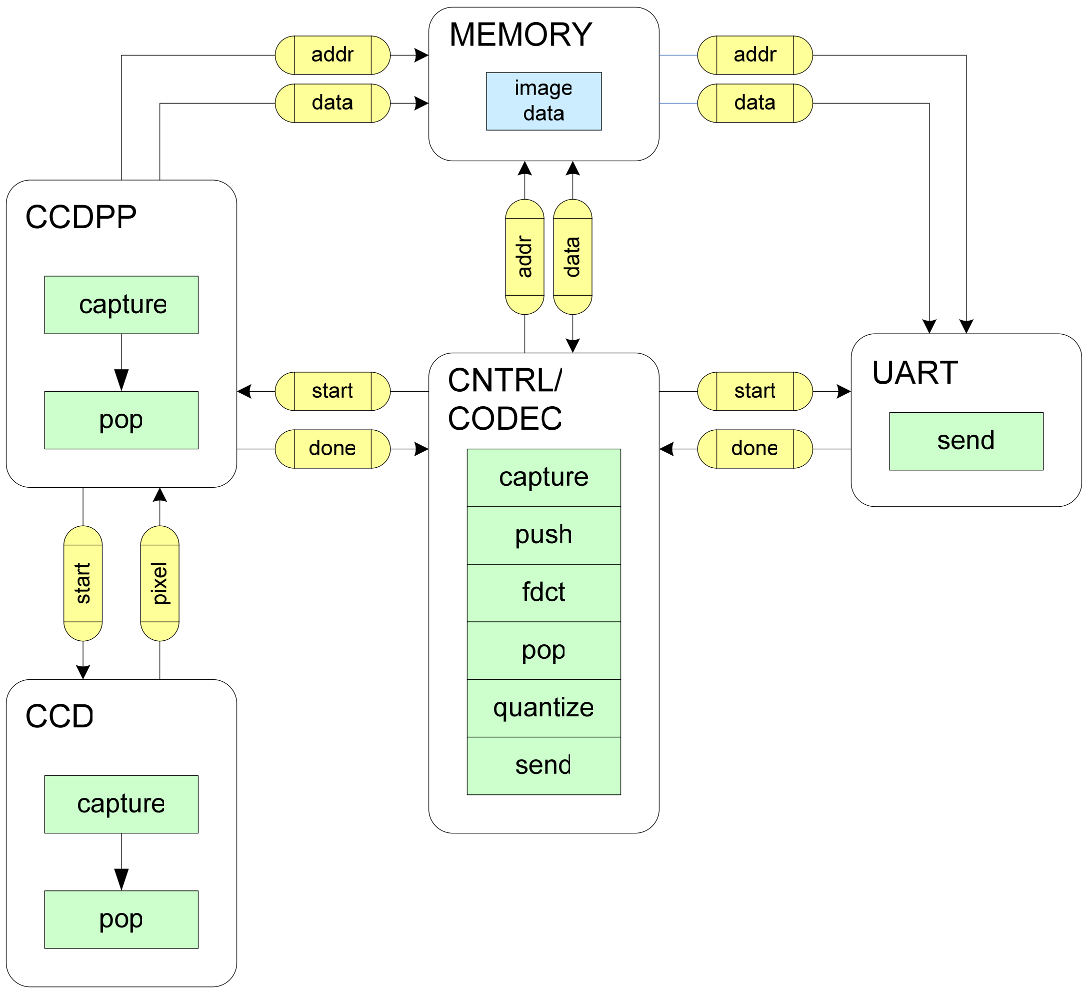


Figure 5.13 Digital camera after the architecture partitioning phase

Following the architecture partitioning phase of transaction-level model refinement, the next step was to apply the rules of the communication partitioning phase. In the digital camera, two busses were formed from the existing global channels. The main bus connects the shared memory, CNTRL/CODEC, CCDPP, and UART modules. The data and address channels were combined with the synchronization channels of the CCDPP and UART. The CCD bus groups the global channels used between the CCD and CCDPP modules. The ports of each module were modified to connect to their respective interface and the port accesses inside the processes were changed to inline function calls. A portion of the code that implements the interface functions of the CCD bus is shown in Figure 5.14.

```
class CcdBus:  public sc_module,
               public CcdToCcdBusIf,
               public CcdppToCcdBusIf
{
private:
   char     pixel;
   bool     valid, busy;
   sc_event StartEvt, ValidEvt;
.
.
.
   // CcdppToCcdBusIf Interface Functions
   void CcdBus::start()
   {
      do{
         wait( 1*CLK_CYCLE, SC_NS );
      }while(busy == true);
      notify(StartEvt);
      return;
   }

   char CcdBus::read()
   {
      if(!valid)
         wait(ValidEvt);
      valid = false;
      return pixel;
   }
.
.
.
```

Figure 5.14 CCD bus interface function definitions for the CCDPP module

Note that the variables that are transferred over the channel are encapsulated within the interface definition as private variables. Thus, the communication between processing elements is simplified because accessing the channel is reduced to the matter of making a function call, without knowledge of the details regarding the implementation of the channel. The interfaces for the main bus were implemented in the same fashion. The transaction-level model of the digital camera following the communication partitioning phase is shown in Figure 5.15.

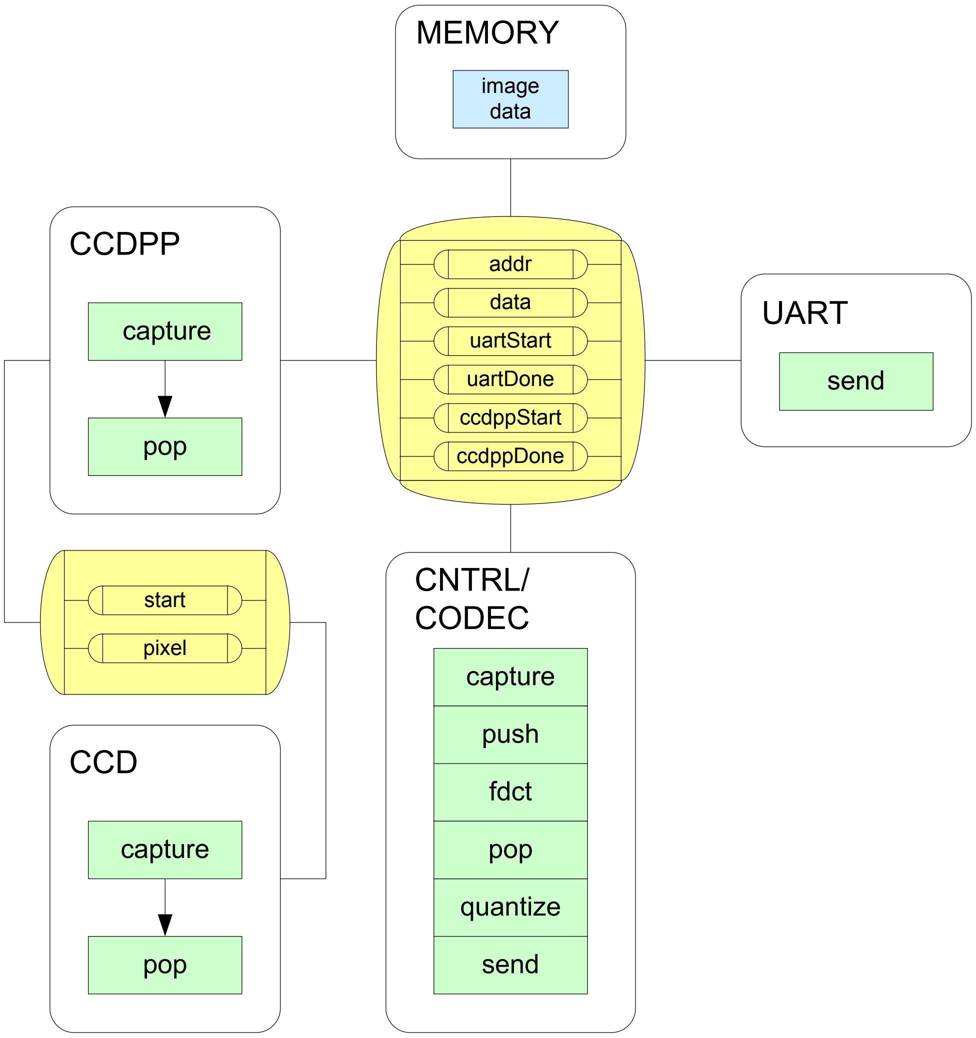


Figure 5.15 Transaction-level model of the digital camera

5.4.3 The Digital Camera System as a Communication Model in SystemC

In the transaction-level model of the digital camera system, approximate models of the processing elements and communication busses were implemented. When applying the refinement rules to obtain the communication model, the first step was to replace communication channels with pin-level implementations of the bus in the form of wires and adapters to preserve the ability to use the interface functions. While the variables inside the busses were changed to wires, the actual protocol is implemented by the interface functions inside the adapters. As an example, a diagram of the converted CCD bus is shown in Figure 5.16.

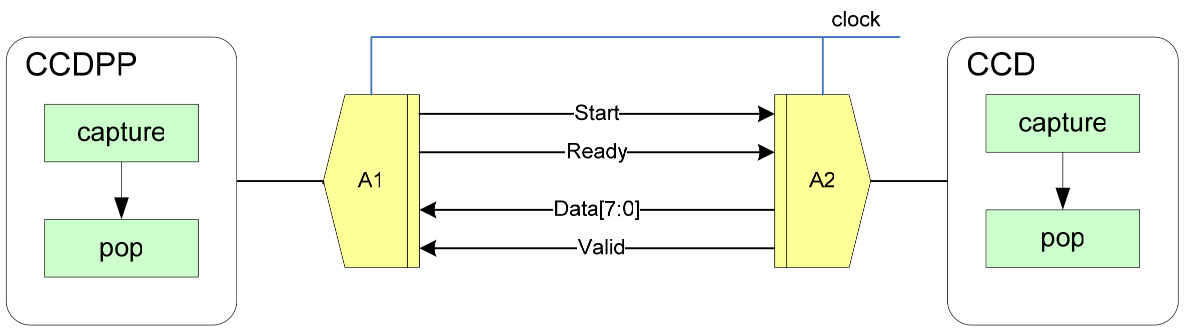


Figure 5.16 CCD bus implemented with protocol adapters

Since the interfaces were introduced in the transaction-level model, the CCD and CCDPP processing elements can access the newly refined communication channel without any significant changes. However the definitions of the interface functions contained within the adapter modules must be changed to reflect the protocol used to perform the communication correctly over the pin-level implementation of the bus. The interface functions implement the protocol by asserting and analyzing the data transferred on the wires. In the case of the CCD bus, a simple handshaking protocol was introduced to synchronize the data transfers. The modified code of the interface functions for the adapter that connects the CCDPP module to the CCD bus is shown in Figure 5.17.

```
class CcdppToCcdBusAdapter:  public sc_module,
                             public CcdppToCcdBusIf
{
private:
   char  temp;

public:
   sc_in<bool>    ClockI;
   sc_in<char>    DataI;
   sc_in<bool>    ValidI;
   sc_out<bool>   StartO;
   sc_out<bool>   ReadyO;

   void CcdppToCcdBusAdapter::start( void )
   {
      StartO.write(TRUE);
      wait( ClockI->posedge_event() );
   }

   char CcdppToCcdBusAdapter::read( void )
   {
      ReadyO.write( TRUE );
      do {
         wait( ClockI->posedge_event() );
      }while( ValidI.read() != TRUE );

      ReadyO.write( FALSE );
      temp = DataI.read();
      wait( ClockI->posedge_event() );

      return temp;
   }

   SC_CTOR(CcdppToCcdBusAdapter) {
      ReadyO.initialize( false );
   }
};
```

Figure 5.17 Updated CCD bus interface functions for the CCDPP adapter

The main bus that connects the shared memory module, CNTRL, UART, and CCDPP is refined in a similar fashion, although it is more complicated. The target implementation of the main bus is a shared data bus with designated control wires designed to allow the CNTRL module to instruct the CCDPP and UART modules. The implementation of the shared memory is moved into its own module and the rest of the channel is converted into wires. A high-level diagram of the digital camera system after the hierarchical channels have been replaced with adapters and pin-level implementations of the busses is shown in Figure 5.18.

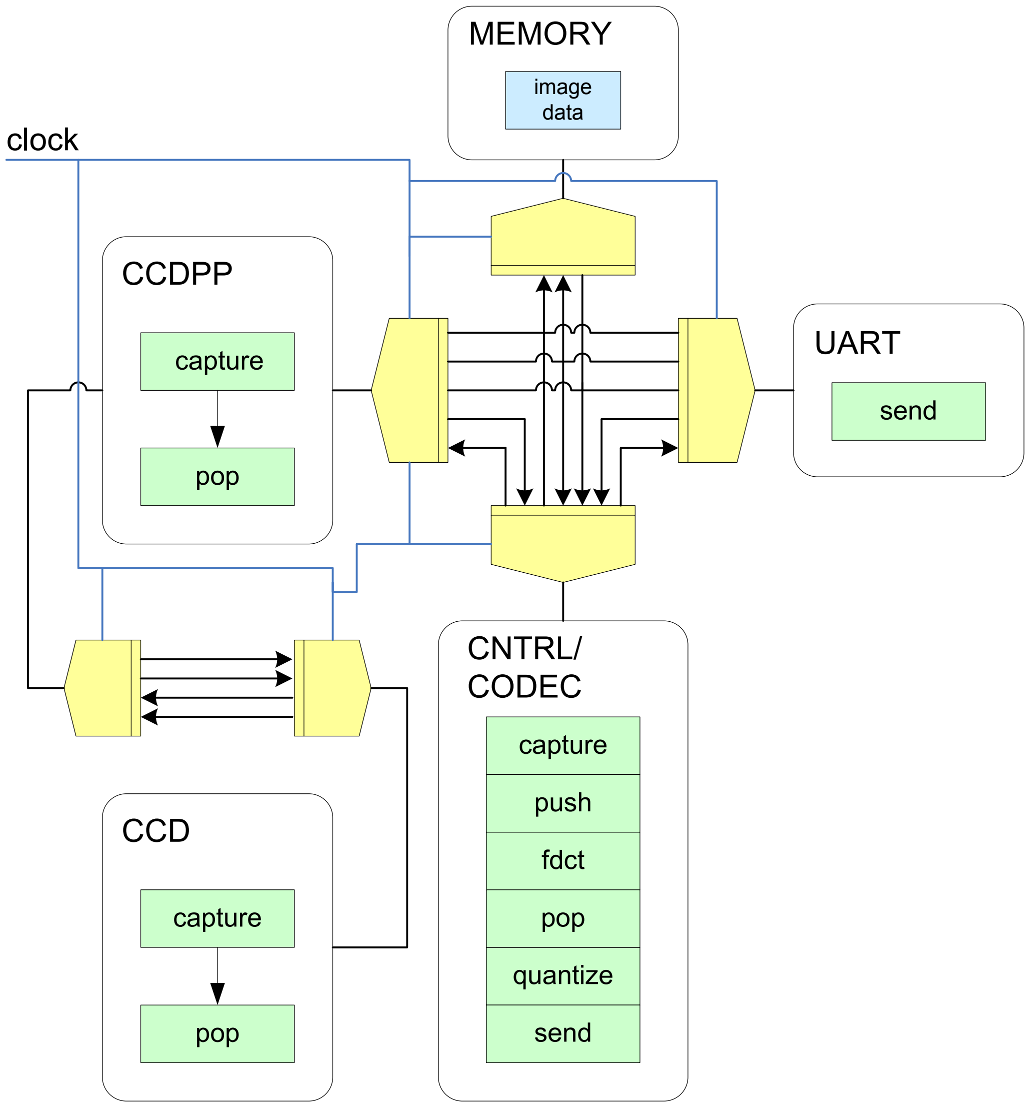


Figure 5.18 Digital camera after the adapter synthesis phase

Next, the protocol insertion phase takes the protocols and function calls contained within the adapters and moves them into their respective processing elements. The pin-level interfaces defined by the adapters are moved into the processing elements. The interface functions used in the adapters are inserted into the processing elements. Finally the interface function calls are replaced by local function calls. Once this phase is completed, the model of the digital camera system consists of abstracted computational units connected through actual ports and wires of busses in the target implementation of the system. The communication model of the digital camera system is shown in Figure 5.19.

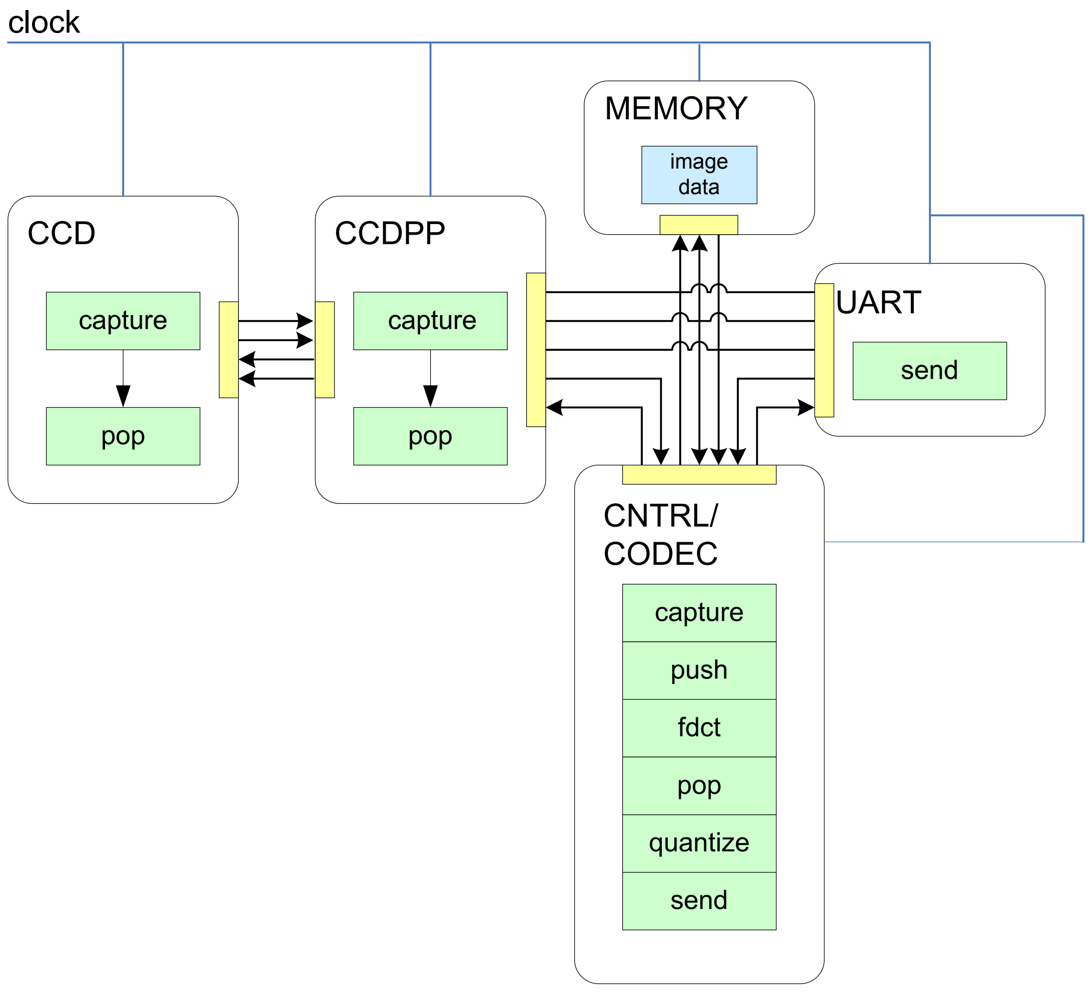


Figure 5.19 Communication model of the digital camera system

5.4.4 The Digital Camera System as an Implementation Model in SystemC

The implementation of the digital camera goes beyond the scope of this case study. Although the goal of SLD is to develop implementations by deriving high-level models, the case study served as a practical approach to applying the refinement rules presented in

Chapter 4. However, given the SystemC source code of the digital camera in Appendix B, an implementation could be derived.

CHAPTER 6 RELATED WORK

This chapter provides an overview of other research that addresses the development and use of SLDLs as well as SLD methodologies.

The Ptolemy Project [13] consists of a research group at U.C. Berkeley that focuses on modeling, simulating, and designing embedded systems. The result of their research is the open source software design environment. Like SystemC, earlier generations of the tool took an object-oriented approach by modeling various components of the system in C++. Additional C++ classes were developed to cover many characteristics of embedded system design including communication strategies, simulation, hardware-software codesign, and parallel computing [14]. In order to be useful in modeling components with wildly different purposes, Ptolemy supported many different models of computation. The significant contribution made by the Ptolemy Project was the ability to simulate systems consisting of a mixture of behavioral, hardware, and software components simultaneously, paving the way for cosimulation of embedded systems. The current generation of the tool, Ptolemy II, moved from C++ to Java in order to take advantage of the inherent support of threading, web integration, and graphical user-interface capabilities [15]. A key development in Ptolemy II over previous generations of the tool is the inclusion of modeling support for embedded software, which has more constraints due to heavy interaction with the hardware. The open source nature of Ptolemy II has resulted in the development of a number of additional frameworks libraries to allow the modeling of more specialized systems such as wireless sensor networks [16] and image processing [17].

The use of finite state machines (FSMs) to represent embedded systems at a high-level was addressed by the POLIS [18] system. In POLIS, each component of a system is modeled using specialized FSMs, called codesign finite state machines (CFSMs). A CFSM is considered to be globally asynchronous but locally synchronous FSM. The POLIS system utilizes a top-down design methodology where the system is first written in Esterel [19], a high-level synchronous programming language that supports CFSMs. The functional implementation is then tested using the VIS [20] verification and synthesis system. Next, the architectural decisions are made and the design is partitioned into hardware and software components. POLIS uses tools such as Ptolemy for cosimulation at the high-level using CFSMs as well as the implementation-level model of the system. The key contribution by POLIS was the separation of functionality and architecture during the design process. This idea later served as a basis for the approach to a commercial SLD tool, Cadence Virtual Component Codesign (VCC).

The Metropolis project represents the evolution of the POLIS tool into unified design environment and formal design methodology [21]. The issue that Metropolis attempts to address is that currently system-level designers must use a variety of different tools, which may use different file formats or languages, in order to take a design from a functional specification to a final implementation of the system. This can make debugging difficult, particularly if errors are injected during translation by the tools themselves. Like POLIS, modeling of components is done using an extension of the Java programming language. The Metropolis modeling library contains many of the same core elements found in SLDLs like SystemC and SpecC. Like SystemC, the Metropolis environment does not impose one particular refinement methodology. Densmore presents an overview of several different refinement methodologies [22] that may be applied to systems modeled in Metropolis, with the addition of application-specific methodologies planned in the future.

A heterogeneous design methodology using SpecC, SystemC, and Cadence VCC was introduced by L. Cai et al. in [23]. In this approach, SpecC is used to model the system at the higher levels of abstraction while VCC is used for architectural exploration. At lower levels, the implementation model is derived from the SpecC and VCC models, with the software components modeled in C and the hardware components modeled in SystemC.

Unified Modeling Language (UML) has been applied in the software engineering field for years, but has been gaining acceptance in assisting in the design SOC systems as well. The OWL project [24] used UML as a tool to model the system at high levels of abstraction before modeling the system in SystemC. Their approach of using UML for defining the system requirements and documenting the specifications assisted in overcoming the lack of refinement methodology for SystemC. The recent interest in applying UML to SLD has brought forth the inception of UML 2, which features more modeling capabilities suitable for SLD. Riccobene et al. present a design methodology using UML 2 to develop structural and behavioral models at a high-level with translation to SystemC for modeling at lower levels [25]. The Systems Modeling Language (SysML) initiative is also working towards extending UML 2 to better support system-level modeling capabilities [26].

Work in the automation of some of the more difficult parts of SLD refinement has also produced some interesting results. Abdi, Shin, and Gajski have proposed a methodology and algorithms to assist in automating the process of communication refinement [27]. They also present a methodology in automating the refinement of transaction-level models [28]. Lyonnard et al. presents a design flow to connect heterogeneous processing components using automatically generated communication components [29]. Passerone, Rowson, and Sangiovanni-Vincentelli present an algorithm used to automatically generate interfaces for components that use incompatible protocols [30]. In [31], Baleani et al. proposes a reconfigurable architecture platform that uses the hardware-software codesign methodology from POLIS to generate implementations.

Automatic software code generation from SLDLs has also been an active research topic. In [32], a methodology for generating embedded software and interfaces from SystemC models is proposed. The proposed methodology accomplishes this by redefining and overloading the SystemC class elements to generate new code that runs on a real-time operating system and is functionally equivalent. A similar approach is described in [33], but the proposed methodology is designed to be more universal. The implementation of the methodology in [33] is in the form of an automation tool that is targeted for generating ANSI C code from SpecC models, but the concepts could be applied to SystemC as well.

This work is differentiated from other work in that it takes a homogenous approach by using SystemC exclusively throughout the SLD process, from specification to implementation. SystemC possesses models of computation that allow for a SLD methodology. Although the SystemC models of computation are already known, a top-down refinement methodology for SystemC had not yet been presented.

CHAPTER 7 CONCLUSION AND FUTURE WORK

7.1 Concluding Remarks

As the complexity of embedded systems has increased in recent years, the need for flexible SLDLs to manage designs at different layers of abstraction has become a necessity. SLDLs need to be able to model both the hardware and software aspects of an embedded system. Using SLDLs such as SystemC or SpecC, systems can be modeled at layers of abstraction varying from the functional specification to the target implementation.

The modeling of an embedded system at all layers of abstraction supported by SystemC has been described and demonstrated with the development of a digital camera system. Guidelines for developing a functional specification and refinement rules for transforming the specification to the target implementation has also been presented. These guidelines and refinement rules were applied to the digital camera system to demonstrate the differences between the models of computation and explain the details of applying some of the steps in the refinement process. The digital camera system was refined and verified through each step in the design process, validating the effectiveness of the proposed top-down design methodology.

7.2 Recommendations for Future Work

Capturing the functionality of the system in the initial functional specification is one of the most important steps in a top-down design methodology. After each refinement stage, more details about the target implementation are defined and the more difficult it becomes to make major changes to the system later in the design process. In order to make those design decisions, designers still need to have a working knowledge of how to accurately estimate timing delays in abstract models. One possibility for future research would be a method for quantifying useful approximate computation and communication delays for transaction level models in SystemC. Other applicable research that could aid in the making of informed design decisions early on would be a method of generating and gathering performance analysis information from abstract SystemC models.

Another issue that was acknowledged is that the current features of SystemC are tuned toward the ability to model and develop hardware aspects of the system but are limited in the ways software can be modeled at later layers of abstraction. The lack of ability to schedule tasks and emulate real-time operating systems needs to be addressed in order for SystemC to be considered a SLDL that supports true hardware-software codesign. Task scheduling is rumored to be part of an upcoming version of the SystemC language, SystemC 3.0. The refinement rules presented in this thesis would likely need to be modified and added to when task scheduling and other new modeling features are added to SystemC.

The process of converting an implementation-level SystemC or SpecC model into the system's final implementation is vague. One area of future work lies in the translation of a system-level model into an actual implementation. Currently this process is done manually by the designer and is a tedious process, especially for larger designs. Automation of this process would greatly increase the efficiency of using SLDLs to design actual implementations.

Finally, OSCI has been making significant changes to the transaction-level modeling capabilities through an add-on library to SystemC 2.1. The library is currently incomplete at the time of this writing, but the rules presented in this thesis for refining transaction-level models may need to be modified in order to accommodate these new features.

APPENDIX A DIGITAL CAMERA SYSTEM: SPECC MODELS

This appendix contains the SpecC source code for the functional, transaction-level, and communication models of the digital camera system. Each model was compiled using the SpecC compiler 2.2.0 running on RedHat Linux. All model simulations were performed using the simulator included with SpecC compiler package.

SpecC Functional Model

```
/////////////////////////////////////////////////////////////////////
// File: digcam.sc                                                 //
// Desc: SpecC Functional Model of the Digital Camera              //
/////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include "image.h"

////////////////////////////////////////
// CCD Behaviors                      //
// for Functional model               //
////////////////////////////////////////
behavior CcdCapture( in event    StartCaptureEvt,
                     out char    buffer[ROW_SIZE][COL_SIZE],
                     out event   DoneCaptureEvt )
{
   int row, col;

   void main(void)
   {
      wait(StartCaptureEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
      notify(DoneCaptureEvt);
   }
};

behavior CcdPopPixel( in char    buffer[ROW_SIZE][COL_SIZE],
                      in event   PixelReqEvt,
                      out event  PixelSentEvt,
                      out char   pixel )
{
   int row, col;

   void main(void) {

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            wait(PixelReqEvt);
            pixel = buffer[row][col];
            notify(PixelSentEvt);
         }
      }
   }
};

behavior Ccd(in event   StartCcdCaptureEvt,
             in event   PixelReqEvt,
             out event  DoneCcdCaptureEvt,
             out event  PixelSentEvt,
             out char   pixel )
{
   char buffer[ROW_SIZE][COL_SIZE];

   CcdCapture  CcdCaptureInst( StartCcdCaptureEvt,
                               buffer,
                               DoneCcdCaptureEvt );

   CcdPopPixel CcdPopPixelInst( buffer,
                                PixelReqEvt,
                                PixelSentEvt,
                                pixel );

   void main(void) {
      CcdCaptureInst.main();
      CcdPopPixelInst.main();
   }
};


////////////////////////////////////////
// CCDPP behaviors                    //
// for Functional model               //
////////////////////////////////////////
behavior CcdppCapture( in event  StartCcdppCaptureEvt,
                       in event  DoneCcdCaptureEvt,
                       in event  CcdPixelSentEvt,
                       in char   ccdPixel,
                       out char  buffer[ROW_SIZE][COL_SIZE],
                       out event StartCcdCaptureEvt,
                       out event CcdPixelReqEvt,
                       out event DoneCcdppCaptureEvt )
{
   int row, col;
   char bias;
   char tempRow[COL_SIZE];

   void main(void)
   {
      wait(StartCcdppCaptureEvt);
      notify(StartCcdCaptureEvt);
      wait(DoneCcdCaptureEvt);

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            notify(CcdPixelReqEvt);
            wait(CcdPixelSentEvt);
            tempRow[col] = ccdPixel;
         }

         notify(CcdPixelReqEvt);
         wait(CcdPixelSentEvt);
         bias = ccdPixel;
         notify(CcdPixelReqEvt);
         wait(CcdPixelSentEvt);
         bias = (bias + ccdPixel) / 2;
         for(col=0; col<COL_SIZE; col++)
         {
            tempRow[col] -= bias;
            buffer[row][col] = tempRow[col];
         }
      }
      notify(DoneCcdppCaptureEvt);
   }
};

behavior CcdppPopPixel( in event PixelReqEvt,
                        in char  buffer[ROW_SIZE][COL_SIZE],
                        out event   PixelSentEvt,
                        out char    pixel )
{
   int row, col;

   void main(void)
   {
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            wait(PixelReqEvt);
            pixel = buffer[row][col];
            notify(PixelSentEvt );
         }
      }
   }
};

behavior Ccdpp( in event   StartCcdppCaptureEvt,
                in event   CcdppPixelReqEvt,
                out event  DoneCcdppCaptureEvt,
                out event  CcdppPixelSentEvt,
                out char   ccdppPixel,
                in event   DoneCcdCaptureEvt,
                in event   CcdPixelSentEvt,
                in char    ccdPixel,
                out event  StartCcdCaptureEvt,
                out event  CcdPixelReqEvt )
{
   char buffer[ROW_SIZE][COL_SIZE];

   CcdppCapture   CcdppCaptureInst( StartCcdppCaptureEvt,
                                    DoneCcdCaptureEvt,
                                    CcdPixelSentEvt,
                                    ccdPixel,
                                    buffer,
                                    StartCcdCaptureEvt,
                                    CcdPixelReqEvt,
                                    DoneCcdppCaptureEvt );

   CcdppPopPixel      CcdppPopPixelInst( CcdppPixelReqEvt,
                                         buffer,
                                         CcdppPixelSentEvt,
                                         ccdppPixel );

   void main(void)
   {
      CcdppCaptureInst.main();
      CcdppPopPixelInst.main();
   }
};

////////////////////////////////////////
// CODEC behaviors                    //
// for Functional model               //
////////////////////////////////////////
```

```
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12
};

behavior CodecPushPixel( in short   pixelIn,
                         in event   PixelInSentEvt,
                         out event  PixelInRecvEvt,
                         out short  buffer[8][8] )
{
   int i;
   int idx;

   void main(void) {
      i = 0;
      while(i < 128) {
         for(idx=0; idx<64; idx++)
         {
            wait(PixelInSentEvt);
            buffer[idx / 8][idx % 8] = pixelIn;
            notify(PixelInRecvEvt);
         }
         ++i;
      }
   }
};

behavior CodecDoFdct( inout short   buffer[8][8],
                      in event   StartFdctEvt,
                      out event  DoneFdctEvt )
{
   int x, y;
   int i;
   short tempBuffer[8][8];

   // FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {

      long  s[8];
      long  r;
      unsigned char  a;

       for(a=0; a<8; a++)
       {
        s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
               ((img[a][1] * COS_TABLE[8+v])  >> 6) +
               ((img[a][2] * COS_TABLE[16+v]) >> 6) +
               ((img[a][3] * COS_TABLE[24+v]) >> 6) +
               ((img[a][4] * COS_TABLE[32+v]) >> 6) +
               ((img[a][5] * COS_TABLE[40+v]) >> 6) +
               ((img[a][6] * COS_TABLE[48+v]) >> 6) +
               ((img[a][7] * COS_TABLE[56+v]) >> 6);
      }

      r = 0;
      for(a=0; a<8; a++) {
         r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
      }
      return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
   }

   // FDCT Behavior
   void main(void)
   {
      for(i=0; i < 128; i++)
      {
         wait(StartFdctEvt);

         for(x=0; x<8; x++)
         {
            for(y=0; y<8; y++)
            {
               tempBuffer[x][y] = F(x, y, buffer);
            }
         }
         for(x=0; x<8; x++)
         {
            for(y=0; y<8; y++)
            {
               buffer[x][y] = tempBuffer[x][y];
            }
         }

         notify(DoneFdctEvt);
      }
   }
};

behavior CodecPopPixel( in short    buffer[8][8],
                        in event    PixelOutReqEvt,
                        out short   pixelOut,
                        out event   PixelOutSentEvt )
{
   int i;
   int idx;

   void main(void) {
      i = 0;
      while(i<128)
      {
         for(idx=0; idx<64; idx++)
         {
            wait(PixelOutReqEvt);
            pixelOut = buffer[idx/8][idx%8];
            notify(PixelOutSentEvt);
         }
         ++i;
      }
   }
};

behavior Codec( in short   pixelIn,
                in event   PixelInSentEvt,
                in event   StartFdctEvt,
                in event   PixelOutReqEvt,
                out short  pixelOut,
                out event  PixelInRecvEvt,
                out event  DoneFdctEvt,
                out event  PixelOutSentEvt )
{
   short buffer[8][8];

   CodecPushPixel     CodecPushPixelInst( pixelIn,
                                          PixelInSentEvt,
                                          PixelInRecvEvt,
                                          Buffer );
   CodecDoFdct CodecDoFdctInst( buffer,
                                StartFdctEvt,
                                DoneFdctEvt );

   CodecPopPixel  CodecPopPixelInst( buffer,
                                     PixelOutReqEvt,
                                     pixelOut,
                                     PixelOutSentEvt );

   void main(void) {
      par {
         CodecPushPixelInst.main();
         CodecDoFdctInst.main();
         CodecPopPixelInst.main();
      }
   }
};

////////////////////////////////////////
// UART behaviors                     //
// for Functional model               //
////////////////////////////////////////
behavior UartInitialize( out FILE *outputFileHandle )
{
   void main(void) {
      outputFileHandle = fopen("uart_out.txt", "w");
   }
};

behavior UartSend( in FILE    *outputFileHandle,
                   in char    data,
                   in event   DataSentEvt,
                   out event  DataRecvEvt ) {
   int i;
   void main(void) {
      for(i=0; i < 16384; i++)
      {
         wait(DataSentEvt);
         fprintf(outputFileHandle, "%i\n", (int)data);
         notify(DataRecvEvt);
      }
   }
};

behavior Uart( in char     data,
               in event    DataSentEvt,
               out event   DataRecvEvt ) {

   FILE *outputFileHandle;

   UartInitialize     UartInitializeInst(outputFileHandle);

   UartSend    UartSendInst( outputFileHandle,
                             data,
                             DataSentEvt,
                             DataRecvEvt);

   void main(void) {
      UartInitializeInst.main();
      UartSendInst.main();
   }
};


////////////////////////////////////////
// CNTRL behaviors                    //
// for Functional model               //
////////////////////////////////////////
const unsigned char QUANT_SHIFT_TABLE[64] = {
```

```
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };

behavior CntrlCaptureImage( in event   DoneCaptureEvt,
                            in event   CcdppPixelSentEvt,
                            in char    ccdppPixel,
                            out short  buffer[ROW_SIZE][COL_SIZE],
                            out event  StartCaptureEvt,
                            out event  CcdppPixelReqEvt ) {
   void main(void) {
      int i, j;

      notify(StartCaptureEvt);
      wait(DoneCaptureEvt);

      for(i=0; i<ROW_SIZE; i++)
      {
         for(j=0; j<COL_SIZE; j++)
         {
            notify(CcdppPixelReqEvt);
            wait(CcdppPixelSentEvt);
            buffer[i][j] = ccdppPixel;
         }
      }
   }
};

behavior CntrlCompressImage( in event  CodecPixelPushRecvEvt,
                             in event  CodecPixelPopSentEvt,
                             in event  DoneFdctEvt,
                             in short  codecPixelPop,
                             inout short  buffer[ROW_SIZE][COL_SIZE],
                             out event CodecPixelPushSentEvt,
                             out event CodecPixelPopReqEvt,
                             out event StartFdctEvt,
                             out short codecPixelPush ) {

   void main(void) {
      int i, j, k, l;

      // CNTRL Compress
      for(i=0; i<NUM_ROW_BLOCKS; i++)
      {
         for(j=0; j<NUM_COL_BLOCKS; j++)
         {
            // Push the block and perform FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  codecPixelPush = (char)buffer[i*8 + k][j*8 + l];
                  notify(CodecPixelPushSentEvt);
                  wait(CodecPixelPushRecvEvt);
               }
            }
            notify(StartFdctEvt);
            wait(DoneFdctEvt);

            // Pop the block and store it in buffer
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
```

```
               notify(CodecPixelPopReqEvt);
               wait(CodecPixelPopSentEvt);
               buffer[i*8 + k][j*8 + l] = codecPixelPop;
            }
         }
      }
   }

   // CNTRL Quantization
   for(i=0; i<NUM_ROW_BLOCKS; i++)
   {
      for(j=0; j<NUM_COL_BLOCKS; j++)
      {
         // Quantize the block in place
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               buffer[i*8 + k][j*8 + l] >>= QUANT_SHIFT_TABLE[k*8 + l];
            }
         }
      }
   }
  }
};

behavior CntrlSendImage( in event   UartPixelRecvEvt,
                         in short   buffer[ROW_SIZE][COL_SIZE],
                         out event  UartPixelSentEvt,
                         out char   uartPixel )
{
  short temp;
  int i, j;

  void main(void) {
   for(i=0; i<ROW_SIZE; i++)
   {
       for(j=0; j<COL_SIZE; j++)
      {
         temp = buffer[i][j];

         notify(UartPixelSentEvt);
         uartPixel = ((char*)&temp)[0];    /* send upper byte */
         wait(UartPixelRecvEvt);

         notify(UartPixelSentEvt);
         uartPixel = ((char*)&temp)[1];    /* send lower byte */
         wait(UartPixelRecvEvt);
      }
   }
  }
};

behavior Cntrl( in event   DoneCaptureEvt,
                in event   CcdppPixelSentEvt,
                in char    ccdppPixel,
                out event  StartCaptureEvt,
                out event  CcdppPixelReqEvt,
                in event   CodecPixelPushRecvEvt,
                in event   CodecPixelPopSentEvt,
                in event   DoneFdctEvt,
                in short   codecPixelPop,
                out event  CodecPixelPushSentEvt,
                out event  CodecPixelPopReqEvt,
                out event  StartFdctEvt,
                out short  codecPixelPush,
                in event   UartPixelRecvEvt,
                out event  UartPixelSentEvt,
                out char   uartPixel )
```

```
{

   short buffer[ROW_SIZE][COL_SIZE];

   CntrlCaptureImage CntrlCaptureImageInst( DoneCaptureEvt,
                                            CcdppPixelSentEvt,
                                            ccdppPixel,
                                            buffer,
                                            StartCaptureEvt,
                                            CcdppPixelReqEvt );

   CntrlCompressImage CntrlCompressImageInst( CodecPixelPushRecvEvt,
                                              CodecPixelPopSentEvt,
                                              DoneFdctEvt,
                                              codecPixelPop,
                                              buffer,
                                              CodecPixelPushSentEvt,
                                              CodecPixelPopReqEvt,
                                              StartFdctEvt,
                                              codecPixelPush );

   CntrlSendImage     CntrlSendImageInst( UartPixelRecvEvt,
                                      buffer,
                                      UartPixelSentEvt,
                                      uartPixel);

   void main(void) {
      CntrlCaptureImageInst.main();
      CntrlCompressImageInst.main();
      CntrlSendImageInst.main();
   }
};


////////////////////////////////////////
// Testbench                          //
// for Functional model               //
////////////////////////////////////////
behavior Main
{
   event StartCcdCaptureEvt,
         DoneCcdCaptureEvt,
         CcdPixelReqEvt,
         CcdPixelSentEvt,
         StartCcdppCaptureEvt,
         DoneCcdppCaptureEvt,
         CcdppPixelReqEvt,
         CcdppPixelSentEvt,
         CodecPixelPushRecvEvt,
         CodecPixelPushSentEvt,
         CodecPixelPopReqEvt,
         CodecPixelPopSentEvt,
         StartFdctEvt,
         DoneFdctEvt,
         UartPixelSentEvt,
         UartPixelRecvEvt;

   char  ccdPixelOut,
         ccdppPixelOut,
         uartData;

   short codecPixelPop,
         codecPixelPush;

   Ccd   CcdInst( StartCcdCaptureEvt,
                  CcdPixelReqEvt,
                  DoneCcdCaptureEvt,
                  CcdPixelSentEvt,
                  ccdPixelOut );
   Ccdpp CcdppInst( StartCcdppCaptureEvt,
                    CcdppPixelReqEvt,
                    DoneCcdppCaptureEvt,
                    CcdppPixelSentEvt,
                    ccdppPixelOut,
                    DoneCcdCaptureEvt,
                    CcdPixelSentEvt,
                    ccdPixelOut,
                    StartCcdCaptureEvt,
                    CcdPixelReqEvt );

   Codec CodecInst( codecPixelPush,
                    CodecPixelPushSentEvt,
                    StartFdctEvt,
                    CodecPixelPopReqEvt,
                    codecPixelPop,
                    CodecPixelPushRecvEvt,
                    DoneFdctEvt,
                    CodecPixelPopSentEvt );

   Uart  UartInst( uartData,
                   UartPixelSentEvt,
                   UartPixelRecvEvt );

   Cntrl CntrlInst( DoneCcdppCaptureEvt,
                    CcdppPixelSentEvt,
                    ccdppPixelOut,
                    StartCcdppCaptureEvt,
                    CcdppPixelReqEvt,
                    CodecPixelPushRecvEvt,
                    CodecPixelPopSentEvt,
                    DoneFdctEvt,
                    codecPixelPop,
                    CodecPixelPushSentEvt,
                    CodecPixelPopReqEvt,
                    StartFdctEvt,
                    codecPixelPush,
                    UartPixelRecvEvt,
                    UartPixelSentEvt,
                    uartData );

   int main(void) {
      par {
         CcdInst.main();
         CcdppInst.main();
         CodecInst.main();
         UartInst.main();
         CntrlInst.main();
      }

      return 0;
   }
};
```

SpecC Transaction-level Model

```
////////////////////////////////////////////////////////////////////
// File: digcam.sc                                               //
// Desc: SpecC Transaction-level Model of the Digital Camera      //
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include "image.h"

#define CLK_CYCLE 1
```

```
////////////////////////////////////////
// Interface Definitions              //
// for Transaction-level model        //
////////////////////////////////////////
interface CcdToCcdBusIf
{
   void   ready();
   void   write(char);
};

interface CcdppToCcdBusIf
{
   void start();
   char read();
};

interface CntrlToMainBusIf
{
   short  read(short);
   void   write(short, short);
   void   start_ccdpp();
   void   start_uart();
};

interface CcdppToMainBusIf
{
   void   write(short, short);
   void   ccdpp_ready();
   void   ccdpp_done();
};

interface UartToMainBusIf
{
   short  read(short);
   void   uart_ready();
   void   uart_done();
};


////////////////////////////////////////
// CCD BUS Channel                    //
// for Transaction-level model        //
////////////////////////////////////////
channel CcdBus() implements CcdToCcdBusIf,
                            CcdppToCcdBusIf
{
   char  pixel;
   bool  valid = false;
   bool  busy = true;
   event StartEvt, ValidEvt;

   // CcdToCcdBusIf Interface Functions
   void ready()
   {
      busy = false;
      wait(StartEvt);
      busy = true;
      return;
   }

   void write(char data)
   {
      do {waitfor(1);}
      while(valid == true);
      pixel = data;
      valid = true;
      notify(ValidEvt);
   }
   // CcdppToCcdBusIf Interface Functions
   void start()
   {
      do {waitfor(1);}
      while(busy == true);
      notify(StartEvt);
      return;
   }

   char read()
   {
      if(!valid)
         wait(ValidEvt);
      valid = false;
      return pixel;
   }
};


////////////////////////////////////////
// Main Bus and Shared Memory Module  //
// for Transaction-level model        //
////////////////////////////////////////
channel MainBus() implements CntrlToMainBusIf,
                             CcdppToMainBusIf,
                             UartToMainBusIf
{
   bool  ccdppBusy, uartBusy;
   short memory[ROW_SIZE*COL_SIZE];
   event StartCcdppEvt;
   event CcdppDoneEvt;
   event StartUartEvt;
   event UartDoneEvt;

   // Cntrl/Ccdpp/UartToMainBusIf Interface Functions
   short read(short addr)
   {
      waitfor(2*CLK_CYCLE);
      return memory[addr];
   }

   void write(short addr, short data)
   {
      waitfor(2*CLK_CYCLE);
      memory[addr] = data;
   }

   // CntrlToMainBusIf Interface Functions
   void start_ccdpp()
   {
      if(ccdppBusy)
         wait(CcdppDoneEvt);
      notify(StartCcdppEvt);
      wait(CcdppDoneEvt);
      return;
   }

   void start_uart()
   {
      if(uartBusy)
         wait(UartDoneEvt);
      notify(StartUartEvt);
      wait(UartDoneEvt);
      return;
   }

   // CcdppToMainBusIf Interface Functions
   void ccdpp_ready()
   {
      ccdppBusy = false;
      wait(StartCcdppEvt);
      ccdppBusy = true;
      return;
   }

   void ccdpp_done()
   {
      ccdppBusy = false;
      notify(CcdppDoneEvt);
      return;
   }

   // UartToMainBusIf Interface Functions
   void uart_ready()
   {
      uartBusy = false;
      wait(StartUartEvt);
      uartBusy = true;
      return;
   }

   void uart_done()
   {
      uartBusy = false;
      notify(UartDoneEvt);
      return;
   }
};


////////////////////////////////////////
// CCD Module                         //
// for Transaction-level model        //
////////////////////////////////////////
behavior CcdCapture( CcdToCcdBusIf CcdBus0,
                     out char buffer[ROW_SIZE][COL_SIZE+2] )
{
   int row, col;

   void main(void)
   {
      CcdBus0.ready();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            // IMAGE array is defined in image.h
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
   }
};

behavior CcdPopPixel( CcdToCcdBusIf CcdBus0,
                      in char buffer[ROW_SIZE][COL_SIZE+2] )
{
   int row, col;

   void main(void)
   {
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            CcdBus0.write( buffer[row][col] );
         }
      }
   }
```

```
};

behavior Ccd( CcdToCcdBusIf CcdBus0 )
{
       char buffer[ROW_SIZE][COL_SIZE+2];

   CcdCapture  CcdCaptureInst( CcdBus0,
                               buffer );

   CcdPopPixel CcdPopPixelInst( CcdBus0,
                                buffer );

       void main(void) {
               CcdCaptureInst.main();
               CcdPopPixelInst.main();
       }
};


////////////////////////////////////////
// CCDPP module                       //
// for Transaction-level model        //
////////////////////////////////////////
behavior Ccdpp( CcdppToCcdBusIf    CcdBus0,
                CcdppToMainBusIf   MainBus0 )
{
   int row, col;
   char bias;
   char tempRow[COL_SIZE];

   void main(void)
   {
      MainBus0.ccdpp_ready();
      CcdBus0.start();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            tempRow[col] = CcdBus0.read();
                       }

                       bias = CcdBus0.read();
                       bias = (bias + CcdBus0.read()) / 2;
                       waitfor(12*CLK_CYCLE);
                       for(col=0; col<COL_SIZE; col++) {

                               tempRow[col] -= bias;
                               waitfor(4*CLK_CYCLE);
            MainBus0.write(row*COL_SIZE+col, tempRow[col]);
                       }
               }
               MainBus0.ccdpp_done();
       }
};


////////////////////////////////////////
// UART module                        //
// for Transaction-level model        //
////////////////////////////////////////
behavior UartInitialize( out FILE *outputFileHandle )
{
       void main(void) {
               outputFileHandle = fopen("uart_out.txt", "w");
       }
};

behavior UartSend( in FILE *outputFileHandle,
                   UartToMainBusIf  MainBus0 )
{
   int i;
   short data;

   void main(void) {
      MainBus0.uart_ready();
      for(i=0; i<(16384/2); i++)
      {
         data = MainBus0.read(i);

         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[0]);
         waitfor(2*CLK_CYCLE);
         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[1]);
         waitfor(2*CLK_CYCLE);
      }

      fclose(outputFileHandle);
      MainBus0.uart_done();
   }
};

behavior Uart( UartToMainBusIf   MainBus0 )
{
   FILE *outputFileHandle;

        UartInitialize UartInitializeInst(outputFileHandle);

        UartSend       UartSendInst( outputFileHandle,
                          MainBus0 );

        void main(void) {
               UartInitializeInst.main();
               UartSendInst.main();
        }
};


////////////////////////////////////////
// CNTRL/CODEC behaviors              //
// for Transaction-level model        //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12 };

const unsigned char QUANT_SHIFT_TABLE[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };

behavior Cntrl( CntrlToMainBusIf MainBus0 )
{
   // CNTRL Vars
   short inBuffer[8][8], outBuffer[8][8];
   short temp;
   short addr;
   int i, j, k, l;
```

```
// Local FDCT Functions
unsigned char C(int h) {
   return h ? 64 : 5;
}

int F(int u, int v, short img[8][8]) {

   long s[8];
   long r;
   unsigned char a;

   for(a=0; a<8; a++)
   {
      s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
             ((img[a][1] * COS_TABLE[8+v])  >> 6) +
             ((img[a][2] * COS_TABLE[16+v]) >> 6) +
             ((img[a][3] * COS_TABLE[24+v]) >> 6) +
             ((img[a][4] * COS_TABLE[32+v]) >> 6) +
             ((img[a][5] * COS_TABLE[40+v]) >> 6) +
             ((img[a][6] * COS_TABLE[48+v]) >> 6) +
             ((img[a][7] * COS_TABLE[56+v]) >> 6);
   }

   r = 0;
   for(a=0; a<8; a++)
   {
      r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
   }

   return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
}

void main(void)
{
   // CNTRL Capture
   MainBus0.start_ccdpp();

   // CNTRL Compress
   for(i=0; i<NUM_ROW_BLOCKS; i++)
   {
      for(j=0; j<NUM_COL_BLOCKS; j++)
      {
         // Push the block and perform FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
               inBuffer[k][l] = (MainBus0.read( addr ) << 6);
            }
         }

         // FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] = F(k, l, inBuffer);
               waitfor(72*CLK_CYCLE);
            }
         }

         // Quantize
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] >>= QUANT_SHIFT_TABLE[k * 8 + l];
```

```
                waitfor(4*CLK_CYCLE);
             }
          }

          // Pop the block and store it in buffer
          for(k=0; k<8; k++)
          {
             for(l=0; l<8; l++)
             {
                addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                MainBus0.write( addr, outBuffer[k][l] );

             }
          }
       }
    }

    // CNTRL Send Image
    MainBus0.start_uart();

    return;
  }
};

////////////////////////////////////////
// Testbench                          //
// for Transaction-level model        //
////////////////////////////////////////
behavior Main {

   CcdBus   CcdBus0;
   MainBus  MainBus0;

   Ccd   CcdInst( CcdBus0 );

   Ccdpp CcdppInst( CcdBus0,
                    MainBus0 );

   Uart  UartInst( MainBus0 );

   Cntrl CntrlInst( MainBus0 );

   int main(void) {
      par {
                  CcdInst.main();
                  CcdppInst.main();
                  UartInst.main();
                  CntrlInst.main();
             }

             return 0;
       }
};
```

SpecC Communication Model

```
////////////////////////////////////////////////////////////////////
// File: digcam.sc                                                //
// Desc: SpecC Communication Model of the Digital Camera          //
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include "image.h"

#define CLK_CYCLE 1

// States for Memory behavior
#define IDLE   0
```

```
#define READ   1
#define WRITE  2
#define DONE   3

////////////////////////////////////////
// Mem Module                         //
// for Communication model            //
////////////////////////////////////////
behavior Mem( in  signal bit[1]       ClockI,
              in  signal bit[1]       ReqI,
              in  signal bit[1]       RwI,
              in  signal bit[15:0]    AddrI,
              inout  signal bit[15:0] DataIO,
              out signal bit[1]       ValidO )
{
   short memory[ROW_SIZE*COL_SIZE];

   char  nextState = IDLE;

   void main(void)
   {
      ValidO = 0;
      while(1) {
         wait ClockI rising;

         printf("%d %d\n", nextState, ReqI); fflush(stdout);

         switch(nextState)
         {
         case IDLE:
            ValidO = 0;
            if((ReqI == 1) && (RwI == 1))
            {
               nextState = READ;
            }
            else if((ReqI == 1) && (RwI == 1))
            {
               nextState = WRITE;
            }
            break;
         case READ:
            DataIO = memory[AddrI];
            ValidO = 1;
            nextState = DONE;
            break;
         case WRITE:
            memory[AddrI] = DataIO;
            ValidO = 1;
            nextState = DONE;
            break;
         case DONE:
            if(ReqI == 0)
            {
               ValidO = 0;
               nextState = IDLE;
            }
            break;
         }
      }
   }
};


////////////////////////////////////////
// CCD Module                         //
// for Communication model            //
////////////////////////////////////////
behavior CcdCapture( in  signal bit[1]  ClockI,
                     in  signal bit[1]  StartI,
                     out char buffer[ROW_SIZE][COL_SIZE+2] )
{
   void ready()
   {
      do {
         wait ClockI rising;
      } while (StartI == 0);
      return;
   }

   int row, col;

   void main(void)
   {
      ready();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            // IMAGE array is defined in image.h
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
   }
};

behavior CcdPopPixel( in  signal bit[1]  ClockI,
                      in  signal bit[1]  ReadyI,
                      out signal bit[7:0]  DataO,
                      out signal bit[1]  ValidO,
                      in char buffer[ROW_SIZE][COL_SIZE+2] )
{
   void write(char data)
   {
      DataO = data;
      ValidO = 1;
      do {
         wait ClockI rising;
      } while (ReadyI == 0);

      ValidO = 0;
   }

   int row, col;

   void main(void)
   {
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            write( buffer[row][col] );
         }
      }
   }
};

behavior Ccd( in  signal bit[1]  ClockI,
              in  signal bit[1]  StartI,
              in  signal bit[1]  ReadyI,
              out signal bit[7:0]  DataO,
              out signal bit[1]  ValidO )
{
       char buffer[ROW_SIZE][COL_SIZE+2];



   CcdCapture  CcdCaptureInst( ClockI,
                               StartI,
                               buffer );

   CcdPopPixel CcdPopPixelInst( ClockI,
                                ReadyI,
                                DataO,
                                ValidO,
                                buffer );

        void main(void) {
                CcdCaptureInst.main();
                CcdPopPixelInst.main();
        }
};


////////////////////////////////////////
// CCDPP module                        //
// for Communication model             //
////////////////////////////////////////
behavior Ccdpp( in  signal bit[1]   ClockI,
                in  signal bit[1]   CcdBusValidI,
                in  signal bit[7:0] CcdBusDataI,
                out signal bit[1]   CcdBusStartO,
                out signal bit[1]   CcdBusReadyO,

                out signal bit[1]   MainBusReqO,
                out signal bit[1]   MainBusRwO,
                out signal bit[15:0]   MainBusAddrO,
                inout signal bit[15:0] MainBusDataIO,
                in  signal bit[1]   MainBusValidI,
                in  signal bit[1]   MainBusStartCcdppI,
                out signal bit[1]   MainBusCcdppBusyO )
{
   int row, col;
   char bias;
   char tempRow[COL_SIZE];

   // CCD Bus Interface Functions
   void ccd_start( void )
   {
      CcdBusStartO = 1;
      wait ClockI rising;
   }

   char ccdbus_read( void )
   {
      char temp;

      CcdBusReadyO = 1;
      do {
         wait ClockI rising;
      }while( CcdBusValidI == 0 );

      CcdBusReadyO = 0;
      temp = CcdBusDataI;
      wait ClockI rising;

      return temp;
   }

   // MAIN Bus Interface Functions
   void mainbus_write(short addr, short data)
   {
      MainBusReqO = 1;
      MainBusRwO = 0;
      MainBusAddrO = addr ;
      MainBusDataIO = data ;
      do {
         wait ClockI rising;
      }while( MainBusValidI == 0 );

      MainBusReqO = 0;
      wait ClockI rising;
      return;
   }

   void ccdpp_ready()
   {
      MainBusCcdppBusyO = 0;
      do {
         wait ClockI rising;
      }while( MainBusStartCcdppI == 0 );

      MainBusCcdppBusyO = 1;
      return;
   }

   void ccdpp_done()
   {
      MainBusCcdppBusyO = 0;
      wait ClockI rising;
      return;
   }

   void main(void)
   {
      ccdpp_ready();
      ccd_start();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            tempRow[col] = ccdbus_read();
                      }

                      bias = ccdbus_read();
                      bias = (bias + ccdbus_read()) / 2;
                      waitfor(12*CLK_CYCLE);
                      for(col=0; col<COL_SIZE; col++) {

                              tempRow[col] -= bias;
                              waitfor(4*CLK_CYCLE);
            mainbus_write(row*COL_SIZE+col, tempRow[col]);
                      }
               }
               ccdpp_done();
       }
};


////////////////////////////////////////
// UART module                        //
// for Communication model            //
////////////////////////////////////////
behavior UartInitialize( out FILE *outputFileHandle )
{
       void main(void) {
               outputFileHandle = fopen("uart_out.txt", "w");
       }
};

behavior UartSend( in FILE *outputFileHandle,
                   in  signal bit[1]    ClockI,
                   out signal bit[1]    ReqO,
                   out signal bit[1]    RwO,
                   out signal bit[15:0] AddrO,
                   inout signal bit[15:0]  DataIO,
                   in  signal bit[1]    ValidI,
                   in  signal bit[1]    StartUartI,
                   out signal bit[1]    UartBusyO )
{
   int i;
   short data;

   // MAIN Bus Interface Functions
   short read(short addr)
   {
      short  temp;

      ReqO = 1;
      RwO = 1;
      AddrO = addr;
      do {
         wait ClockI rising;
      }while( ValidI == 0 );

      temp = DataIO;
      ReqO = 0;
      wait ClockI rising;

      return temp;
   }

   void uart_ready()
   {
      UartBusyO = 0;
      do {
         wait ClockI rising;
      }while( StartUartI == 0 );

      UartBusyO = 1;
      return;
   }

   void uart_done()
   {
      UartBusyO = 0;
      wait ClockI rising;
      return;
   }

   void main(void) {
      uart_ready();
      for(i=0; i<(16384/2); i++)
      {
         data = read(i);

         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[0]);
         waitfor(2*CLK_CYCLE);
         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[1]);
         waitfor(2*CLK_CYCLE);
      }

      fclose(outputFileHandle);
      uart_done();
   }
};

behavior Uart( in  signal bit[1]    ClockI,
               out signal bit[1]    ReqO,
               out signal bit[1]    RwO,
               out signal bit[15:0] AddrO,
               inout signal bit[15:0]  DataIO,
               in  signal bit[1]    ValidI,
               in  signal bit[1]    StartUartI,
               out signal bit[1]    UartBusyO )
```

```
{
   FILE *outputFileHandle;

        UartInitialize UartInitializeInst(outputFileHandle);

        UartSend       UartSendInst( outputFileHandle,
                                ClockI,
                           ReqO,
                           RwO,
                           AddrO,
                           DataIO,
                           ValidI,
                           StartUartI,
                           UartBusyO );

        void main(void) {
               UartInitializeInst.main();
               UartSendInst.main();
        }
};


////////////////////////////////////////
// CNTRL/CODEC behaviors              //
// for Communication model            //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12 };

const unsigned char QUANT_SHIFT_TABLE[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };

behavior Cntrl( in  signal bit[1]    ClockI,
                out signal bit[1]    ReqO,
                out signal bit[1]    RwO,
                out signal bit[15:0] AddrO,
                inout signal bit[15:0]  DataIO,
                in  signal bit[1]    ValidI,
                out signal bit[1]    StartCcdppO,
                out signal bit[1]    StartUartO,
                in  signal bit[1]    CcdppBusyI,
                in  signal bit[1]    UartBusyI )
{
   // CNTRL Vars
   short inBuffer[8][8], outBuffer[8][8];
   short temp;
   short addr;
   int i, j, k, l;

   // Local FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {
```

```
   long s[8];
   long r;
   unsigned char a;

   for(a=0; a<8; a++)
   {
      s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
             ((img[a][1] * COS_TABLE[8+v])  >> 6) +
             ((img[a][2] * COS_TABLE[16+v]) >> 6) +
             ((img[a][3] * COS_TABLE[24+v]) >> 6) +
             ((img[a][4] * COS_TABLE[32+v]) >> 6) +
             ((img[a][5] * COS_TABLE[40+v]) >> 6) +
             ((img[a][6] * COS_TABLE[48+v]) >> 6) +
             ((img[a][7] * COS_TABLE[56+v]) >> 6);
   }

   r = 0;
   for(a=0; a<8; a++)
   {
      r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
   }

   return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
}

// MAIN Bus Interface Functions
short read(short address)
{
   short temp;

   ReqO = 1;
   RwO = 1;
   AddrO = address;
   do {
      wait ClockI rising;
   }while( ValidI == 0 );

   temp = DataIO;
   ReqO = 0;
   wait ClockI rising;

   return temp;
}

void write(short address, short data)
{
   ReqO = 1;
   RwO = 0;
   AddrO = address;
   DataIO = data;
   do {
      wait ClockI rising;
   }while( ValidI == 0 );

   ReqO = 0;
   wait ClockI rising;
   return;
}

void start_ccdpp()
{
   do {
      wait ClockI rising;
   }while( CcdppBusyI == 1 );

   StartCcdppO = 1;

   wait ClockI rising;
   StartCcdppO = 0;

   do {
      wait ClockI rising;
   }while( CcdppBusyI == 1 );

   return;
}

void start_uart()
{
   do {
      wait ClockI rising;
   }while( UartBusyI == 1 );

   StartUartO = 1;

   wait ClockI rising;
   StartUartO = 0;

   do {
      wait ClockI rising;
   }while( UartBusyI == 1 );

   return;
}

void main(void)
{
   // CNTRL Capture
   start_ccdpp();

   // CNTRL Compress
   for(i=0; i<NUM_ROW_BLOCKS; i++) {

      for(j=0; j<NUM_COL_BLOCKS; j++) {

         // Push the block and perform FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
               inBuffer[k][l] = (read( addr ) << 6);
            }
         }

         // FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] = F(k, l, inBuffer);
               waitfor(72*CLK_CYCLE);
            }
         }

         // Quantize
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] >>= QUANT_SHIFT_TABLE[k * 8 + l];
               waitfor(4*CLK_CYCLE);
            }
         }

         // Pop the block and store it in buffer
         for(k=0; k<8; k++)
```

```
            {
               for(l=0; l<8; l++)
               {
                  addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                  write( addr, outBuffer[k][l] );

               }
            }
         }
      }

      // CNTRL Send Image
      start_uart();

      return;
   }
};

behavior ClockGen ( out signal bit[1]   ClockO )
{
   void main(void)
   {
      while(1)
      {
         ClockO = 0;
         waitfor(1);
         ClockO = 1;
         waitfor(1);
      }
   }
};

////////////////////////////////////////
// Testbench                          //
// for Communication model            //
////////////////////////////////////////
behavior Main {

   signal bit[1]     Clock = 0;

   signal bit[1]     CcdStart;
   signal bit[1]     CcdReady;
   signal bit[1]     CcdValid;
   signal bit[7:0]   CcdData;

   signal bit[1]     MainReq;
   signal bit[1]     MainRw;
   signal bit[15:0]  MainAddr;
   signal bit[15:0]  MainData;
   signal bit[1]     MainValid;
   signal bit[1]     StartCcdpp;
   signal bit[1]     CcdppBusy;
   signal bit[1]     StartUart;
   signal bit[1]     UartBusy;

   ClockGen ClockGenInst( Clock );

   Mem   MemInst( Clock,
                  MainReq,
                  MainRw,
                  MainAddr,
                  MainData,
                  MainValid );

   Ccd   CcdInst( Clock,
                  CcdStart,
                  CcdReady,
                  CcdData,
                  CcdValid );
   Ccdpp CcdppInst( Clock,
                    CcdValid,
                    CcdData,
                    CcdStart,
                    CcdReady,
                    MainReq,
                    MainRw,
                    MainAddr,
                    MainData,
                    MainValid,
                    StartCcdpp,
                    CcdppBusy );

   Uart  UartInst( Clock,
                   MainReq,
                   MainRw,
                   MainAddr,
                   MainData,
                   MainValid,
                   StartUart,
                   UartBusy );

   Cntrl CntrlInst( Clock,
                    MainReq,
                    MainRw,
                    MainAddr,
                    MainData,
                    MainValid,
                    StartCcdpp,
                    StartUart,
                    CcdppBusy,
                    UartBusy );

   int main(void) {
      par {
         ClockGenInst.main();
         MemInst.main();
         CcdInst.main();
         CcdppInst.main();
         UartInst.main();
         CntrlInst.main();
      }

      return 0;
   }
};
```

APPENDIX B DIGITAL CAMERA SYSTEM: SYSTEMC MODELS

This appendix contains the SpecC source code for the functional, transaction-level, and communication models of the digital camera system. Each model was developed and compiled using Microsoft Visual C++ 6.0 with the SystemC 2.0.1 library. All model simulations were done using the reference simulator provided by OSCI.

SystemC Functional Model

```
////////////////////////////////////////////////////////////////////
// File: digcam.cpp                                               //
// Desc: SystemC Functional Model of the Digital Camera           //
////////////////////////////////////////////////////////////////////
#include "systemc.h"
#include "image.h"

////////////////////////////////////////
// CCD Module                         //
// for Functional model               //
////////////////////////////////////////
SC_MODULE( Ccd )
{
   // CCD Ports
   sc_fifo_in<bool> StartCcdCapture;
   sc_fifo_out<bool> DoneCcdCapture;
   sc_fifo_out<char> Pixel;

   // CCD Vars
   char buffer[ROW_SIZE][COL_SIZE+2];
   int  row, col;

   // CCD Events
   sc_event   StartPopEvt;

   // CCD Processes
   void capture(void)
   {
      StartCcdCapture.read();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
      DoneCcdCapture.write( true );
      notify(StartPopEvt);
   }

   void pop(void)
   {
      wait(StartPopEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            Pixel.write( buffer[row][col] );
         }
      }
   }

   // Module Constructor
   SC_CTOR( Ccd ) {
      SC_THREAD( capture );
      SC_THREAD( pop );
   }
};


////////////////////////////////////////
// CCDPP module                       //
// for Functional model               //
////////////////////////////////////////
SC_MODULE( Ccdpp )
{
   // CCDPP Ports
   sc_fifo_in<bool>  StartCcdppCapture, DoneCcdCapture;
   sc_fifo_in<char>  CcdPixel;
   sc_fifo_out<bool> DoneCcdppCapture, StartCcdCapture;
   sc_fifo_out<char> CcdppPixel;

   // CCDPP Vars
   char buffer[ROW_SIZE][COL_SIZE];
   int  row, col;
   char bias;

   // CCDPP Events
   sc_event   StartPopEvt;

   // CCDPP Processes
   void capture(void)
   {
      StartCcdppCapture.read();
      StartCcdCapture.write( true );
      DoneCcdCapture.read();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[row][col] = CcdPixel.read();
         }

         bias = CcdPixel.read();
         bias = ( bias + CcdPixel.read() ) / 2;
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[row][col] -= bias;
         }
      }

      DoneCcdppCapture.write( true );
      notify(StartPopEvt);
   }

   void pop(void)
   {
      wait(StartPopEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            CcdppPixel.write( buffer[row][col] );
         }
      }
   }
   // Module Constructor
   SC_CTOR( Ccdpp ) {
      SC_THREAD( capture );
      SC_THREAD( pop );
   }
};


////////////////////////////////////////
// CODEC module                       //
// for Functional model               //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12
};

SC_MODULE( Codec )
{
   // CODEC Ports
   sc_fifo_in<short>    PixelIn;
   sc_fifo_out<short>   PixelOut;

   // CODEC Vars
   int   idx;
   int   i;
   short inBuffer[8][8], outBuffer[8][8];

   // FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {

      long  s[8];
      long  r;
      unsigned char  a;

      for(a=0; a<8; a++)
      {
         s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
                ((img[a][1] * COS_TABLE[8+v])  >> 6) +
                ((img[a][2] * COS_TABLE[16+v]) >> 6) +
                ((img[a][3] * COS_TABLE[24+v]) >> 6) +
                ((img[a][4] * COS_TABLE[32+v]) >> 6) +
                ((img[a][5] * COS_TABLE[40+v]) >> 6) +
                ((img[a][6] * COS_TABLE[48+v]) >> 6) +
                ((img[a][7] * COS_TABLE[56+v]) >> 6);
      }

      r = 0;
      for(a=0; a<8; a++)
      {
         r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
      }

      return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
   }

   // CODEC Processes
   void main(void) {
      int x, y;

      i = 0;
      while(i < 128) {
         // CODEC Push Pixel
         for(idx=0; idx<64; idx++)
         {
            inBuffer[idx / 8][idx % 8] = (PixelIn.read() << 6);
         }

         // CODEC Do FDCT
         for(x=0; x<8; x++) {

            for(y=0; y<8; y++) {
               outBuffer[x][y] = F(x, y, inBuffer);
            }
         }

         // CODEC Pop Pixel
         for(idx=0; idx<64; idx++)
         {
            PixelOut.write( outBuffer[idx / 8][idx % 8] );
         }
         ++i;
      }
   }

   // Module Constructor
   SC_CTOR( Codec ) {
      SC_THREAD( main );
   }
};


////////////////////////////////////////
// UART module                        //
// for Functional model               //
////////////////////////////////////////
SC_MODULE( Uart )
{
   // UART Ports
   sc_fifo_in<char>  DataIn;

   // UART Vars
   FILE  *outputFileHandle;
   char   data;

   // UART Processes
   void uartSend(void)
   {
      while(1) {
         data = DataIn.read();
         fprintf(outputFileHandle, "%i\n", (int)data);
      }
   }

   // Module Constructor
   SC_CTOR( Uart )
   {
      outputFileHandle = fopen("uart_out.txt", "w");
      SC_THREAD( uartSend );
   }

   // Module Destructor
   ~Uart(void)
   {
      fclose(outputFileHandle);
   }
};
```

```
////////////////////////////////////////
// CNTRL module                       //
// for Functional model               //
////////////////////////////////////////
const unsigned char QUANT_SHIFT_TABLE[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };
SC_MODULE(Cntrl) {
   // CNTRL Ports
   sc_fifo_in<bool>  DoneCapture;
   sc_fifo_in<char>  CcdppPixelPop;
   sc_fifo_in<short> CodecPixelPop;

   sc_fifo_out<bool> StartCapture;
   sc_fifo_out<char> UartPixel;
   sc_fifo_out<short>   CodecPixelPush;

   // CNTRL Vars
   short buffer[ROW_SIZE][COL_SIZE];
   short temp;
   int i, j, k, l;

   void main( void )
   {
      // CNTRL Capture
      StartCapture.write( true );
      DoneCapture.read();
      for(i=0; i<ROW_SIZE; i++)
      {
         for(j=0; j<COL_SIZE; j++)
         {
            buffer[i][j] = CcdppPixelPop.read();
         }
      }

      // CNTRL Compress
      for(i=0; i<NUM_ROW_BLOCKS; i++) {

         for(j=0; j<NUM_COL_BLOCKS; j++) {

            // Push the block and perform FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  CodecPixelPush.write( (char)buffer[i*8 + k][j*8 + l] );
               }
            }

            // Pop the block and store it in buffer
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  buffer[i*8 + k][j*8 + l] = CodecPixelPop.read();
               }
            }
         }
      }

      // CNTRL Quantization
      for(i=0; i<NUM_ROW_BLOCKS; i++)
      {
         for(j=0; j<NUM_COL_BLOCKS; j++)
         {
            // Quantize the block in place
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  buffer[i*8 + k][j*8 + l] >>= QUANT_SHIFT_TABLE[k*8 + l];
               }
            }
         }
      }

      // CNTRL Send Image
      for(i=0; i<ROW_SIZE; i++)
      {
         for(j=0; j<COL_SIZE; j++)
         {
            temp = buffer[i][j];

            UartPixel.write( ((char*)&temp)[0] );   // Send Upper Byte
            UartPixel.write( ((char*)&temp)[1] );   // Send Lower Byte
         }
      }
      return;
   }


   // Module Constructor
   SC_CTOR( Cntrl ) {
      SC_THREAD( main );
   }
};


////////////////////////////////////////
// Testbench                          //
// for Functional model               //
////////////////////////////////////////
int sc_main(int, char**)
{
   // Module Instances
   Ccd   CcdInst("Ccd");
   Ccdpp CcdppInst("Ccdpp");
   Codec CodecInst("Codec");
   Uart  UartInst("Uart");
   Cntrl CntrlInst("Cntrl");

   // Channel Instances
   sc_fifo<bool>      StartCcdCapture("StartCcdCapture", 1);
   sc_fifo<bool>      DoneCcdCapture("DoneCcdCapture", 1);
   sc_fifo<char>  CcdPixel("CcdPixel", 1);
   sc_fifo<bool>  StartCcdppCapture("StartCcdppCapture", 1);
   sc_fifo<bool>      DoneCcdppCapture("DoneCcdppCapture", 1);
   sc_fifo<char>      CcdppPixel("CcdppPixel", 1);
   sc_fifo<short>     CodecPixelPush("CodecPixelPush", 1);
   sc_fifo<short> CodecPixelPop("CodecPixelPop", 1);
   sc_fifo<char>  UartPixel("UartPixel", 1);


   // CCD Port Bindings
   CcdInst.StartCcdCapture( StartCcdCapture );
   CcdInst.DoneCcdCapture( DoneCcdCapture );
   CcdInst.Pixel( CcdPixel );

   // CCDPP Port Bindings
   CcdppInst.StartCcdCapture( StartCcdCapture );
   CcdppInst.DoneCcdCapture( DoneCcdCapture );
   CcdppInst.CcdPixel( CcdPixel );
   CcdppInst.StartCcdppCapture( StartCcdppCapture );
   CcdppInst.DoneCcdppCapture( DoneCcdppCapture );
   CcdppInst.CcdppPixel( CcdppPixel );

   // CODEC Port Bindings
   CodecInst.PixelIn( CodecPixelPush );
   CodecInst.PixelOut( CodecPixelPop );

   // UART Port Bindings
   UartInst.DataIn( UartPixel );

   // CNTRL Port Bindings
   CntrlInst.StartCapture( StartCcdppCapture );
   CntrlInst.DoneCapture( DoneCcdppCapture );
   CntrlInst.CcdppPixelPop( CcdppPixel );
   CntrlInst.CodecPixelPush( CodecPixelPush );
   CntrlInst.CodecPixelPop( CodecPixelPop );
   CntrlInst.UartPixel( UartPixel );

   // Begin UNTIMED Simulation
   sc_start(-1);

   return 0;
}
```

SystemC Transaction-level Model

```
////////////////////////////////////////////////////////////////////
// File: digcam.cpp                                               //
// Desc: SystemC Transaction-level Model of the Digital Camera    //
////////////////////////////////////////////////////////////////////

#include "systemc.h"
#include "image.h"

#define CLK_CYCLE 1

////////////////////////////////////////
// Interface Definitions              //
// for Transaction-level model        //
////////////////////////////////////////
class CcdToCcdBusIf: virtual public sc_interface
{
public:
   virtual void   ready() = 0;
   virtual void   write(char) = 0;
};

class CcdppToCcdBusIf: virtual public sc_interface
{
public:
   virtual void   start() = 0;
   virtual char   read() = 0;
};
class CntrlToMainBusIf: virtual public sc_interface
{
public:
   virtual short  read(short) = 0;
   virtual void   write(short, short) = 0;
   virtual void   start_ccdpp() = 0;
   virtual void   start_uart() = 0;
};

class CcdppToMainBusIf: virtual public sc_interface
{
public:
   virtual void   write(short, short) = 0;
   virtual void   ccdpp_ready() = 0;
   virtual void   ccdpp_done() = 0;
};

class UartToMainBusIf: virtual public sc_interface
{
public:
   virtual short  read(short) = 0;
   virtual void   uart_ready() = 0;
   virtual void   uart_done() = 0;
};


////////////////////////////////////////
// CCD BUS Channel                    //
// for Transaction-level model        //
////////////////////////////////////////
class CcdBus:  public sc_module,
               public CcdToCcdBusIf,
               public CcdppToCcdBusIf
{
private:
   char pixel;
   bool valid, busy;
   sc_event StartEvt, ValidEvt;

public:
   // CCD BUS Constructor
   CcdBus(sc_module_name nm) : sc_module(nm), valid(FALSE), busy(TRUE)
   {
   }

   // CcdToCcdBusIf Interface Functions
   void CcdBus::ready()
   {
      busy = FALSE;
      wait(StartEvt);
      busy = TRUE;
      return;
   }

   void CcdBus::write(char data)
   {
      do {
         wait( 1*CLK_CYCLE, SC_NS );
      }while(valid == TRUE);
      pixel = data;
      valid = TRUE;
      notify(ValidEvt);
   }

   // CcdppToCcdBusIf Interface Functions
   void CcdBus::start()
   {
      do{
         wait( 1*CLK_CYCLE, SC_NS );
      }while(busy == TRUE);
      notify(StartEvt);
      return;
   }

   char CcdBus::read()
   {
      if(!valid)
         wait(ValidEvt);
      valid = FALSE;
      return pixel;
   }
```

```
};


////////////////////////////////////////
// Main Bus and Shared Memory Module  //
// for Transaction-level model        //
////////////////////////////////////////
class MainBus:  public sc_module,
                public CntrlToMainBusIf,
                public CcdppToMainBusIf,
                public UartToMainBusIf
{
private:
   bool  ccdppBusy, uartBusy;
   short memory[ROW_SIZE*COL_SIZE];
   sc_event StartCcdppEvt;
   sc_event CcdppDoneEvt;
   sc_event StartUartEvt;
   sc_event UartDoneEvt;
public:
   // Channel Constructor
   MainBus(sc_module_name nm) : sc_module(nm),
                                ccdppBusy(FALSE),
                                uartBusy(FALSE)
   {}

   // Cntrl/Ccdpp/UartToMainBusIf Interface Functions
   short MainBus::read(short addr)
   {
      wait(2*CLK_CYCLE, SC_NS);
      return memory[addr];
   }

   void MainBus::write(short addr, short data)
   {
      wait(2*CLK_CYCLE, SC_NS);
      memory[addr] = data;
   }

   // CntrlToMainBusIf Interface Functions
   void MainBus::start_ccdpp()
   {
      if(ccdppBusy)
         wait(CcdppDoneEvt);
      notify(StartCcdppEvt);
      wait(CcdppDoneEvt);
      return;
   }

   void MainBus::start_uart()
   {
      if(uartBusy)
         wait(UartDoneEvt);
      notify(StartUartEvt);
      wait(UartDoneEvt);
      return;
   }

   // CcdppToMainBusIf Interface Functions
   void MainBus::ccdpp_ready()
   {
      ccdppBusy = FALSE;
      wait(StartCcdppEvt);
      ccdppBusy = TRUE;
      return;
   }

   void MainBus::ccdpp_done()
   {
      ccdppBusy = FALSE;
      notify(CcdppDoneEvt);
      return;
   }

   // UartToMainBusIf Interface Functions
   void MainBus::uart_ready()
   {
      uartBusy = FALSE;
      wait(StartUartEvt);
      uartBusy = TRUE;
      return;
   }

   void MainBus::uart_done()
   {
      uartBusy = FALSE;
      notify(UartDoneEvt);
      return;
   }
};


////////////////////////////////////////
// CCD Module                         //
// for Transaction-level model        //
////////////////////////////////////////
SC_MODULE( Ccd )
{
   // CCD Ports
   sc_port<CcdToCcdBusIf>   CcdBus;
   double   simTime;

   // CCD Vars
   char buffer[ROW_SIZE][COL_SIZE+2];
   int  row, col;

   // CCD Events
   sc_event    StartPopEvt;

   // CCD Processes
   void capture(void)
   {
      CcdBus->ready();
      simTime = sc_simulation_time();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            // IMAGE array is defined in image.h
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
      notify(StartPopEvt);
   }

   void pop(void)
   {
      wait(StartPopEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            CcdBus->write( buffer[row][col] );
         }
      }
      cout << "CCD\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccd ) {
      SC_THREAD( capture );
      SC_THREAD( pop );
   }
};


////////////////////////////////////////
// CCDPP module                       //
// for Transaction-level model        //
////////////////////////////////////////
SC_MODULE( Ccdpp )
{
   // CCDPP Ports
   sc_port<CcdppToCcdBusIf>   CcdBus;
   sc_port<CcdppToMainBusIf>  MainBus;

   // CCDPP Vars
   char buffer[COL_SIZE];
   int  row, col;
   char bias;

   double   simTime;

   // CCDPP Events
   sc_event    StartPopEvt;

   // CCDPP Processes
   void capture(void)
   {
      MainBus->ccdpp_ready();

      simTime = sc_simulation_time();

      CcdBus->start();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] = CcdBus->read();
         }

         // Perform Zero Bias Adjustment
         bias = CcdBus->read();
         bias = ( bias + CcdBus->read() ) / 2;
         wait(12*CLK_CYCLE, SC_NS);
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] -= bias;
            wait(4*CLK_CYCLE, SC_NS);
            MainBus->write(row*COL_SIZE+col, buffer[col]);
         }
      }

      MainBus->ccdpp_done();

      cout << "CCDPP\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccdpp ) {
      SC_THREAD( capture );
   }
};


////////////////////////////////////////
// UART module                        //
// for Transaction-level model        //
////////////////////////////////////////
SC_MODULE( Uart )
{
   // UART Ports
   sc_port<UartToMainBusIf> MainBus;

   // UART Vars
   FILE *outputFileHandle;
   short       data;
   short i;

   double simTime;

   // UART Process
   void send(void)
   {
      MainBus->uart_ready();
      simTime = sc_simulation_time();
      for(i=0; i<(16384/2); i++)
      {
         data = MainBus->read(i);

         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[0]);
         wait(2*CLK_CYCLE, SC_NS);
         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[1]);
         wait(2*CLK_CYCLE, SC_NS);
      }
      MainBus->uart_done();
      cout << "UART\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // UART Constructor
   SC_CTOR( Uart )
   {
      outputFileHandle = fopen("uart_out.txt", "w");
      SC_THREAD( send );
   }

   // UART Destructor
   ~Uart(void)
   {
      fclose(outputFileHandle);
   }
};


////////////////////////////////////////
// CNTRL module                       //
// for Transaction-level model        //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12
};
const unsigned char QUANT_SHIFT_TABLE[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };

SC_MODULE(Cntrl) {
   // CNTRL Ports
   sc_port<CntrlToMainBusIf>   MainBus;

   // CNTRL Vars
   short inBuffer[8][8], outBuffer[8][8];
   short temp;
   short addr;
   int i, j, k, l;

   double   simTime;

   // Local FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {

      long s[8];
      long r;
      unsigned char a;

      for(a=0; a<8; a++)
      {
         s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
                ((img[a][1] * COS_TABLE[8+v])  >> 6) +
                ((img[a][2] * COS_TABLE[16+v]) >> 6) +
                ((img[a][3] * COS_TABLE[24+v]) >> 6) +
                ((img[a][4] * COS_TABLE[32+v]) >> 6) +
                ((img[a][5] * COS_TABLE[40+v]) >> 6) +
                ((img[a][6] * COS_TABLE[48+v]) >> 6) +
                ((img[a][7] * COS_TABLE[56+v]) >> 6);
      }

      r = 0;
      for(a=0; a<8; a++)
      {
         r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
      }

      return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
   }

   // CNTRL Process
   void main( void )
   {
      simTime = sc_simulation_time();

      // CNTRL Capture
      MainBus->start_ccdpp();

      // CNTRL Compress
      for(i=0; i<NUM_ROW_BLOCKS; i++) {

         for(j=0; j<NUM_COL_BLOCKS; j++) {
            // Push the block and perform FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                  inBuffer[k][l] = (MainBus->read( addr ) << 6);
               }
            }

            // FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  outBuffer[k][l] = F(k, l, inBuffer);
                  wait(72*CLK_CYCLE, SC_NS);
               }
            }

            // Quantize
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  outBuffer[k][l] >>= QUANT_SHIFT_TABLE[k * 8 + l];
                  wait(4*CLK_CYCLE, SC_NS);
               }
            }

            // Pop the block and store it in buffer
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                  MainBus->write( addr, outBuffer[k][l] );

               }
            }
         }
      }

      // CNTRL Send Image
      MainBus->start_uart();
      cout << "CNTRL\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
      // Stop the simulation manually
      sc_stop();
      return;
   }

   // CNTRL Constructor
   SC_CTOR( Cntrl ) {
      SC_THREAD( main );
   }
};


////////////////////////////////////////
// Testbench                          //
// for Transaction-level model        //
////////////////////////////////////////
int sc_main(int, char**)
{
   // Module Instances
   Ccd   CcdInst("CcdInst");
   Ccdpp CcdppInst("CcdppInst");
   Uart  UartInst("UartInst");
   Cntrl CntrlInst("CntrlInst");

   // Channel Instances
   CcdBus CcdBusInst("CcdBusInst");
   MainBus MainBusInst("MainBusInst");

   // Bind Channels to Modules via Interfaces
   CcdInst.CcdBus( CcdBusInst );
   CcdppInst.CcdBus( CcdBusInst );

   CcdppInst.MainBus( MainBusInst );
   UartInst.MainBus( MainBusInst );
   CntrlInst.MainBus( MainBusInst );

   // Begin TIMED Simulation (Run for 100 milliseconds)
   cout << "Simulation started at "
        << sc_simulation_time()/1000 << " us" << endl;
   sc_start(100, SC_MS);
   cout << "Simulation stopped at "
        << sc_simulation_time()/1000 << " us" << endl;
   return 0;
}
```

SystemC Communication Model after Adapter Synthesis Phase

```
////////////////////////////////////////////////////////////////////
// File: digcam.cpp                                              //
// Desc: SystemC Communication Model of the Digital Camera       //
////////////////////////////////////////////////////////////////////

#include "systemc.h"
#include "image.h"

#define CLK_CYCLE 1

////////////////////////////////////////
// Interface Definitions              //
// for Communication model            //
////////////////////////////////////////
class CcdToCcdBusIf: virtual public sc_interface
{
public:
   virtual void   ready() = 0;
   virtual void   write(char) = 0;
};

class CcdppToCcdBusIf: virtual public sc_interface
{
public:
   virtual void   start() = 0;
   virtual char   read() = 0;
};
class CntrlToMainBusIf: virtual public sc_interface
{
public:
   virtual short  read(short) = 0;
   virtual void   write(short, short) = 0;
   virtual void   start_ccdpp() = 0;
   virtual void   start_uart() = 0;
};

class CcdppToMainBusIf: virtual public sc_interface
{
public:
   virtual void   write(short, short) = 0;
   virtual void   ccdpp_ready() = 0;
   virtual void   ccdpp_done() = 0;
};

class UartToMainBusIf: virtual public sc_interface
{
public:
   virtual short  read(short) = 0;
   virtual void   uart_ready() = 0;
   virtual void   uart_done() = 0;
};


////////////////////////////////////////
// CCD BUS Adapters                   //
// for Communication model            //
////////////////////////////////////////
class CcdToCcdBusAdapter:  public sc_module,
                           public CcdToCcdBusIf
{
public:
   sc_in<bool>    ClockI;
   sc_in<bool>    StartI;
   sc_in<bool>    ReadyI;
   sc_out<char>   DataO;
   sc_out<bool>   ValidO;

   void CcdToCcdBusAdapter::ready( void )
   {
      do {
         wait( ClockI->posedge_event() );
      } while ( StartI.read() != TRUE );
   }

   void CcdToCcdBusAdapter::write(char data)
   {
      DataO.write( data );
      ValidO.write( TRUE );

      do {
         wait( ClockI->posedge_event() );
      }while( ReadyI.read() != TRUE );

      ValidO.write( FALSE );
   }

   SC_CTOR(CcdToCcdBusAdapter) {
      ValidO.initialize( false );
   }
};


class CcdppToCcdBusAdapter:  public sc_module,
                             public CcdppToCcdBusIf
{
private:
   char  temp;

public:
   sc_in<bool>    ClockI;
   sc_in<char>    DataI;
   sc_in<bool>    ValidI;
   sc_out<bool>   StartO;
   sc_out<bool>   ReadyO;

   void CcdppToCcdBusAdapter::start( void )
   {
      StartO.write(TRUE);
      wait( ClockI->posedge_event() );
   }

   char CcdppToCcdBusAdapter::read( void )
   {
      ReadyO.write( TRUE );
      do {
         wait( ClockI->posedge_event() );
      }while( ValidI.read() != TRUE );

      ReadyO.write( FALSE );
      temp = DataI.read();
      wait( ClockI->posedge_event() );

      return temp;
   }

   SC_CTOR(CcdppToCcdBusAdapter) {
      ReadyO.initialize( false );
   }
};


class CcdppToMainBusAdapter:  public sc_module,
                              public CcdppToMainBusIf
{
public:
   sc_in<bool>    ClockI;
   sc_out<bool>   ReqO;
   sc_out<bool>   RwO;
   sc_out<short>  AddrO;
   sc_inout<short>   DataIO;
   sc_in<bool>    ValidI;
   sc_in<bool>    StartCcdppI;
   sc_out<bool>   CcdppBusyO;

   void CcdppToMainBusAdapter::write(short addr, short data)
   {
      ReqO.write( TRUE );
      RwO.write( FALSE );
      AddrO.write( addr );
      DataIO.write( data );
      do {
         wait( ClockI->posedge_event() );
      }while( ValidI.read() == FALSE );

      ReqO.write( FALSE );
      wait( ClockI->posedge_event() );
      return;
   }

   void CcdppToMainBusAdapter::ccdpp_ready()
   {
      CcdppBusyO.write( FALSE );
      do {
         wait( ClockI->posedge_event() );
      }while( StartCcdppI.read() == FALSE );

      CcdppBusyO.write( TRUE );
      return;
   }

   void CcdppToMainBusAdapter::ccdpp_done()
   {
      CcdppBusyO.write( FALSE );
      wait( ClockI->posedge_event() );
      return;
   }

   SC_CTOR(CcdppToMainBusAdapter) {
      CcdppBusyO.initialize( false );
   }
};


class UartToMainBusAdapter:   public sc_module,
                              public UartToMainBusIf
{
private:
   short  temp;

public:
   sc_in<bool>    ClockI;
   sc_out<bool>   ReqO;
   sc_out<bool>   RwO;
   sc_out<short>  AddrO;
   sc_inout<short>   DataIO;
   sc_in<bool>    ValidI;
   sc_in<bool>    StartUartI;
   sc_out<bool>   UartBusyO;

   short UartToMainBusAdapter::read(short addr)
   {
      ReqO.write( TRUE );
      RwO.write( TRUE );
      AddrO.write( addr );
      do {
         wait( ClockI->posedge_event() );
      }while( ValidI.read() == FALSE );

      temp = DataIO.read();
      ReqO.write( FALSE );
      wait( ClockI->posedge_event() );

      return temp;
   }

   void UartToMainBusAdapter::uart_ready()
   {
      UartBusyO.write( FALSE );
      do {
         wait( ClockI->posedge_event() );
      }while( StartUartI.read() == FALSE );

      UartBusyO.write( TRUE );
      return;
   }

   void UartToMainBusAdapter::uart_done()
   {
      UartBusyO.write( FALSE );
      wait( ClockI->posedge_event() );
      return;
   }

   SC_CTOR(UartToMainBusAdapter) {
      UartBusyO.initialize( FALSE );
   }
};


class CntrlToMainBusAdapter:  public sc_module,
                              public CntrlToMainBusIf
{
private:
   short  temp;

public:
   sc_in<bool>    ClockI;
```

```
sc_out<bool>   ReqO;
sc_out<bool>   RwO;
sc_out<short>  AddrO;
sc_inout<short>   DataIO;
sc_in<bool>    ValidI;
sc_out<bool>   StartCcdppO;
sc_out<bool>   StartUartO;
sc_in<bool>    CcdppBusyI;
sc_in<bool>    UartBusyI;

short CntrlToMainBusAdapter::read(short addr)
{
   ReqO.write( TRUE );
   RwO.write( TRUE );
   AddrO.write( addr );
   do {
      wait( ClockI->posedge_event() );
   }while( ValidI.read() == FALSE );

   temp = DataIO.read();
   ReqO.write( FALSE );
   wait( ClockI->posedge_event() );

   return temp;
}

void CntrlToMainBusAdapter::write(short addr, short data)
{
   ReqO.write( TRUE );
   RwO.write( FALSE );
   AddrO.write( addr );
   DataIO.write( data );
   do {
      wait( ClockI->posedge_event() );
   }while( ValidI.read() == FALSE );

   ReqO.write( FALSE );
   wait( ClockI->posedge_event() );
   return;
}

void CntrlToMainBusAdapter::start_ccdpp()
{
   do {
      wait( ClockI->posedge_event() );
   }while( CcdppBusyI.read() == TRUE );

   StartCcdppO.write( TRUE );

   wait( ClockI->posedge_event() );
   StartCcdppO.write( FALSE );

   do {
      wait( ClockI->posedge_event() );
   }while( CcdppBusyI.read() == TRUE );

   return;
}

void CntrlToMainBusAdapter::start_uart()
{
   do {
      wait( ClockI->posedge_event() );
   }while( UartBusyI.read() == TRUE );

   StartUartO.write( TRUE );

   wait( ClockI->posedge_event() );
   StartUartO.write( FALSE );
```

```
      do {
         wait( ClockI->posedge_event() );
      }while( UartBusyI.read() == TRUE );

      return;
   }

   SC_CTOR(CntrlToMainBusAdapter) {
      StartCcdppO.initialize( FALSE );
      StartUartO.initialize( FALSE );
   }
};


////////////////////////////////////////
// MEM Module                         //
// for COMM model                     //
////////////////////////////////////////
SC_MODULE( Mem )
{
   short memory[ROW_SIZE*COL_SIZE];
   enum  state{ IDLE, READ, WRITE, DONE };

   sc_signal<state>  NextState;

   sc_in<bool>    ClockI;

   sc_in<bool>    ReqI;
   sc_in<bool>    RwI;
   sc_in<short>   AddrI;
   sc_inout<short>   DataIO;
   sc_out<bool>   ValidO;

   void main(void)
   {

      switch(NextState.read())
      {
      case IDLE:
         ValidO.write( FALSE );
         if((ReqI.read() == true) && (RwI.read() == true))
         {
            NextState = READ;
         }
         else if((ReqI.read() == true) && (RwI.read() == false))
         {
            NextState = WRITE;
         }
         break;
      case READ:
         DataIO.write( memory[AddrI.read()]  );
         ValidO.write( true );
         NextState = DONE;
         break;
      case WRITE:
         memory[AddrI.read()] = DataIO.read();
         ValidO.write( true );
         NextState = DONE;
         break;
      case DONE:
         if(ReqI.read() == false)
         {
            ValidO.write( FALSE );
            NextState = IDLE;
         }
         break;
      }
   }
   // Module Constructor
   SC_CTOR( Mem ) {
      SC_METHOD( main );
      sensitive << ClockI.pos();

      ValidO.initialize( FALSE );
   }
};


////////////////////////////////////////
// CCD Module                         //
// for Communication model            //
////////////////////////////////////////
SC_MODULE( Ccd )
{
   // CCD Ports
   sc_port<CcdToCcdBusIf>   CcdBus;
   double   simTime;

   // CCD Vars
   char buffer[ROW_SIZE][COL_SIZE+2];
   int  row, col;

   // CCD Events
   sc_event    StartPopEvt;

   // CCD Processes
   void capture(void)
   {
      CcdBus->ready();
      simTime = sc_simulation_time();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            // IMAGE array is defined in image.h
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
      notify(StartPopEvt);
   }

   void pop(void)
   {
      wait(StartPopEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            CcdBus->write( buffer[row][col] );
         }
      }
      cout << "CCD\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccd ) {
      SC_THREAD( capture );
      SC_THREAD( pop );
   }
};


////////////////////////////////////////
```

```
// CCDPP module                        //
// for Communication model             //
////////////////////////////////////////
SC_MODULE( Ccdpp )
{
   // CCDPP Ports
   sc_port<CcdppToCcdBusIf>   CcdBus;
   sc_port<CcdppToMainBusIf>  MainBus;

   // CCDPP Vars
   char buffer[COL_SIZE];
   int  row, col;
   char bias;

   double   simTime;

   // CCDPP Events
   sc_event    StartPopEvt;

   // CCDPP Processes
   void capture(void)
   {
      MainBus->ccdpp_ready();

      simTime = sc_simulation_time();

      CcdBus->start();

      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] = CcdBus->read();
         }

         // Perform Zero Bias Adjustment
         bias = CcdBus->read();
         bias = ( bias + CcdBus->read() ) / 2;
         wait(12*CLK_CYCLE, SC_NS);
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] -= bias;
            wait(4*CLK_CYCLE, SC_NS);
            MainBus->write(row*COL_SIZE+col, buffer[col]);
         }
      }

      MainBus->ccdpp_done();

      cout << "CCDPP\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccdpp ) {
      SC_THREAD( capture );
   }
};


////////////////////////////////////////
// UART module                         //
// for Communication model             //
////////////////////////////////////////
SC_MODULE( Uart )
{
   // UART Ports
   sc_port<UartToMainBusIf> MainBus;

   // UART Vars
   FILE *outputFileHandle;
   short       data;
   short i;

   double simTime;

   // UART Process
   void send(void)
   {
      MainBus->uart_ready();
      simTime = sc_simulation_time();
      for(i=0; i<(16384/2); i++)
      {
         data = MainBus->read(i);

         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[0]);
         wait(2*CLK_CYCLE, SC_NS);
         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[1]);
         wait(2*CLK_CYCLE, SC_NS);
      }
      MainBus->uart_done();
      cout << "UART\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // UART Constructor
   SC_CTOR( Uart )
   {
      outputFileHandle = fopen("uart_out.txt", "w");
      SC_THREAD( send );
   }

   // UART Destructor
   ~Uart(void)
   {
      fclose(outputFileHandle);
   }
};


////////////////////////////////////////
// CNTRL module                       //
// for Communication model            //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12
};
const unsigned char QuantShiftTable[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };
```

```
SC_MODULE(Cntrl) {
   // CNTRL Ports
   sc_port<CntrlToMainBusIf>   MainBus;

   // CNTRL Vars
   short inBuffer[8][8], outBuffer[8][8];
   short temp;
   short addr;
   int i, j, k, l;

   double   simTime;

   // Local FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {

      long s[8];
      long r;
      unsigned char a;

      for(a=0; a<8; a++)
      {
         s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
                ((img[a][1] * COS_TABLE[8+v])  >> 6) +
                ((img[a][2] * COS_TABLE[16+v]) >> 6) +
                ((img[a][3] * COS_TABLE[24+v]) >> 6) +
                ((img[a][4] * COS_TABLE[32+v]) >> 6) +
                ((img[a][5] * COS_TABLE[40+v]) >> 6) +
                ((img[a][6] * COS_TABLE[48+v]) >> 6) +
                ((img[a][7] * COS_TABLE[56+v]) >> 6);
      }

      r = 0;
      for(a=0; a<8; a++)
      {
         r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
      }

      return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
   }

   // CNTRL Process
   void main( void )
   {
      simTime = sc_simulation_time();

      // CNTRL Capture
      MainBus->start_ccdpp();

      // CNTRL Compress
      for(i=0; i<NUM_ROW_BLOCKS; i++) {

         for(j=0; j<NUM_COL_BLOCKS; j++) {

            // Push the block and perform FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                  inBuffer[k][l] = (MainBus->read( addr ) << 6);
               }
            }

            // FDCT
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  outBuffer[k][l] = F(k, l, inBuffer);
                  wait(72*CLK_CYCLE, SC_NS);
               }
            }

            // Quantize
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  outBuffer[k][l] >>= QuantShiftTable[k * 8 + l];
                  wait(4*CLK_CYCLE, SC_NS);
               }
            }

            // Pop the block and store it in buffer
            for(k=0; k<8; k++)
            {
               for(l=0; l<8; l++)
               {
                  addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
                  MainBus->write( addr, outBuffer[k][l] );
               }
            }
         }
      }

      // CNTRL Send Image
      MainBus->start_uart();
      cout << "CNTRL\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
      // Stop the simulation manually
      sc_stop();
      return;
   }

   // CNTRL Constructor
   SC_CTOR( Cntrl ) {
      SC_THREAD( main );
   }
};


////////////////////////////////////////
// Testbench                          //
// for Communication model            //
////////////////////////////////////////
int sc_main(int, char**)
{
   // Signal Instances
   sc_clock Clock("Clock", 1, SC_NS);
   sc_signal<bool>   CcdStart;
   sc_signal<bool>   CcdReady;
   sc_signal<bool>   CcdValid;
   sc_signal<char>   CcdData;

   sc_signal<bool>   MainReq;
   sc_signal<bool>   MainRw;
   sc_signal<short>  MainAddr;
   sc_signal<short>  MainData;
   sc_signal<bool>   MainValid;

   sc_signal<bool>   StartCcdpp;
   sc_signal<bool>   CcdppBusy;
```

```
sc_signal<bool>   StartUart;
sc_signal<bool>   UartBusy;

// Module Instances
Ccd   CcdInst("Ccd");
Ccdpp CcdppInst("Ccdpp");
Uart  UartInst("Uart");
Cntrl CntrlInst("Cntrl");
Mem   MemInst("Mem");

// Channel Instances
CcdToCcdBusAdapter CcdToCcdBusAdapterInst("CcdToCcdBusAdapter");
CcdppToCcdBusAdapter CcdppToCcdBusAdapterInst("CcdppToCcdBusAdapter");
CcdppToMainBusAdapter CcdppToMainBusAdapterInst("CcdppToMainBusAdapter");
UartToMainBusAdapter UartToMainBusAdapterInst("UartToMainBusAdapter");
CntrlToMainBusAdapter CntrlToMainBusAdapterInst("CntrlToMainBusAdapter");

// Bind Channels to Modules via Interfaces
CcdInst.CcdBus( CcdToCcdBusAdapterInst );
CcdppInst.CcdBus( CcdppToCcdBusAdapterInst );

CcdppInst.MainBus( CcdppToMainBusAdapterInst );
UartInst.MainBus( UartToMainBusAdapterInst );
CntrlInst.MainBus( CntrlToMainBusAdapterInst );

// Bind Signals to CCD Adapter Modules
CcdToCcdBusAdapterInst.ClockI( Clock );
CcdToCcdBusAdapterInst.StartI( CcdStart );
CcdToCcdBusAdapterInst.ReadyI( CcdReady );
CcdToCcdBusAdapterInst.ValidO( CcdValid );
CcdToCcdBusAdapterInst.DataO( CcdData );

CcdppToCcdBusAdapterInst.ClockI( Clock );
CcdppToCcdBusAdapterInst.StartO( CcdStart );
CcdppToCcdBusAdapterInst.ReadyO( CcdReady );
CcdppToCcdBusAdapterInst.ValidI( CcdValid );
CcdppToCcdBusAdapterInst.DataI( CcdData );

CcdppToMainBusAdapterInst.ClockI( Clock );
CcdppToMainBusAdapterInst.ReqO( MainReq );
CcdppToMainBusAdapterInst.RwO( MainRw );
CcdppToMainBusAdapterInst.AddrO( MainAddr );
CcdppToMainBusAdapterInst.DataIO( MainData );
CcdppToMainBusAdapterInst.ValidI( MainValid );
CcdppToMainBusAdapterInst.StartCcdppI( StartCcdpp );
CcdppToMainBusAdapterInst.CcdppBusyO( CcdppBusy );

UartToMainBusAdapterInst.ClockI( Clock );
UartToMainBusAdapterInst.ReqO( MainReq );
UartToMainBusAdapterInst.RwO( MainRw );
UartToMainBusAdapterInst.AddrO( MainAddr );
UartToMainBusAdapterInst.DataIO( MainData );
UartToMainBusAdapterInst.ValidI( MainValid );
UartToMainBusAdapterInst.StartUartI( StartUart );
UartToMainBusAdapterInst.UartBusyO( UartBusy );

CntrlToMainBusAdapterInst.ClockI( Clock );
CntrlToMainBusAdapterInst.ReqO( MainReq );
CntrlToMainBusAdapterInst.RwO( MainRw );
CntrlToMainBusAdapterInst.AddrO( MainAddr );
CntrlToMainBusAdapterInst.DataIO( MainData );
CntrlToMainBusAdapterInst.ValidI( MainValid );
CntrlToMainBusAdapterInst.StartCcdppO( StartCcdpp );
CntrlToMainBusAdapterInst.StartUartO( StartUart );
CntrlToMainBusAdapterInst.CcdppBusyI( CcdppBusy );
CntrlToMainBusAdapterInst.UartBusyI( UartBusy );

MemInst.ClockI( Clock );
MemInst.ReqI( MainReq );
```

```
   MemInst.RwI( MainRw );
   MemInst.AddrI( MainAddr );
   MemInst.DataIO( MainData );
   MemInst.ValidO( MainValid );

   // Begin TIMED Simulation (Run for 100 milliseconds)
   cout << "Simulation started at "
        << sc_simulation_time()/1000 << " us" << endl;
   sc_start(100, SC_MS);
   cout << "Simulation stopped at "
        << sc_simulation_time()/1000 << " us" << endl;

   return 0;
}
```

SystemC Communication Model after Protocol Insertion

```
//////////////////////////////////////////////////////////////////////
// File: digcam.cpp                                                 //
// Desc: SystemC Communication Model of the Digital Camera          //
//////////////////////////////////////////////////////////////////////

#include "systemc.h"
#include "image.h"

#define CLK_CYCLE 1

////////////////////////////////////////
// MEM Module                         //
// for Communication model            //
////////////////////////////////////////
SC_MODULE( Mem )
{
   short memory[ROW_SIZE*COL_SIZE];
   enum  state{ IDLE, READ, WRITE, DONE };

   sc_signal<state>  NextState;

   sc_in<bool>    ClockI;

   sc_in<bool>    ReqI;
   sc_in<bool>    RwI;
   sc_in<short>   AddrI;
   sc_inout<short>   DataIO;
   sc_out<bool>   ValidO;

   void main(void)
   {

      switch(NextState.read())
      {
      case IDLE:
         ValidO.write( FALSE );
         if((ReqI.read() == true) && (RwI.read() == true))
         {
            NextState = READ;
         }
         else if((ReqI.read() == true) && (RwI.read() == false))
         {
            NextState = WRITE;
         }
         break;
      case READ:
         DataIO.write( memory[AddrI.read()]  );
         ValidO.write( true );
         NextState = DONE;
         break;
      case WRITE:
         memory[AddrI.read()] = DataIO.read();
         ValidO.write( true );
         NextState = DONE;
         break;
      case DONE:
         if(ReqI.read() == false)
         {
            ValidO.write( FALSE );
            NextState = IDLE;
         }
         break;
      }
   }

   // Module Constructor
   SC_CTOR( Mem ) {
      SC_METHOD( main );
      sensitive << ClockI.pos();

      ValidO.initialize( FALSE );
   }
};


////////////////////////////////////////
// CCD Module                         //
// for Communication model            //
////////////////////////////////////////
SC_MODULE( Ccd )
{
   // CCD Ports
   sc_in<bool>    ClockI;
   sc_in<bool>    StartI;
   sc_in<bool>    ReadyI;
   sc_out<char>   DataO;
   sc_out<bool>   ValidO;

   // CCD Vars
   double   simTime;
   char buffer[ROW_SIZE][COL_SIZE+2];
   int  row, col;

   // CCD Events
   sc_event    StartPopEvt;

   // CCD Bus Interface Functions
   void ready( void )
   {
      do {
         wait( ClockI->posedge_event() );
      } while ( StartI.read() != TRUE );
   }

   void write(char data)
   {
      DataO.write( data );
      ValidO.write( TRUE );

      do {
         wait( ClockI->posedge_event() );
      }while( ReadyI.read() != TRUE );

      ValidO.write( FALSE );
   }

   // CCD Processes
   void capture(void)
   {
      ready();
      simTime = sc_simulation_time();
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            // IMAGE array is defined in image.h
            buffer[row][col] = IMAGE[row*(COL_SIZE+2) + col];
         }
      }
      notify(StartPopEvt);
   }

   void pop(void)
   {
      wait(StartPopEvt);
      for(row=0; row<ROW_SIZE; row++)
      {
         for(col=0; col<(COL_SIZE+2); col++)
         {
            write( buffer[row][col] );
         }
      }
      cout << "CCD\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccd ) {
      ValidO.initialize( false );
      SC_THREAD( capture );
      SC_THREAD( pop );
   }
};


////////////////////////////////////////
// CCDPP module                       //
// for Communication model            //
////////////////////////////////////////
SC_MODULE( Ccdpp )
{
   // CCDPP Ports
   sc_in<bool>    ClockI;
   sc_in<char>    CcdBusDataI;
   sc_in<bool>    CcdBusValidI;
   sc_out<bool>   CcdBusStartO;
   sc_out<bool>   CcdBusReadyO;

   sc_out<bool>   MainBusReqO;
   sc_out<bool>   MainBusRwO;
   sc_out<short>  MainBusAddrO;
   sc_inout<short>   MainBusDataIO;
   sc_in<bool>    MainBusValidI;
   sc_in<bool>    MainBusStartCcdppI;
   sc_out<bool>   MainBusCcdppBusyO;

   // CCDPP Vars
   char buffer[COL_SIZE];
   int  row, col;
   char bias;

   double   simTime;

   // CCDPP Events
   sc_event    StartPopEvt;

   // CCD Bus Interface Functions
```

```
void ccd_start( void )
{
   CcdBusStartO.write(TRUE);
   wait( ClockI->posedge_event() );
}

char ccdbus_read( void )
{
   char temp;

   CcdBusReadyO.write( TRUE );
   do {
      wait( ClockI->posedge_event() );
   }while( CcdBusValidI.read() != TRUE );

   CcdBusReadyO.write( FALSE );
   temp = CcdBusDataI.read();
   wait( ClockI->posedge_event() );

   return temp;
}

// MAIN Bus Interface Functions
void mainbus_write(short addr, short data)
{
   MainBusReqO.write( TRUE );
   MainBusRwO.write( FALSE );
   MainBusAddrO.write( addr );
   MainBusDataIO.write( data );
   do {
      wait( ClockI->posedge_event() );
   }while( MainBusValidI.read() == FALSE );

   MainBusReqO.write( FALSE );
   wait( ClockI->posedge_event() );
   return;
}

void ccdpp_ready()
{
   MainBusCcdppBusyO.write( FALSE );
   do {
      wait( ClockI->posedge_event() );
   }while( MainBusStartCcdppI.read() == FALSE );

   MainBusCcdppBusyO.write( TRUE );
   return;
}

void ccdpp_done()
{
   MainBusCcdppBusyO.write( FALSE );
   wait( ClockI->posedge_event() );
   return;
}

// CCDPP Processes
void capture(void)
{
   ccdpp_ready();

   simTime = sc_simulation_time();

   ccd_start();

   for(row=0; row<ROW_SIZE; row++)
   {
      for(col=0; col<COL_SIZE; col++)
      {
            buffer[col] = ccdbus_read();
         }

         // Perform Zero Bias Adjustment
         bias = ccdbus_read();
         bias = ( bias + ccdbus_read() ) / 2;
         wait(12*CLK_CYCLE, SC_NS);
         for(col=0; col<COL_SIZE; col++)
         {
            buffer[col] -= bias;
            wait(4*CLK_CYCLE, SC_NS);
            mainbus_write(row*COL_SIZE+col, buffer[col]);
         }
      }

      ccdpp_done();

      cout << "CCDPP\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // Module Constructor
   SC_CTOR( Ccdpp ) {
      CcdBusReadyO.initialize( FALSE );
      MainBusCcdppBusyO.initialize( FALSE );

      SC_THREAD( capture );
   }
};


///////////////////////////////////////
// UART module                       //
// for Communication model           //
///////////////////////////////////////
SC_MODULE( Uart )
{
   // UART Ports
   sc_in<bool>    ClockI;
   sc_out<bool>   ReqO;
   sc_out<bool>   RwO;
   sc_out<short>  AddrO;
   sc_inout<short>   DataIO;
   sc_in<bool>    ValidI;
   sc_in<bool>    StartUartI;
   sc_out<bool>   UartBusyO;

   // UART Vars
   FILE *outputFileHandle;
   short       data;
   short i;

   double simTime;

   // MAIN Bus Interface Functions
   short read(short addr)
   {
      short  temp;

      ReqO.write( TRUE );
      RwO.write( TRUE );
      AddrO.write( addr );
      do {
         wait( ClockI->posedge_event() );
      }while( ValidI.read() == FALSE );

      temp = DataIO.read();
      ReqO.write( FALSE );
      wait( ClockI->posedge_event() );

      return temp;
   }

   void uart_ready()
   {
      UartBusyO.write( FALSE );
      do {
         wait( ClockI->posedge_event() );
      }while( StartUartI.read() == FALSE );

      UartBusyO.write( TRUE );
      return;
   }

   void uart_done()
   {
      UartBusyO.write( FALSE );
      wait( ClockI->posedge_event() );
      return;
   }

   // UART Process
   void send(void)
   {
      uart_ready();
      simTime = sc_simulation_time();
      for(i=0; i<(16384/2); i++)
      {
         data = read(i);

         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[0]);
         wait(2*CLK_CYCLE, SC_NS);
         fprintf(outputFileHandle, "%i\n", (int)((char*)&data)[1]);
         wait(2*CLK_CYCLE, SC_NS);
      }
      uart_done();
      cout << "UART\tdone at "
           << (sc_simulation_time()/1000) << " us\t"
           << "execution time = "
           << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   }

   // UART Constructor
   SC_CTOR( Uart )
   {
      UartBusyO.initialize( FALSE );
      outputFileHandle = fopen("uart_out.txt", "w");
      SC_THREAD( send );
   }

   // UART Destructor
   ~Uart(void)
   {
      fclose(outputFileHandle);
   }
};


////////////////////////////////////////
// CNTRL module                       //
// for Communication model            //
////////////////////////////////////////
const short COS_TABLE[64] = {
            64,  62,  59,  53,  45,  35,  24,  12,
            64,  53,  24, -12, -45, -62, -59, -35,
            64,  35, -24, -62, -45,  12,  59,  53,
            64,  12, -59, -35,  45,  53, -24, -62,
            64, -12, -59,  35,  45, -53, -24,  62,
            64, -35, -24,  62, -45, -12,  59, -53,
            64, -53,  24,  12, -45,  62, -59,  35,
            64, -62,  59, -53,  45, -35,  24, -12
};
const unsigned char QuantShiftTable[64] = {
                    0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 1, 1, 1, 1,
                    1, 1, 1, 1, 1, 1, 2, 2,
                    2, 2, 2, 2, 2, 2, 2, 3,
                    3, 3, 3, 3, 3, 3, 3, 4,
                    4, 4, 4, 4, 4, 4, 5, 5,
                    5, 5, 5, 5, 6, 6, 6, 6,
                    6, 7, 7, 7, 7, 8, 8, 8 };

SC_MODULE(Cntrl) {
   // CNTRL Ports
   sc_in<bool>    ClockI;
   sc_out<bool>   ReqO;
   sc_out<bool>   RwO;
   sc_out<short>  AddrO;
   sc_inout<short>   DataIO;
   sc_in<bool>    ValidI;
   sc_out<bool>   StartCcdppO;
   sc_out<bool>   StartUartO;
   sc_in<bool>    CcdppBusyI;
   sc_in<bool>    UartBusyI;

   // CNTRL Vars
   short inBuffer[8][8], outBuffer[8][8];
   short temp;
   short addr;
   int i, j, k, l;

   double   simTime;

   // Local FDCT Functions
   unsigned char C(int h) {
      return h ? 64 : 5;
   }

   int F(int u, int v, short img[8][8]) {

      long s[8];
      long r;
      unsigned char a;

      for(a=0; a<8; a++)
      {
         s[a] = ((img[a][0] * COS_TABLE[v])    >> 6) +
                ((img[a][1] * COS_TABLE[8+v])  >> 6) +
                ((img[a][2] * COS_TABLE[16+v]) >> 6) +
                ((img[a][3] * COS_TABLE[24+v]) >> 6) +
                ((img[a][4] * COS_TABLE[32+v]) >> 6) +
                ((img[a][5] * COS_TABLE[40+v]) >> 6) +
                ((img[a][6] * COS_TABLE[48+v]) >> 6) +
                ((img[a][7] * COS_TABLE[56+v]) >> 6);
      }

      r = 0;
      for(a=0; a<8; a++)
      {
         r += (s[a] * COS_TABLE[a * 8 + u]) >> 6;
      }

      return (short)((((((r * 16) >> 6) * C(u)) >> 6) * C(v)) >> 6);
   }
```

```
// MAIN Bus Interface Functions
short read(short addr)
{
   short temp;

   ReqO.write( TRUE );
   RwO.write( TRUE );
   AddrO.write( addr );
   do {
      wait( ClockI->posedge_event() );
   }while( ValidI.read() == FALSE );

   temp = DataIO.read();
   ReqO.write( FALSE );
   wait( ClockI->posedge_event() );

   return temp;
}

void write(short addr, short data)
{
   ReqO.write( TRUE );
   RwO.write( FALSE );
   AddrO.write( addr );
   DataIO.write( data );
   do {
      wait( ClockI->posedge_event() );
   }while( ValidI.read() == FALSE );

   ReqO.write( FALSE );
   wait( ClockI->posedge_event() );
   return;
}

void start_ccdpp()
{
   do {
      wait( ClockI->posedge_event() );
   }while( CcdppBusyI.read() == TRUE );

   StartCcdppO.write( TRUE );

   wait( ClockI->posedge_event() );
   StartCcdppO.write( FALSE );

   do {
      wait( ClockI->posedge_event() );
   }while( CcdppBusyI.read() == TRUE );

   return;
}

void start_uart()
{
   do {
      wait( ClockI->posedge_event() );
   }while( UartBusyI.read() == TRUE );

   StartUartO.write( TRUE );

   wait( ClockI->posedge_event() );
   StartUartO.write( FALSE );

   do {
      wait( ClockI->posedge_event() );
   }while( UartBusyI.read() == TRUE );

   return;
}
```

```
// CNTRL Process
void main( void )
{
   simTime = sc_simulation_time();

   // CNTRL Capture
   start_ccdpp();

   // CNTRL Compress
   for(i=0; i<NUM_ROW_BLOCKS; i++) {

      for(j=0; j<NUM_COL_BLOCKS; j++) {

         // Push the block and perform FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
               inBuffer[k][l] = (read( addr ) << 6);
            }
         }

         // FDCT
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] = F(k, l, inBuffer);
               wait(72*CLK_CYCLE, SC_NS);
            }
         }

         // Quantize
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               outBuffer[k][l] >>= QuantShiftTable[k * 8 + l];
               wait(4*CLK_CYCLE, SC_NS);
            }
         }

         // Pop the block and store it in buffer
         for(k=0; k<8; k++)
         {
            for(l=0; l<8; l++)
            {
               addr = (COL_SIZE * (i * 8 + k)) + (j * 8 + l);
               write( addr, outBuffer[k][l] );
            }
         }
      }
   }

   // CNTRL Send Image
   start_uart();
   cout << "CNTRL\tdone at "
        << (sc_simulation_time()/1000) << " us\t"
        << "execution time = "
        << (sc_simulation_time() - simTime)/1000 << " us" << endl;
   // Stop the simulation manually
   sc_stop();
   return;
}

// CNTRL Constructor
SC_CTOR( Cntrl ) {
      StartCcdppO.initialize( FALSE );
      StartUartO.initialize( FALSE );
      SC_THREAD( main );
   }
};


////////////////////////////////////////
// Testbench                          //
// for Communication model            //
////////////////////////////////////////
int sc_main(int, char**)
{
   // Signal Instances
   sc_clock Clock("Clock", 1, SC_NS);

   sc_signal<bool>   CcdStart;
   sc_signal<bool>   CcdReady;
   sc_signal<bool>   CcdValid;
   sc_signal<char>   CcdData;

   sc_signal<bool>   MainReq;
   sc_signal<bool>   MainRw;
   sc_signal<short>  MainAddr;
   sc_signal<short>  MainData;
   sc_signal<bool>   MainValid;
   sc_signal<bool>   StartCcdpp;
   sc_signal<bool>   CcdppBusy;
   sc_signal<bool>   StartUart;
   sc_signal<bool>   UartBusy;

   // Module Instances
   Mem   MemInst("Mem");
   Ccd   CcdInst("Ccd");
   Ccdpp CcdppInst("Ccdpp");
   Uart  UartInst("Uart");
   Cntrl CntrlInst("Cntrl");

   // Bind Bus Signals to Modules
   MemInst.ClockI( Clock );
   MemInst.ReqI( MainReq );
   MemInst.RwI( MainRw );
   MemInst.AddrI( MainAddr );
   MemInst.DataIO( MainData );
   MemInst.ValidO( MainValid );

   CcdInst.ClockI( Clock );
   CcdInst.StartI( CcdStart );
   CcdInst.ReadyI( CcdReady );
   CcdInst.DataO( CcdData );
   CcdInst.ValidO( CcdValid );

   CcdppInst.ClockI( Clock );
   CcdppInst.CcdBusStartO( CcdStart );
   CcdppInst.CcdBusReadyO( CcdReady );
   CcdppInst.CcdBusDataI( CcdData );
   CcdppInst.CcdBusValidI( CcdValid );
   CcdppInst.MainBusReqO( MainReq );
   CcdppInst.MainBusRwO( MainRw );
   CcdppInst.MainBusAddrO( MainAddr );
   CcdppInst.MainBusDataIO( MainData );
   CcdppInst.MainBusValidI( MainValid );
   CcdppInst.MainBusStartCcdppI( StartCcdpp );
   CcdppInst.MainBusCcdppBusyO( CcdppBusy );

   UartInst.ClockI( Clock );
   UartInst.ReqO( MainReq );
   UartInst.RwO( MainRw );
   UartInst.AddrO( MainAddr );
   UartInst.DataIO( MainData );
   UartInst.ValidI( MainValid );
   UartInst.StartUartI( StartUart );
   UartInst.UartBusyO( UartBusy );

   CntrlInst.ClockI( Clock );
   CntrlInst.ReqO( MainReq );
   CntrlInst.RwO( MainRw );
   CntrlInst.AddrO( MainAddr );
   CntrlInst.DataIO( MainData );
   CntrlInst.ValidI( MainValid );
   CntrlInst.StartCcdppO( StartCcdpp );
   CntrlInst.StartUartO( StartUart );
   CntrlInst.CcdppBusyI( CcdppBusy );
   CntrlInst.UartBusyI( UartBusy );

   // Begin TIMED Simulation (Run for 100 milliseconds)
   cout << "Simulation started at "
        << sc_simulation_time()/1000 << " us" << endl;
   sc_start(100, SC_MS);
   cout << "Simulation stopped at "
        << sc_simulation_time()/1000 << " us" << endl;

   return 0;
}
```

APPENDIX C INPUT IMAGE ARRAY

```
///////////////////////////////////////////////////////////////////
// File: image.h                                                 //
// Desc: Array of input image data used by the CCD module.       //
// NOTE: The file is the same for all SpecC and SystemC models.  //
///////////////////////////////////////////////////////////////////

#define ROW_SIZE  64
#define COL_SIZE  128
#define NUM_ROW_BLOCKS  (ROW_SIZE / 8)
#define NUM_COL_BLOCKS  (COL_SIZE / 8)

const char IMAGE[ROW_SIZE * (COL_SIZE+2)] = {
           86,   92,  -63,   -5,  -98,   19,  -87,  -47,   45,  -17,
           54, -114, -114,  -60, -106, -108,  -52,  -39,  -92,   67,
           -8,  115,   65,  -26,   91,   33,  114,   99,  121,  -56,
.
.
.
File not listed in its entirety.
```

BIBLIOGRAPHY

[1] W. Wolf, "A Decade of Hardware/Software Codesign," *IEEE Computer*, vol. 36, no. 4, Apr. 2003, pp. 38-43.

[2] SpecC Technology Open Consortium (STOC), Date Retrieved: July 20 2005; http://www.specc.org.

[3] Open SystemC Initiative (OSCI), Date Retrieved: July 20 2005; http://www.systemc.org.

[4] D. Gajski, J. Zhu, and R. Dömer, *The SpecC+ Language*, tech. report ICS-97-15, Dept. Information and Computer Science, Univ. of California, Irvine, 1997.

[5] D. Gajski, G. Aggarwal, E. Chang, R. Dömer, T. Ishii, J. Kleinsmith, and J. Zhu, *Methodology for Co-design of Embedded Systems*, tech. report ICS-98-07, Dept. Information and Computer Science, Univ. of California, Irvine, 1998.

[6] R. Dömer, *System-level Modeling and Design with the SpecC Language*, doctoral dissertation, Dept. Computer Science, Univ. of Dortmund, 2000.

[7] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *Spec C: Specification Language and Methodology*, Kluwer Academic Publishers, 2000.

[8] A. Gerstlauer, *SpecC Modeling Guidelines*, tech. report CECS-02-16, Center for Embedded Computer Systems, Univ. of California, Irvine, 2002.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[10] L. Cai, S. Verma, and D. Gajski, *Comparison of SpecC and SystemC Languages for System Design*, tech. report CECS-03-11, Center for Embedded Computer Systems, Univ. of California, Irvine, 2003.

[11] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, 2002.

[12] F. Vahid and T. Givargis, “DIGCAM Example source files”, Date Retrieved: July 20 2005; http://www.cs.ucr.edu/content/esd/digcam/DIGCAM/.

[13] C. Hylands, E.A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, *Overview of the Ptolemy Project*, tech. memo UCB/ERL M03/25, Dept. of Elec. Eng. and Comp. Science, Univ. California, Berkeley, 2003.

[14] J. T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, “Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems,” *Int. Journal of Computer Simulation*, vol. 4, no. 155, Apr. 1994, pp. 155-182.

[15] C. Hylands, E.A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng, *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*, tech. memo UCB ERL M02/23, Dept. of Electrical Eng. and Computer Science, Univ. California, Berkeley, 2002.

[16] P. Baldwin, S. Kohli, E.A. Lee, X. Liu, and Y. Zhao, “Modeling of Sensor Nets in Ptolemy II,” *Proc. 3rd Int’l. Symp. Information Processing in Sensor Networks* (IPSN’04), ACM Press, 2004, pp. 359-368.

[17] J. Yeh, *Image and Video Processing Libraries in Ptolemy II*, master’s thesis, Dept. of Electrical Eng. and Computer Science, Univ. California, Berkeley, 2003.

[18] POLIS: A Framework for Hardware-Software Co-Design of Embedded Systems, Date Retrieved: July 20 2005; http://www-cad.eecs.berkeley.edu/~polis/.

[19] G. Berry, *The Esterel v5 Language Primer*, ver. 5.91, Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, Sophia-Antipolis, 2000.

[20] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa, “VIS: A System for Verification and Synthesis,” *Proc. 8th Int’l Conf. on Computer Aided Verification* (CAV’96), Springer Verlag, 1996, pp. 428-432.

[21] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment," *IEEE Computer*, vol. 36, no. 4, pp. 45-52.

[22] D. Densmore, *Metropolis Architecture Refinement Styles and Methodology*, tech. report UCB ERL M04/36, Dept. of Electrical Eng. And Computer Science, Univ. California, Berkeley, 2004.

[23] L. Cai, D. Gajski, P. Kritzinger, and M. Olivares, "Top-Down System Level Design Methodology Using SpecC, VCC, and SystemC," *Proc. Conf. On Design, Automation and Test in Europe* (DATE'02), IEEE Computer Society, 2002, pp. 1137.

[24] T. Moore, Y. Vanderperren, G. Sonck, P. Van Oostende, and W. Dehaene, "A Design Methodology for the Development of a Complex System-On-Chip Using UML and Executable System Models," *5th Forum on Specification and Design Languages* (FSL'02), 2002.

[25] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio, "A SoC Design Methdology Involving a UML 2.0 Profile for SystemC," *Proc. Conf. on Design, Automation and Test in Europe* (DATE'05), IEEE Computer Society, vol. 2, no. 2, 2005, pp. 704-709.

[26] Y. Vanderperren and W. Dehaene, "UML 2 and SysML: an Approach to Deal with Complexity in SoC/NoC Design," *Proc. Conf. on Design, Automation and Test in Europe* (DATE'05), IEEE Computer Society, vol. 2, no. 2, 2005, pp. 716-717.

[27] S. Abdi, D. Shin, and D. Gajski, "Automatic Communicatoin Refinement for System Level Design," *Proc. 40th Intl. Conf. on Design Automation* (DAC'03), ACM Press, 2003, pp. 300-305.

[28] D. Shin, S. Abdi, and D. Gajski, "Automatic Generation of Bus Functional Models from Transaction Level Models," *Proc. Conf. on Asia South Pacific Design Automation* (ASPDAC'04), IEEE, 2004, pp. 756-758.

[29] A.A. Jerraya, S. Yoo, A. Baghdadi, and D. Lyonnard, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," *Proc. 38th Intl. Conf. on Design Automation* (DAC'01), ACM Press, 2001, pp. 518-523.

[30] R. Passerone, J.A. Rowson, and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," *Proc. 35th Intl. Conf. on Design Automation* (DAC'98), ACM Press, 1998, pp. 8-13.

[31] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R.K. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform," *Proc. 10th Intl. Conf. on Hardware Software Codesign* (CODES'02), ACM Press, 2002, pp. 151-156.

[32] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systematic Embedded Software Generation from SystemC," *Proc. Conf. on Design, Automation and Test in Europe* (DATE'03), IEEE Computer Society, 2003.

[33] H. Yu, R. Dömer, and D. Gajski, "Embedded Software Generation from System Level Design Languages," *Proc. Conf. on Asia South Pacific Design Automation* (ASPDAC'04), IEEE, 2004, pp. 463-468.