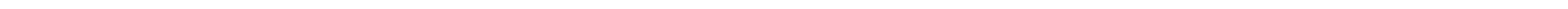
Center for Embedded Computer Systems
University of California, Irvine

System-on-Chip Communication Modeling Style Guide

Dongwan Shin
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-25
July, 2004



System-on-Chip Communication Modeling Style Guide

Dongwan Shin
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-25
July, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
+1 (949) 824-8919

http://www.cecs.uci.edu

Abstract

Within SoC Design Environment (SCE), starting from an initial system specification, an implementation of the system is created through a series of interactive and automated steps by gradually synthesizing and assembling a system design using components taken out of a set of databases.

SCE uses four models to reflect design decisions during system-level synthesis: specication model, carchitecture model, network model and communication model. The communication model can be pin-accurate or transaction-level model. This report defines and describes pin-accurate communication model required for system-on-chip (SoC) design.

Generally, pin-accurate communication models need to represent processing elements (PEs), memories, communication elements (CEs) and bus wires connecting components. In this report we aim to provide an exhaustive list of requirements for pin-accurate communication in an automated SoC design flow using the example of concrete models. Specifically, the communication model in this report is used successfully in our SCE.

Contents

1 Introduction **1**

2 Overview of Communication Model **2**
2.1 Layered Structure of Communication Model 5
2.1.1 Layers of Protocol Stack 5
2.1.2 Layered Shells of PEs 8

3 Processing Elements **10**
3.1 Programmable PEs 10
3.2 Hardware PEs 15
3.2.1 Hardware PE With a Local Memory 15

4 Memories **17**

5 Communication Elements **17**
5.1 Bridges 21
5.2 Transducers 21
5.3 Arbiter 21
5.4 Interrupt Controller 25

6 Bus Wires **26**

7 Example **27**

References **29**

List of Figures

1 SCE design flow. 1
2 Top-level structure of a communication model. 3
3 Top-level code of a communication model. 4
4 A design in the pin-accurate communication model (I). 6
5 A design in the pin-accurate communication model (II). 7
6 An example of programmable PE in communication model. 11
7 An example of a programmable PE in SpecC (HAL shell). 12
8 An example of a programmable PE in SpecC (HW layer shell). 13
9 An example of a programmable PE in SpecC (BF layer shell). 14
10 An example of a hardware PE in the communication model. 16
11 An example of hardware PE with local memory in communication model. 16
12 An example of a hardware PE with a local memory in SpecC (application layer shell). 18
13 An example of a hardware PE with a local memory in SpecC (BF layer shell). 19
14 An example of a shared memory in the communication model. 20
15 An example of a shared memory in SpecC. 20
16 An example of bridge in communication model. 21
17 An example of bridge in SpecC (I). 22
18 An example of bridge in SpecC (II). 23
19 An example of a transducer in the communication model. 23
20 An example of transducer in SpecC (network model). 23
21 An example of transducer in SpecC (BF layer shell). 24
22 An example of arbiter. 25
23 An example of interrupt controller. 26
24 Communication model example. 28

System-on-Chip Communication Modeling Style Guide

D. Shin, A. Gerstlauer, R. Dömer, D. Gajski
Center for Embedded Computer Systems
University of California, Irvine

July, 2004

1 Introduction

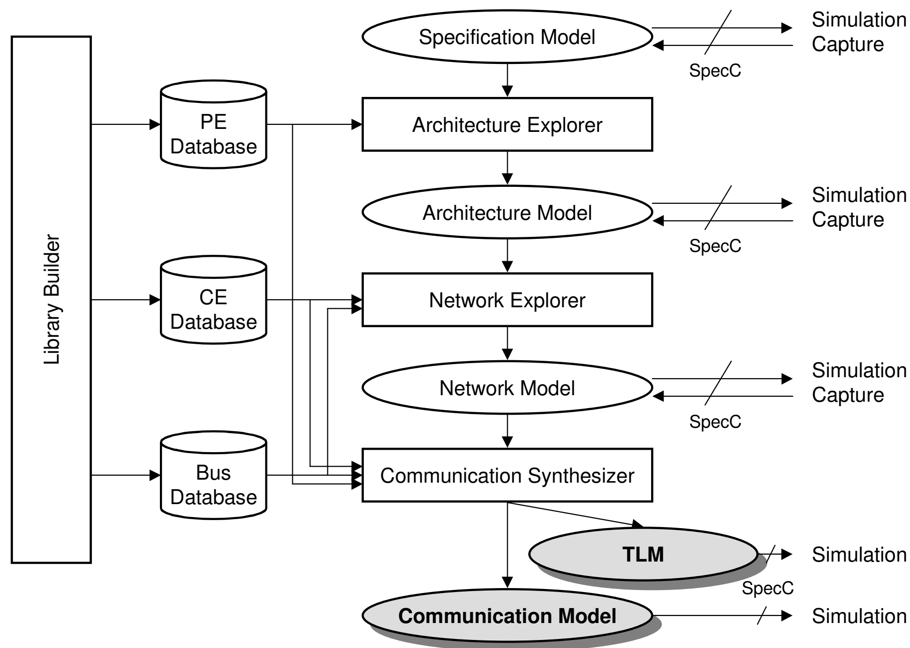


Figure 1: SCE design flow.

SoC Design Environment (SCE) [4] is an environment for capturing the architecture and platform specification of embedded computer systems. It supports the design of such embedded systems from the specification down to the communication model. It does so by capturing of design decisions and automatic generation of new models as shown in Figure 1. SCE follows a **Specify-Explore-Refine** methodology. The design flow starts with a model representing the design functionality (**Specify**). At each following design step, SCE users first explore the design space (**Explore**) and then make design decisions. Integrating those decisions, SCE then automatically generates a new model at lower level of abstraction (**Refine**).

In the SCE design flow (Figure 1), four models are used for the representation of a design at different levels of abstraction. Each design model is executable; it can be simulated to verify the correctness of the design and obtain design performance metrics at each design step.

The specification model [5] is the most abstract model, which serves the as an input to SCE tools. It is a purely functional model that captures the functionality of the desired design and should not imply any implementation details.

The architecture model [6] reflects the allocation of system components and the mapping of the specified functionality onto the allocated components. The communication between those components is still described very abstract by message passing channels.

The network model [7] reflects the communication network of the design. It represents the allocation and selection of network stations and logical links between them. While the communication between components in the architecture model is captured end-to-end, this communication is refined down to point-to-point in the network model.

Finally, the communication model incorporates bus protocols into the model. The communication model can be pin-accurate or a transaction-level model [8]. The transaction level model abstracts away the pin-accurate protocol details and thereby gains higher simulation speeds.

All models are captured in SpecC [1], therefore they do have to adhere to the syntax and semantics of the SpecC language. It is recommended that the designer starts with the specification model and later uses the SCE tools to automatically generate lower level models. However, the SCE tools also support manually written low level models as long as they obey certain modeling rules. This report defines the modeling style required for the a SCE communication model (pin-accurate communication model), which is highlighted in the Figure 1.

This report can be used for two purposes. First, it can help user to interpret the code of the communication model, which is automatically generated by the communication synthesizer. Second, it gives the user guidelines to manually write a valid communication model that is acceptable to the SCE tools.

The rest of the report is organized as follows: First, Section 2 shows the overall structure of the communication model. The major elements of a communication model are described one by one in detail. Section 3 describes the guidelines to model processing elements shown in the communication model. Section 4 describes the modeling of shared memories in the communication model. Section 5 describes the modeling of communication elements, such as bridges and transducers in the communication model. Section 6 describes the modeling of bus wires in the communication model. Finally, Section 7 describes a example of communication model in SCE.

2 Overview of Communication Model

The communication model is the final result of the system synthesis process and defines the structure of the system architecture in terms of components and connections. Computation in the specification has been mapped onto components and communication onto busses. At the top-level of the behavior hierarchy, a design consists of concurrent, non-terminating system components that communicate through busses.

Inside the components, behavior models of bus drivers and bus interfaces (protocol stacks) describe the communication functionality of the component, i.e. implementations of all communication layers in the protocol stacks are inlined into the components in the form of channel adapters and implement the abstract transaction in architecture model over busses. Those bus adapters (a stack of communication layers) specify how the component implements the semantics of the abstract transactions by driving and sampling the wires of the system bus. Behavioral blocks inside the component, in turn, connect to the equivalent message passing channel interface provided by bus adapters.

Component communication models can be thought of as additional communication layers that wrap

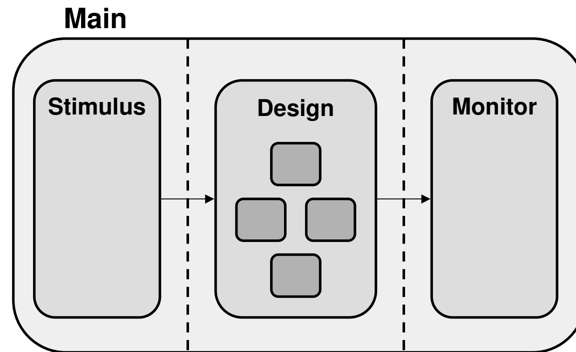


Figure 2: Top-level structure of a communication model.

around the component behavioral model. A communication model can consist of several layers of behaviors that create a hierarchy or tree of behavior instantiations. At minimum, a top level bus functional layer shell has to exist that provides a pin accurate model of the component. Through this layer and its optional sublayer instance hierarchy, the bus functional component model describes the communication behavior of the component at its pins and it has to provide the same computational functionality as the behavioral model of the component.

Figure 2 and Figure 3 show a template of a valid communication model. A communication model has to be an executable SpecC model, therefore it has to define a `Main` behavior. A communication model is composed of three parts: a stimuli generator, a monitor and a actual design unit as shown in Figure 2. The stimuli generator (`Stimulus`) supplies test vector to the input ports of the design. The output produced by the design unit is observed and validated by the monitor (`Monitor`). The design unit (`Design`) is the target of the design space exploration in SCE environment. SCE tools require the design unit to follow certain modeling rules and restrictions. Note that the modeling rules and restrictions defined in this report only apply to the design unit, since the stimuli generator and monitor will not be considered and touched by SCE tools. Therefore, the stimuli generator and monitor can be freely described using any valid SpecC code.

In general, it is hard for SCE tools to determine which behaviors are part of the design unit. Thus the user has to specify the behaviors comprising the design unit. In practice, this is realized by attaching a pre-defined annotation to the communication model.

Rule 1 *A communication model has an annotation `_SCE_TOP_LEVEL`, which contains the name of the top-level behavior of the design unit.*

For example in Figure 3, the annotation in the line 39, specifies the design unit. Once the top-level behavior of the design unit is specified, the SCE tools can conveniently figure out all other behaviors that belong to the design unit.

The design unit of the communication model must obey the following rules.

Rule 2 *Design unit has exactly the same set of ports as the corresponding behavior in the specification model.*

Leaving the interface of the design unit unchanged, allows connecting it to the testbench behaviors (stimulus and monitor) without changing the latter. Thus it allows simulation of the communication model.

Rule 3 *Design unit has exactly one method, the `main()` method, which contains exactly one statement that is a `par` statement.*

```
1  import "c_double_handshake";
2
3  behavior Stimulus(i_sender input) {           // Stimuli creator
4    void main(void) {
5      // while(...) { ... ; input.send(...) ; ... }
6    }
7  };
8
9  behavior Monitor(i_receiver output) {            // Output monitor
10   void main(void) {
11     // while(...) { ... ; output.receive(...) ; ... }
12   }
13 };
14
15 behavior Design(i_receiver input, i_sender output) {  // System design
16   // ...
17
18   void main(void) {
19     // fsm { ... }
20   }
21 };
22
23 behavior Main() {                        // Top level
24   c_double_handshake input, output;
25
26   Stimulus stimulus(input);
27   Design   design(input, output);
28   Monitor  monitor(output);
29
30   int main(void) {
31     par {
32       stimulus.main();
33       design.main();
34       monitor.main();
35     }
36   }
37 };
38
39 note _SER_TOP_LEVEL = "Design";
```

Figure 3: Top-level code of a communication model.

Note that by the definition of a hierarchical behavior, each sub-behavior instance inside the design unit can be called at most once in the `par` statement. For example, having two `PE.main()` calls in the `par` statement is not allowed.

Rule 4 *A design unit has a set of sub-behaviors instances and variables in pin-accurate communication model.*

The sub-behaviors of the design unit may be PE behaviors, memory behaviors, or communication elements behaviors. They are defined in more details in the following sections. In the pin-accurate communication model, variables represent the bus wires connecting PEs, memories and CEs as shown in Figure 4 and Figure 5.

Rule 5 *A design unit contains a set of PEs, memories, CEs and bus wires at the top-level behavior. They are connected by wires.*

The design unit usually contains finer model elements (system components). Those finer model elements capture both the computation architecture and the communication network. The system components are:

- Processing element behaviors model the processing elements (PEs) allocated to perform the desired computation.
- Memory behaviors model the shared memories allocated to store data shared by PEs.
- Communication element behaviors model the bridge and transducer interfacing between different communication protocols.
- Protocol channels or bus wires model the connection between PEs, memories and CEs.

The model elements are defined one by one in the following sections.

2.1 Layered Structure of Communication Model

In order to separate the concerns of modeling different aspects of a functionality in terms of communication and computation, we take a layered approach, which has been well known in the network community for describing protocols.

2.1.1 Layers of Protocol Stack

For the communication functionality, we will implement several layers of protocol stack based on OSI reference model: *application layer*, *presentation layer*, *session layer*, *transport layer*, *network layer*, *link layer*, *media access layer*, and *protocol layer*. The upper part of the protocol stack (from application layer to network layer) is implemented during the network exploration. The remaining part of the protocol stack (from link layer to protocol layer) will be inserted by communication synthesizer. In the following, we will describe and define each layer in more detail.

The application layer corresponds to the computation functionality of the system, which defines the behavior of the application implemented by the system design. The application layers describe the processing of data in the system components that exchange data by passing messages over channels.

The presentation layer is used to describe the formatting between typed data in the application and typeless byte-stream transferred through the network. The presentation layer performs the type conversion. If the data in the application is already untyped, the adapter channels may be omitted.

```
1  behavior Design(void)
2  {
3      // CPUBus wires
4      signal bit[35:0] GA_CPUBus;
5      signal bit[7:0] GBE_CPUBus = 00000000b;
6      signal bit[63:0] GDIN_CPUBus;
7      signal bit[63:0] GDOUT_CPUBus;
8      signal bit[0:0] GRD_CPUBus = 1b;
9      signal bit[0:0] GWR_CPUBus = 1b;
10     signal bit[0:0] GACK_CPUBus = 1b;
11     signal bit[0:0] GREQ0_CPUBus = 1b;
12     signal bit[0:0] GGNT0_CPUBus = 1b;
13     signal bit[0:0] GREQ1_CPUBus = 1b;
14     signal bit[0:0] GGNT1_CPUBus = 1b;
15     signal bit[1:0] GINT0_CPUBus = 11b;
16     signal bit[1:0] GINT1_CPUBus = 11b;
17     signal bit[1:0] GINT2_CPUBus = 11b;
18     signal bit[1:0] GINT3_CPUBus = 11b;
19
20     // SlaveBus wires
21     signal bit[35:0] GA_SlaveBus;
22     signal bit[7:0] GBE_SlaveBus = 00000000b;
23     signal bit[63:0] GDIN_SlaveBus;
24     signal bit[63:0] GDOUT_SlaveBus;
25     signal bit[0:0] GRD_SlaveBus = 1b;
26     signal bit[0:0] GWR_SlaveBus = 1b;
27     signal bit[0:0] GACK_SlaveBus = 1b;
28
29     // SRAMBus wires
30     signal bit[18:0] A_SRAMBus;
31     signal bit[0:0] CS_SRAMBus = 1b;
32     signal bit[7:0] IO_SRAMBus;
33     signal bit[0:0] OE_SRAMBus = 1b;
34     signal bit[0:0] WE_SRAMBus = 1b;
35
36     Toshiba_TX49H2_BF CPU(
37         // CPUBus master interface
38         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus, GRD_CPUBus,
39         GWR_CPUBus, GACK_CPUBus,
40         GREQ0_CPUBus, GGNT0_CPUBus, GREQ1_CPUBus, GGNT1_CPUBus,
41         GINT0_CPUBus, GINT1_CPUBus, GINT2_CPUBus, GINT3_CPUBus
42     );
```

Figure 4: A design in the pin-accurate communication model (I).

```
43     HW_Standard_DMA_BF DMA(
44         // CPUBus master/slave interface
45         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus, GRD_CPUBus,
46         GWR_CPUBus, GACK_CPUBus, GINT0_CPUBus, GREQ_CPUBus, GGNT_CPUBus,
47     );
48     HW_Standard_HW_BF HW(
49         // SlaveBus slave interface
50         GA_SlaveBus, GDOUT_SlaveBus, GDIN_SlaveBus, GBE_SlaveBus,
51         GRD_SlaveBus, GWR_SlaveBus, GACK_SlaveBus, GINT1_CPUBus
52     );
53     SRAM_BF SRAM(
54         // SRAMBus slave interface
55         A_SRAMBus, IO_SRAMBus, CS_SRAMBus, OE_SRAMBus, WE_SRAMBus
56     );
57     SRAMCtrl_BF SRAMCtrl(
58         // CPUBus slave interface
59         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus, GRD_CPUBus,
60         GWR_CPUBus, GACK_CPUBus,
61         // SRAMBus master interface
62         A_SRAMBus, IO_SRAMBus, CS_SRAMBus, OE_SRAMBus, WE_SRAMBus
63     );
64     Bridge_BF Bridge(
65         // CPUBus slave interface
66         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus, GRD_CPUBus,
67         GWR_CPUBus,
68         // SlaveBus master interface
69         GA_SlaveBus, GDOUT_SlaveBus, GDIN_SlaveBus, GBE_SlaveBus,
70         GRD_SlaveBus, GWR_SlaveBus,
71     );
72
73     void main(void) {
74         par {
75             CPU.main();
76             HW.main();
77             DMA.main();
78             Bridge.main();
79             SRAM.main();
80             SRAMCtrl.main();
81         }
82     }
83 };
```

Figure 5: A design in the pin-accurate communication model (II).

The session layer establishes a connection between components and is responsible for end-to-end synchronization. The session layer marks the interface between application and operating system. In the session layer, different streams originating from different sources may be combined, hence messages of different channels need to be multiplexed on shared streams.

The transport layer is needed to break up the byte stream into smaller packets that will be routed over the network if a transducer participates in the data transfer. The transport layer is modeled as a hierarchical behavior. It contains an instantiation of the transport layer behavior. It may also contain a set of adapter channels that perform the packetization.

The network layer determines routing of data packets from sender to receiver. Assuming reliable stations and links, routing in SoCs is usually done statically, i.e. all packets of a channel take the same fixed, pre-determined path through the system. In a standard bus-based communication, a dedicated logical link is established between two stations for each channel routed through them, assuming the underlying layers support a large enough number of simultaneous logical links between all pairs of stations. The network layer may also contain a set of adapter channels that perform the routing.

The link layer provides services to establish logical links between adjacent stations and to exchange data packets those. The link layer is the highest layer of drivers for external interfaces and peripherals in the operating system. The link layer defines the type of a station (e.g. master/slave) for each of its incoming and outgoing links. As a result, it implements any necessary synchronization between stations by interrupt or acknowledgment. As part of link layer, polling might be required for synchronization, for example, in case of interrupt sharing.

The media access layer is responsible for slicing blocks of bytes into transfer units available at the physical interface. In the process, its implementation has to guarantee that the rates of successive transfers within a block match for all communication partners. Furthermore, the media access layer determines who is allowed to access a shared medium at a given point in time. In other words, it resolves the simultaneous access of bus masters by means of arbitration. Depending on the chosen arbitration scheme, additional arbitration stations are introduced into the system as part of the media access layer.

The protocol layer implements the transfer protocols over a medium in the hardware of component's interface. It is responsible for driving and sampling the external pins according to the protocol timing diagrams and thereby matching the transmission timing on the sender and receiver side. As part of the protocol layer, protocol converters are introduced into the system. Protocol converter connects two busses with different protocols by translating different protocols.

Due to characteristics of standard bus-based SoC communication, layers have been tailored specifically to these requirements. For example, in a reliable bus-based communication architecture, error correction, flow control, buffering or dynamic routing are not required. Therefore, the transport layer is empty and the network layer is largely simplified.

2.1.2 Layered Shells of PEs

In order to separate the concerns of modeling different aspects of a programmable PE, we also take a layered approach [9]. From inside out, five layers have to be followed to model a PE: *application layer shell*, *operating system layer shell*. *hardware abstraction layer shell*. *hardware layer shell*, and *bus-functional layer shell*.

The inner-most application layer shell encapsulates the computation required by the application that is executed on the PE. In general, the application layer shell is hierarchically composed of smaller behaviors, each contains a piece of the computation assigned to the PE. For inter-behavior communication inside the application layer, both channels and variables can be connected to the behavior ports. The modeling styles inside the application layer can be found in SpecC specification model reference manual. However, the application layer shell can only have interface type ports and no variable ports. Furthermore, only certain

interface types are allowed for the ports.

Rule 6 *An application layer shell has only interface ports and no variable ports. It may contain a set of channel adapter instances such as application and presentation layer. The interface types allowed are as follows:*

`i_sender` *(un-typed)*

`i_receiver` *(un-typed)*

`i_tranceiver` *(un-typed)*

`i_send`

`i_receive`

memory interfaces

The operating system layer shell encapsulates the communication functionality related to operating system on the PE.

Rule 7 *The operating system layer shell contains exactly one behavior instance of the type of application layer shell. It may contain a set of channel adapter instances such session, transport, and network layer.*

Same as the application layer shell, the operating system layer shell only has interface ports and obviously, only untyped interface types are allowed.

Rule 8 *An operating system layer shell has only interface ports and no variable ports. The interface types allowed are as follows:*

`i_sender` *(un-typed)*

`i_receiver` *(un-typed)*

`i_tranceiver` *(un-typed)*

`i_send`

`i_receive`

memory interfaces

Rule 9 *An operating system layer shell needs to have interrupt handling tasks, which invoke the applications that are waiting on the corresponding interrupt.*

The rest of the shells, such as hardware abstraction layer, hardware layer, and bus-functional layer shell, are defined in PE database [9]. The refinement tools will insert the shells into the model and connected them properly.

3 Processing Elements

During the architecture exploration, the processing elements (PEs) are allocated. They perform the computation part of the specification. A PE can be an off-the-shelf software processor or a synthesizable FPGA or ASIC hardware unit. During the network exploration, upper layers of the protocol stack are inlined into the PE.

Rule 10 *A PE is represented by a SpecC behavior that must be a hierarchical behavior.*

A PE is represented by a SpecC behavior and should be instantiated in the top-level behavior. The definition of hierarchical behavior can be found in specification model report. The composition type of the PE behavior can be either `seq`, `par`, or `fsm` as defined in LRM.

Rule 11 *To declare a behavior as a PE, the behavior has to be annotated by* `_AR_MAPPED_TO`*.*

The PE allocation table contains the names of the allocated PEs. The PE is assigned to one of the allocated PEs by `_AR_MAPPED_TO`. This annotation is attached during architecture exploration.

Rule 12 *Each PE has only variable ports which are connected to bus wires.*

The port type of PE behaviors can be only signal type of bit vector, since the port needs to be connected to bus wires.

Rule 13 *A PE communicates to the outside through an implementation of a protocol stack.*

Inside each processing element, a protocol stack is implemented to interface with the outside wires. Some layers in the protocol stack are automatically generated by refinement tools, while others are taken out of bus database and are inserted into the PE.

3.1 Programmable PEs

For a programmable PE with flexible computation behavior, (i.e. no functionality provided in the PE behavior) but fixed, pre-defined interfaces and communication functionality, the bus-functional PE model has to provide a hierarchy with three layer shells: a top-level bus-functional layer shell, hardware layer shell and hardware abstraction layer (HAL) shell as shown in Figure 6. As described in database manual [9], a pin-accurate programmable PE model in the database can be thought of as additional communication layers that wrap around the PE behavioral model which comes from architecture model.

A PE is considered programmable in terms of its computation if the bus-functional PE model provides a hardware abstraction layer.

Rule 14 *The bus functional implementation of a programmable PE must have* `_PE_HAL_MODEL` *annotation.*

As part of the design process in SCE, communication synthesis will use the HAL of the bus-functional PE model as a template and modify it to implement computation on the PE on top of the services provided by the HAL model. In terms of services, the HAL defines the insertion point for implementing the PE's computation and it provides communication services for stream and memory I/O and interrupt handling as shown in Figure 7.

The HAL together with the outer hardware layer shell models the corresponding capabilities of the predefined PE implementation (e.g. number and type of external interfaces, amount and level of interrupts, etc.). The HAL shell describes the interface for accessing the PE's communication implementation from the programmable computation.

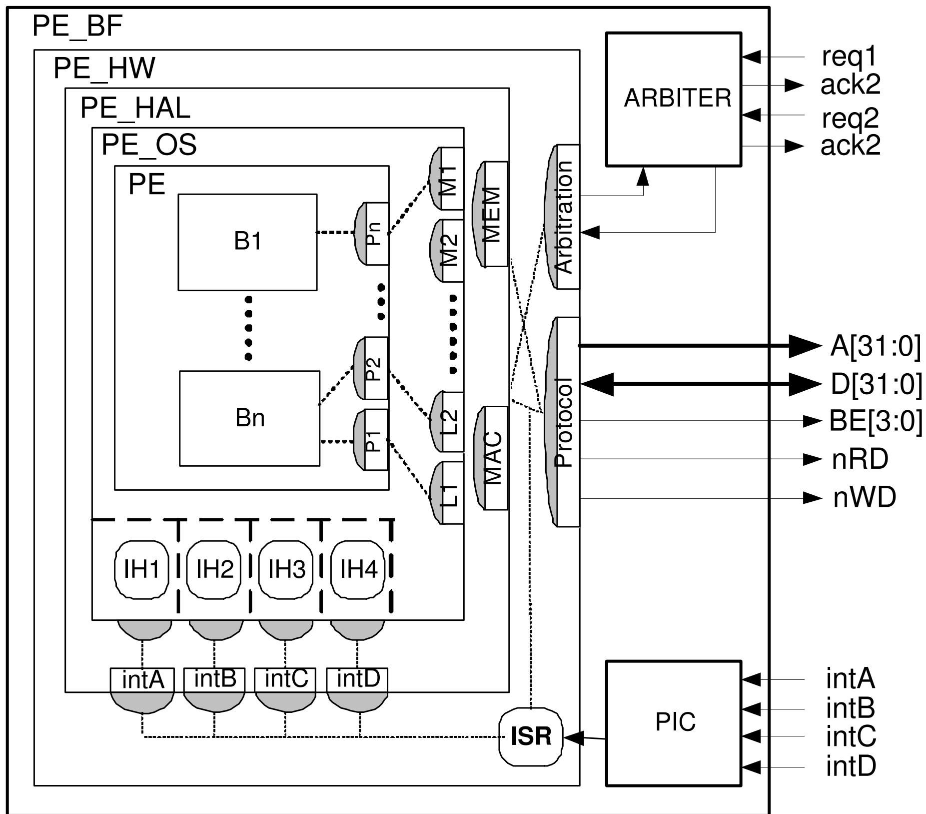


Figure 6: An example of programmable PE in communication model.

```
1  behavior Toshiba_TX49H2_HAL(
2      IToshibaGBusMaster mac,
3      inout bit[31:0] SR,
4      inout bit[31:0] CR) implements IToshiba_TX49H2_IntVectors
5  {
6      // MAC layer adapters
7      ToshibaGBusMasterLinkAccess link(mac);
8      ToshibaGBusMasterMemAccess mem(mac);
9
10     // operating system layer shell
11     Toshiba_TX49H2_OS cpu_os(link, mem);
12
13     // interrupt handlers
14     void int0handler(unsigned int num) {
15         cpu_os.DMAHandler();
16     }
17     void int1handler(unsigned int num) {
18         cpu_os.HWHandler();
19     }
20     void intHandler(void) {
21         unsigned int num;
22         if (CR[11]) {
23             num = mac.LoadDoubleword(0 + 1);
24             int1handler(num);
25         }
26         else if (CR[10]) {
27             num = mac.LoadDoubleword(0 + 0);
28             int0handler(num);
29         }
30     }
31
32     void main(void) {
33         cpu_os.main();
34     }
35 };
```

Figure 7: An example of a programmable PE in SpecC (HAL shell).

```
36 // hardware shell of processor
37 behavior Toshiba_TX49H2_HW(
38     out signal bit[35:0] GA,
39     out signal bit[63:0] GDOUT,
40     in signal bit[63:0] GDIN,
41     out signal bit[7:0] GBE,
42     out signal bit[0:0] GRD,
43     out signal bit[0:0] GWR,
44     in signal bit[0:0] GACK,
45     out signal bit[0:0] GREQ,
46     in signal bit[0:0] GGNT,
47     in signal bit[5:0] GINT)
48 {
49     bit[31:0] CR = 00000000000000000000000000000000b;
50     bit[31:0] SR = 00000000010000001111111100000101b;
51 
52     // arbitration
53     ToshibaGBusAccess access(GREQ, GGNT);
54 
55     // master interface (protocol and MAC layer)
56     ToshibaGBusMaster master(GA, GDOUT, GDIN, GBE, GRD, GWR, GACK);
57     ToshibaGBusMAC mac(master, access);
58 
59     // HAL shell
60     Toshiba_TX49H2_HAL hal(mac, SR, CR);
61 
62     // interrupt handler
63     Toshiba_TX49H2_Int intr(GINT, CR, SR, SR, SR, hal);
64 
65     void main(void) {
66         try {
67             hal.main();
68         }
69         interrupt(GINT, GHAVEIT) {
70             intr.main();
71         }
72     }
73 };
```

Figure 8: An example of a programmable PE in SpecC (HW layer shell).

```
74 // bus-functional layer shell of processor
75 behavior Toshiba_TX49H2_BF(
76     inout signal bit[35:0] GA,
77     inout signal bit[63:0] GDOUT,
78     inout signal bit[63:0] GDIN,
79     inout signal bit[7:0] GBE,
80     inout signal bit[0:0] GRD,
81     inout signal bit[0:0] GWR,
82     inout signal bit[0:0] GACK,
83     in signal bit[0:0] GREQ,
84     out signal bit[0:0] GGNT,
85     in signal bit[0:0] GSREQ,
86     out signal bit[0:0] GSGNT,
87     in signal bit[1:0] GINT0,
88     in signal bit[1:0] GINT1,
89     in signal bit[1:0] GINT2,
90     in signal bit[1:0] GINT3)
91 {
92     signal bit[3:0] GINT = 1111b;
93     signal bit[0:0] GNT = 1b;
94     signal bit[0:0] REQ = 1b;
95
96     // arbiter
97     Toshiba_TX49H2_Arbiter arbiter(
98         REQ, GNT, GREQ, GGNT, GSREQ, GSGNT);
99
100     // interrupt controller
101     Toshiba_TX49H2_IC ic(
102         GA, GDOUT, GDIN, GBE, GRD, GWR, GACK, GINT,
103         GINT0, GINT1, GINT2, GINT3);
104
105     // hardware shell for processor
106     Toshiba_TX49H2_HW hw(
107         GA, GDOUT, GDIN, GBE, GRD, GWR, GACK, REQ, GNT, GINT);
108
109     void main(void) {
110         par {
111             arbiter.main();
112             ic.main();
113             hw.main();
114         }
115     }
116 };
```

Figure 9: An example of a programmable PE in SpecC (BF layer shell).

Rule 15 *The bus-functional layer shell of the programmable PE can contain an interrupt controller and an arbiter.*

The bus-functional layer shell describes the PE pin interface to the outside and contains an interrupt controller for interrupt handling and an arbiter for arbitrating concurrent bus accesses to a bus. Figure 9 shows the bus-functional layer shell of the programmable PE, which contains an arbiter and an interrupt controller.

The programmable PE is associated with interrupts for synchronization with other components. The interrupt controller usually is a part of the programmable PE. An arbiter is necessary to resolve the multiple bus accesses on a bus. Usually, arbiters are not a part of the programmable PE, but a part of bus implementation, therefore they can be instantiated at the top-level behavior. In case the interrupt controller and arbiter are implemented outside of the programmable PE, the bus-functional layer shell will be collapsed with the hardware layer shell.

Rule 16 *The hardware layer shell between HAL shell and BF layer shell of a programmable PE can implement an interface for interrupt handling if it has interrupt pins.*

The interrupt handling is accomplished in HAL layer of the programmable PE. The HAL layer needs to provide interrupt service routines. The methods in the implementation interface of the PE will be called by the synchronization channels as shown in Figure 8.

Rule 17 *The hardware layer shell of the programmable PE has to contain an implementation of a protocol layer for each connected bus.*

Rule 18 *The hardware layer shell must have one instance of the HAL shell of the programmable PE and one interrupt handling behavior.*

As shown in Figure 8, the hardware layer shell contains the HAL shell of the programmable PE and interrupt handling behavior.

Rule 19 *The BF shell of the programmable PE must contain one instance of a hardware layer shell.*

As shown in Figure 9, the hardware layer shell is instantiated in BF layer shell of the programmable PE.

3.2 Hardware PEs

Programmable PEs are general purpose processors. Hardware PEs, on the other hand, are application specific hardware devices that need to be synthesized. Figure 10 shows a typical hardware PE in the communication model. The protocol stack implementation is instantiated and connected to others at the interface of the hardware PE. Hardware PEs can be synthesized by backend tools.

3.2.1 Hardware PE With a Local Memory

A hardware component may contain a local memory, which other components can access by I/O operations. The local memory model needs to provide read/write methods so that its contents are accessed by the PE itself and other PEs. Thus local memory is modeled as a behavior with interfaces as shown in Figure 11

Rule 20 *A hardware PE with a local memory must have a memory sub-behavior running concurrently with the rest of sub-behaviors.*

The local memory sub-behavior must run concurrently with other sub-behaviors in the HW PE, because it needs to be accessed by the PEs without the help of synchronization.

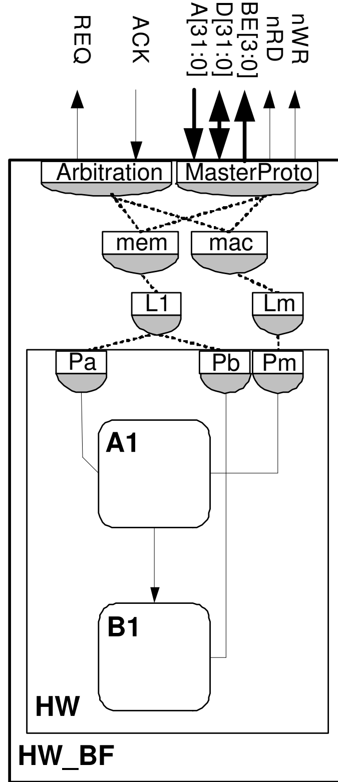


Figure 10: An example of a hardware PE in the communication model.

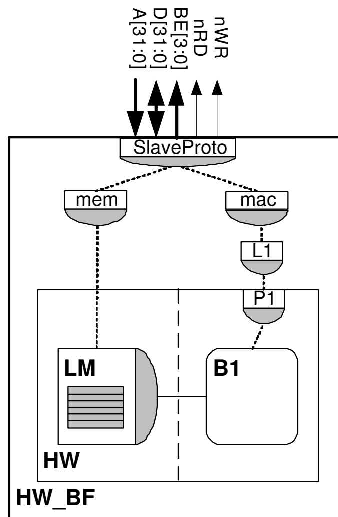


Figure 11: An example of hardware PE with local memory in communication model.

Rule 21 *The local memory sub-behavior in the HW PE must implement the read/write interfaces, which can be used by the rest of sub-behaviors in the HW PE and can be accessed by other PEs.*

The memory behavior inside the hardware PE provides read/write interfaces, so that the PE can access the local memory by interface method calls.

Figure 12 and Figure 13 shows SpecC code of a hardware PE with a local memory.

4 Memories

Memory is represented in SpecC by behaviors and should be instantiated in the top-level behavior. The bus-functional implementation of the memory behavior is stored in the PE database. Figure 14 shows an example of a shared memory in the communication model.

The SCE tools require that memory behaviors follow the rules outlined below.

Rule 22 *A memory is represented by a SpecC behavior, which must be a leaf behavior.*

Rule 23 *A memory can have only signal type of bit vector ports.*

The port type of memories has to be of signal type, because the port needs to be connected to the bus wires of the memory interface bus.

Rule 24 *The memory behavior contains one array variable that the variables of the architecture model are mapped onto.*

The size of memory is the same as the size of the array variable. The memory behavior provides methods to read/write the array variable.

Rule 25 *The memory behavior must contain an instance of a MAC layer and a protocol layer implementation of the associated memory interface protocol.*

The memory behavior instantiates the MAC layer implementation of the slave memory interface protocol from the bus database.

Rule 26 *The memory behavior must be a slave on the bus. Thus, the memory behavior invokes the `serve` method in its `main` method.*

The `serve` method is defined in the bus database and provides accesses to the array variable in the memory behavior. The `serve` method takes the base address of the memory, the array variable, and the size of the memory as arguments.

Figure 15 show the SpecC code of the shard memory.

5 Communication Elements

Communication elements such as transducers and bridges are represented by SpecC behaviors. They should be instantiated in the top-level behavior. Mapping a behavior to a CE is done by the annotation `_CR_MAPPED_TO`. The CE allocation table contains the names of all allocated CEs.

Rule 27 *A CE is represented by a SpecC behavior, which can be hierarchical behavior.*

Rule 28 *A CE has only signal type of bit vector ports and no interface ports.*

```
 1 behavior Local_Mem_TLM(
 2     IToshibaGBusSlaveMemAccess shm) implements I_AR_HW_Standard
 3 {
 4     char mem[MEM_SIZE]; // array or struct type variable
 5
 6     // memory adapter
 7     long int read_v1(void) {
 8         long int *ptr;
 9         ptr = (long int*) mem+OFS_v1;
10         return *ptr;
11     }
12     void write_v1(long int data) {
13         long int *ptr;
14         ptr = (long int*) mem+OFS_v1;
15         *ptr = data;
16     }
17
18     void main(void) {
19         while (true) {
20             shm.serve(ADDR_MEM, mem, MEM_SIZE);
21         }
22     }
23 };
24 behavior HW_Standard(
25     i_tranceiver L1,
26     IToshibaGBusSlaveMemAccess shm)
27 {
28     // link layer adapter
29     c_hw_cpu_double_handshake c1(L1);
30
31     // local memory behavior
32     Local_Mem_TLM LM(shm);
33
34     // original hardware behavior
35     B2 b2(c1, LM);
36
37     void main(void) {
38         par {
39             b2.main();
40             LM.main();
41         }
42     }
43 };
```

Figure 12: An example of a hardware PE with a local memory in SpecC (application layer shell).

```
44 behavior HW_Standard_BF(
45     in signal bit[35:0] GA_CPUBus,
46     in signal bit[63:0] GDOUT_CPUBus,
47     out signal bit[63:0] GDIN_CPUBus,
48     in signal bit[7:0] GBE_CPUBus,
49     in signal bit[0:0] GRD_CPUBus,
50     in signal bit[0:0] GWR_CPUBus,
51     out signal bit[0:0] GACK_CPUBus,
52     out signal bit[1:0] Intr1_CPUBus)
53 {
54     // interrupt generator
55     ToshibaGBusIntGenerate IntrGen(Intr1_CPUBus);
56 
57     // CPUBus protocol layer adapter
58     ToshibaGBusSlave slave(
59         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus, GRD_CPUBus,
60         GWR_CPUBus, GACK_CPUBus);
61 
62     // CPUBus MAC layer adapter for general data access
63     ToshibaGBusSlaveLinkAccess access(slave);
64 
65     // CPUBus link layer adapter for general data access
66     ToshibaGBusSlaveLink L1(access, intrA, ADDR_HW);
67 
68     // CPUBus MAC layer adapter for memory access
69     ToshibaGBusSlaveMemAccess shm(slave);
70 
71     HW_Standard HW(L1, shm);
72 
73     void main(void) {
74         HW.main();
75     }
76 };
```

Figure 13: An example of a hardware PE with a local memory in SpecC (BF layer shell).

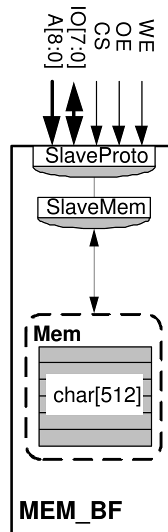


Figure 14: An example of a shared memory in the communication model.

```
1  behavior SRAM_BF(
2      in signal bit[18:0] A,
3      inout signal bit[7:0] IO,
4      in signal bit[0:0] CS,
5      in signal bit[0:0] OE,
6      in signal bit[0:0] WE)
7  {
8      char mem[MEM_SIZE];
9
10     // SRAMBus protocol layer adapter
11     SlaveKM684002A protocol(A, IO, CS, OE, WE);
12
13     // SRAMBus MAC layer adapter
14     KM684002AMemServe shm(protocol);
15
16     void main(void) {
17         while (true) {
18             shm.serve(ADDR_MEM, mem, MEM_SIZE);
19         }
20     }
21 };
```

Figure 15: An example of a shared memory in SpecC.

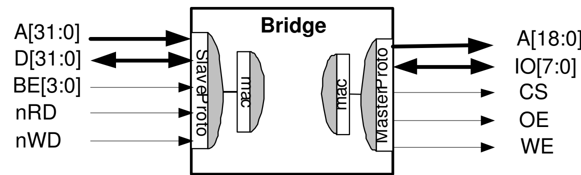


Figure 16: An example of bridge in communication model.

The signal type ports of a CE are connected to the wires of the bus.

Rule 29 *A CE must have* `_CE_BF_BUS` *annotation.*

`_CE_BF_BUS` contains the two interface bus protocols of the CE.

CE can be a bridge, a transducer, an arbiter and an interrupt controller.

5.1 Bridges

During the communication synthesis, the bridge in the network model is replaced with new bridge taken out of CE database.

Rule 30 *Bridges need to implement the MAC layer and protocol layer of the protocol stack.*

Figure 16 shows an example of bridge and the corresponding SpecC code is shown in Figure 17 and Figure 18.

5.2 Transducers

During the communication synthesis, transducers are treated like PEs. The protocol stack at the interface of the transducers needs to be implemented. In the communication model, transducers implement the network layer, link layer, MAC layer and protocol layer of the protocol stack, while PEs implement all layers of the protocol stack.

Rule 31 *Transducers need to implement the network, link, MAC layer and protocol layer of the protocol stack.*

Rule 32 *Transducers need to implement a synchronization scheme.*

As part of the link layer implementation, the synchronization by interrupt is necessary in the transducer.

Figure 19 shows an example of a transducer and the corresponding SpecC code is shown in Figure 20 and Figure 21.

5.3 Arbiter

If multiple masters are connected to a bus, the bus has to supply an arbitration protocol that is used to regulate accesses to the shared bus wires. In a centralized arbitration scheme, the master side of the arbitration protocol instantiated in each master communicates with the slave side of the arbitration protocol instantiated in an additional arbiter component attached to the bus. In a distributed arbitration scheme, there is no slave side of the arbitration protocol and the master sides of the protocol in each master regulate accesses among themselves.

```
1  behavior Bridge_BF(
2      // cpu interface
3      in signal bit[35:0] GA_CPUBus,
4      in signal bit[63:0] GDOUT_CPUBus,
5      out signal bit[63:0] GDIN_CPUBus,
6      in signal bit[7:0] GBE_CPUBus,
7      in signal bit[0:0] GRD_CPUBus,
8      in signal bit[0:0] GWR_CPUBus,
9      out signal bit[0:0] GACK_CPUBus,
10     // slave interface
11     inout signal bit[35:0] GA_SlaveBus,
12     inout signal bit[63:0] GDOUT_SlaveBus,
13     inout signal bit[63:0] GDIN_SlaveBus,
14     inout signal bit[7:0] GBE_SlaveBus,
15     inout signal bit[0:0] GRD_SlaveBus,
16     inout signal bit[0:0] GWR_SlaveBus,
17     inout signal bit[0:0] GACK_SlaveBus,
18     inout signal bit[0:0] GREQ_SlaveBus,
19     inout signal bit[0:0] GGNT_SlaveBus)
20 {
21     // CPUBus slave interface (protocol and MAC layer adapter)
22     ToshibaGBusSlave slave(GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus,
23         GBE_CPUBus, GRD_CPUBus, GWR_CPUBus, GACK_CPUBus);
24     ToshibaGBusAccess access(GREQ_SlaveBus, GGNT_SlaveBus);
25     ToshibaGBusMAC cpuBus(slaveBus, slaveAccess);
26
27     // SlaveBus master interface (protocol and MAC layer adapter)
28     ToshibaGBusMaster master(GA_SlaveBus, GDOUT_SlaveBus, GDIN_SlaveBus,
29         GBE_SlaveBus, GRD_SlaveBus, GWR_SlaveBus, GACK_SlaveBus);
30     ToshibaGBusArbiter slaveAccess(GREQ_SlaveBus, GGNT_SlaveBus);
31     ToshibaGBusMAC mac(master, slaveAccess);
```

Figure 17: An example of bridge in SpecC (I).

```
33     void main(void) {
34         bit[35:0] addr;
35         bit[63:0] d;
36         unsigned bit[10] t;
37         while (true) {
38             addr = 0x00000000;
39             t = cpuBus.Listen(&addr, 0x00000000);
40             if (((unsigned int)addr) == ADDR_HW) {
41                 switch((unsigned int)(t[1:0])) {
42                   case 1:    // cpu read
43                     d = mac.LoadCycle(addr, t[9:2]);
44                     cpuBus.WriteCycle(d);
45                     break;
46                   case 2:    // cpu write
47                     d = cpuBus.ReadCycle();
48                     mac.StoreCycle(addr, d, t[9:2]);
49                     break;
50                 }
51             }
52         }
53     }
54 };
```

Figure 18: An example of bridge in SpecC (II).

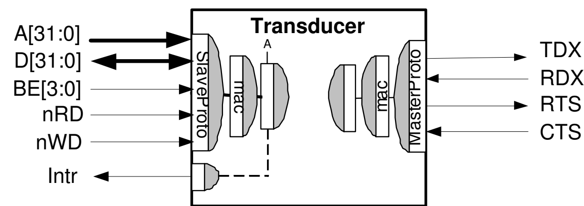


Figure 19: An example of a transducer in the communication model.

```
1 // Network model for transducer
2 behavior Tx_NET(
3     i_sender link_GBus,
4     i_receiver link_SIO)
5 {
6     void main(void) {
7         char buf[128];  // buffer
8         unsigned long int len;
9         while (true) {
10            link_SIO.receive(&len, sizeof(len));
11            link_SIO.receive(buf, len);
12            link_GBus.send(buf, len);
13        }
14    }
15 };
```

Figure 20: An example of transducer in SpecC (network model).

```
17 // BF model for transducer
18 behavior Tx_BF(
19     // CPUBus slave interface
20     in signal bit[35:0] GA_CPUBus,
21     in signal bit[63:0] GDOUT_CPUBus,
22     out signal bit[63:0] GDIN_CPUBus,
23     in signal bit[7:0] GBE_CPUBus,
24     in signal bit[0:0] GRD_CPUBus,
25     in signal bit[0:0] GWR_CPUBus,
26     out signal bit[0:0] GACK_CPUBus,
27     out signal bit[1:0] IntrA_CPUBus,
28     // RS232 master interface
29     in signal bit[0:0] TXD_RS232,
30     out signal bit[0:0] RXD_RS232,
31     in signal bit[0:0] RTS_RS232,
32     out signal bit[0:0] CTS_RS232)
33 {
34     // CPUBus slave interface (protocol, MAC and link layer)
35     ToshibaGBusIntGenerate IntrGen(IntrA_CPUBus);
36     ToshibaGBusSlave cpuBus(
37         GA_CPUBus, GDOUT_CPUBus, GDIN_CPUBus, GBE_CPUBus,
38         GRD_CPUBus, GWR_CPUBus, GACK_CPUBus
39     );
40     ToshibaGBusSlaveLinkAccess cpuAccess(cpuBus);
41     ToshibaGBusSlaveLink cpuLink(cpuAccess, intrA, ADDR_SIO);
42
43     // RS232 master interface (protocol, MAC and link layer)
44     RS232_DCE RS232(TXD_RS232, RXD_RS232, RTS_RS232, CTS_RS232);
45     RS232BusLinkAccess sioAccess(sioBus);
46     RS232BusMasterLink sioLink(sioAccess);
47
48     // network model of transducer
49     Tx_NET Tx(cpuAccess, sioAccess);
50
51     void main(void) {
52         Tx.main();
53     }
54 };
```

Figure 21: An example of transducer in SpecC (BF layer shell).

In order to support complex arbitration capabilities with different priorities and arbitration schemes, the CE database needs to include arbiters Typically, the arbiter provides a set of request/acknowledge lines at the pins of the top-level bus-functional layer. The MAC layer adapter of the master components will use the arbitration protocol to get bus accesses.

Figure 22 shows the arbiter example in SpecC.

```
1  behavior Toshiba_TX49H2_Arbiter(
2      in signal bit[0:0] REQ,
3      out signal bit[0:0] GNT,
4      in signal bit[0:0] GREQ,
5      out signal bit[0:0] GGNT,
6      in signal bit[0:0] GSREQ,
7      out signal bit[0:0] GSGNT)
8  {
9      ToshibaGBusArbiter core(REQ, GNT, REL, HAVEIT);
10     ToshibaGBusArbiter sm(GSREQ, GSGNT, GREL, GHAVEIT);
11     ToshibaGBusArbiter m(GREQ, GGNT, GREL, GHAVEIT);
12
13     void main(void) {
14         while(true) {
15             if (!GSREQ) {
16                 sm.grant();
17                 wait(GHPREQ falling , GHPSREQ falling , GSREQ rising);
18                 sm.release();
19             }
20             else if (!GREQ) {
21                 m.grant();
22                 wait(REQ falling , GSREQ falling , GHPREQ falling ,
23                 GHPSREQ falling , GREQ rising);
24                 m.release();
25             }
26             else {
27                 core.grant();
28                 wait(GREQ falling , GSREQ falling , GHPREQ falling ,
29                 GHPSREQ falling);
30                 core.release();
31             }
32         }
33     }
34 };
```

Figure 22: An example of arbiter.

5.4 Interrupt Controller

In order to support more complex interrupt capabilities with more than one source of interrupts, different priorities and masking the CE database needs to include interrupt controllers as part of the bus-functional PE models. Interrupt controllers sit in front of the basic PE core model and are modeled by adding another layer to the bus-functional PE model between the processor core and the outer bus-functional layer shell.

Typically, the interrupt controller provides a set of interrupt lines at the pins of the top-level bus-functional layer while internally communicating with the core via the PE bus and the core's interrupt condition input. The core then interrupts normal computation and executes the appropriate handler depending on the inputs received from the interrupt controller. Overall, the combination of layers has to simulate the proper interrupt

behavior while maintaining the relationship between interrupt pins at the bus-functional layer and interrupt handlers in the HAL required by the database format for SCE.

Figure 23 shows the interrupt controller example in SpecC.

```
1  behavior Toshiba_TX49H2_IC(
2      in signal bit[35:0] GA,
3      in signal bit[63:0] GDOUT,
4      out signal bit[63:0] GDIN,
5      in signal bit[7:0] GBE,
6      in signal bit[0:0] GRD,
7      in signal bit[0:0] GWR,
8      out signal bit[0:0] GACK,
9      out signal bit[5:0] GINT,
10     in signal bit[1:0] GINT0,
11     in signal bit[1:0] GINT1,
12     in signal bit[1:0] GINT2,
13     in signal bit[1:0] GINT3)
14 {
15     signal bit[1:0] flag0;
16     signal bit[1:0] flag1;
17     signal bit[1:0] flag2;
18     signal bit[1:0] flag3;
19
20     ToshibaGBusSlave bus(GA, GDOUT, GDIN, GBE, GRD, GWR, GACK);
21     Toshiba_TX49H2_IC_Detect int0detect(GINT0, flag0);
22     Toshiba_TX49H2_IC_Detect int1detect(GINT1, flag1);
23     Toshiba_TX49H2_IC_Detect int2detect(GINT2, flag2);
24     Toshiba_TX49H2_IC_Detect int3detect(GINT3, flag3);
25     Toshiba_TX49H2_IC_Control control0(bus, GINT, flag0, (0u));
26     Toshiba_TX49H2_IC_Control control1(bus, GINT, flag1, (1u));
27     Toshiba_TX49H2_IC_Control control2(bus, GINT, flag2, (2u));
28     Toshiba_TX49H2_IC_Control control3(bus, GINT, flag3, (3u));
29
30     void main(void) {
31         par {
32             control0.main();
33             control1.main();
34             control2.main();
35             control3.main();
36             int0detect.main();
37             int1detect.main();
38             int2detect.main();
39             int3detect.main();
40         }
41     }
42 };
```

Figure 23: An example of interrupt controller.

6 Bus Wires

In the pin-accurate communication model, bus wires are represented by signal type of bit vector variables and should be declared in top-level behavior.

Rule 33 *Bus wires on a bus are taken out of the bus database and need to be connected to the components on the bus.*

The type of bus wires is defined in the bus channel of the bus database. The wire name will be the concatenation of the bus name and the wire name out of the bus database. If the wire has a initial value on the bus database, the wire will be initialized to the initial value in its declaration.

Rule 34 *Bus wires on a bus are identified by the naming convention like* `pin_name_bus_name`.

For example, the address line `A` on the bus `Bus0` is defined to `A_Bus0` in the model.

7 Example

Figure 24 shows an example of communication model. The logical link channels in the network model are inlined and connected to next higher layer (presentation layer). MAC and protocol layer channel adapters are taken out of media protocol library and inserted into the bus functional model of the corresponding components and connected to the corresponding inlined logical link adapters.

The shaded parts (the shells *SW_HW* and *SW_BF* for *SW* including *PIC* and *Arbiter* models, the protocol adapter channels, the *MemCtrl* model and the bus-functional model *MEM_BF* for the memory) in the figure indicate that they are taken out of the database instead of being generated by the tool.

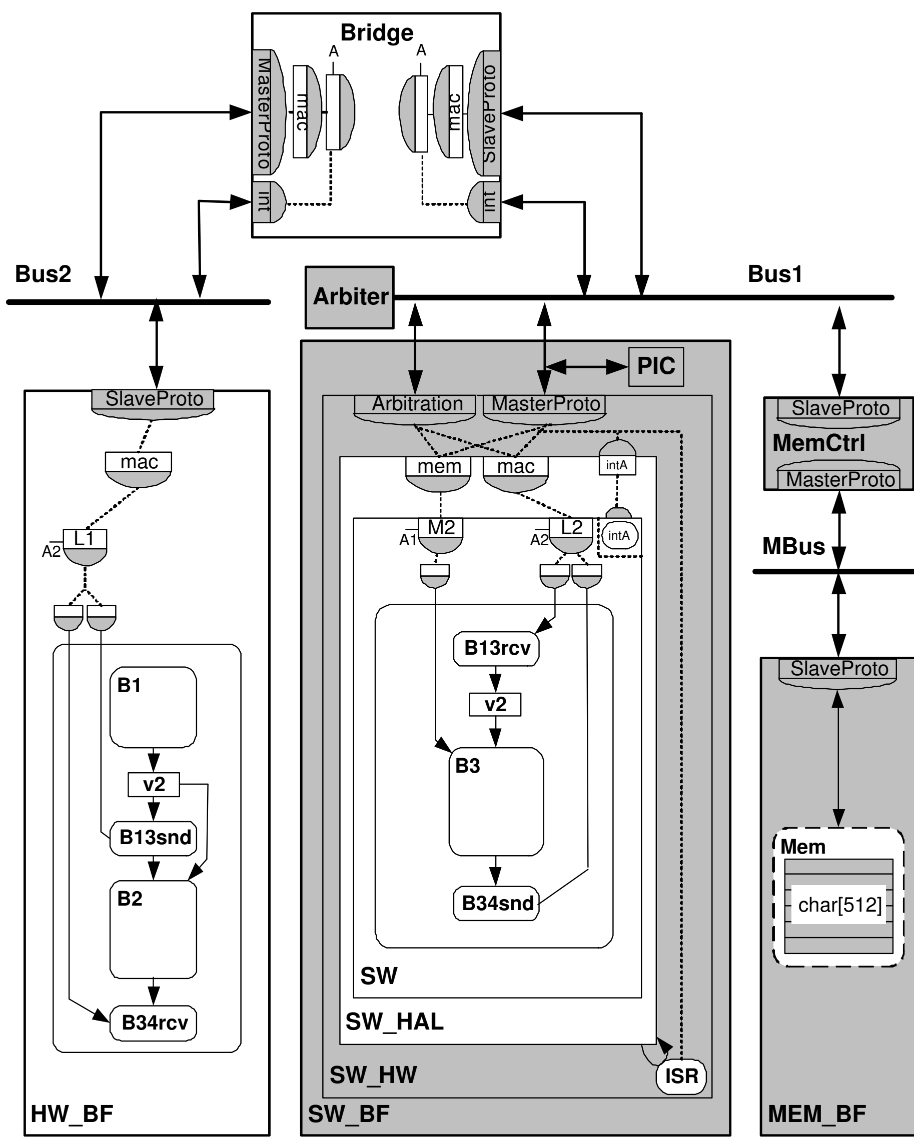


Figure 24: Communication model example.

References

[1] R. Dömer, A. Gerstlauer and D. D. Gajski. *SpecC Language Reference Manual, Version 2.0*, SpecC Technology Open Consortium (STOC), Japan, December 2002.

[2] SpecC Technology Open Consortium. http://www.specc.org.

[3] SpecC Compiler V2.2.0, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

[4] L. Cai, A. Gerstlauer, S. Abdi, J. Peng, D. Shin, H. Yu, R. Dömer and D. D. Gajski. *System-on-Chip Environment (SCE Version 2.2.0 Beta): Manual*, Technical Report CECS-TR-03-45, Center for Embedded Computer Systems, University of California, Irvine, December 2003.

[5] A. Gerstlauer, K. Ramineni, R. Dömer and D. D. Gajski. *System-on-Chip Specification Style Guide*, Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[6] J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Architecture Modeling Style Guide*, Technical Report CECS-TR-04-22, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[7] D. Shin, J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Network Modeling Style Guide*, Technical Report CECS-TR-04-23, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[8] D. Shin, L. Cai, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Transaction-Level Modeling Style Guide*, Technical Report CECS-TR-04-24, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[9] A. Gerstlauer, L. Cai, D. Shin, R. Dömer and D. D. Gajski. *System-on-Chip Component Models*, Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, August 2003.