Making Alive Register Transfer Level and Transaction Level Modeling in Ada

Negin Mahani
Electrical and Computer Engineering Department, Faculty of Engineering, Campus #2
University of Tehran, 14399 Tehran, IRAN
negin@cad.ece.ut.ac.ir

Abstract

Over the past 50 years, design of hardware has evolved from transistor level to register transfer level (RTL), and now to transaction level. Transaction Level Modeling (TLM) enhances simulation performance of today's complex digital systems and also provides the ability of early design space exploration. TLM divides a system into computation parts, i.e. processing elements, and communication parts, i.e. channels and sockets. The inherent concurrency of Ada along with its object oriented features gives it potentials for being used as a TLM language.

In this paper, we use a specialized form of Ada as a system description language, like the way SystemC is used for description of systems. We refer to our form of Ada as SystemAda and we use a public Ada compiler (Gnat) to evaluate system descriptions written in Ada. SystemAda is meant for modeling system behavior and structure at the transaction level and we consider possible approaches for extending Ada to meet these requirements. This paper discusses the specification of our proposed SystemAda, its hardware description style, its RTL link, and description of a TLM 1.0 channel using SystemAda.

1. Introduction

To cope with the complexity of electronic systems, high level description languages have evolved for describing electronic hardware at system level. Today, Transaction Level Modeling has established itself as a common way of describing digital systems at the system level for design and architecture exploration. TLM allows designers to focus on the functionality of the design and not to be concerned with low level (RTL) details. At this level of abstraction, a hardware system is divided into processing elements and communication channels. At the top level, this system consist of a structural interconnections of processing elements, where interconnections become channels and sockets instead of wires at the gate level, or buses at RTL [1].

For the design of such systems, hardware description languages that are different than those used in today's register transfer languages should be used. The focus of such languages should be on processing elements connected by channels, instead of registers and logic units connected by buses as in register transfer level languages. Furthermore, languages for system level descriptions must be capable to easily interface with software languages, to enable designers to design complex hardware/software systems in one environment [1].

Our study of Ada language features, compilers, and support for concurrency indicates that this language is a good candidate for describing hardware at RT and transaction levels. Inspired by an existing RTL package for Ada, we have developed an RTL link for Ada that provides system level hardware designers with sufficient RT level description capabilities. In addition, we have developed a package containing a basic transaction level channel to provide Ada system level designers with hardware description features found in today's transaction level hardware description languages. The combination of our added RTL and transaction level features makes our SystemAda that is described in this paper.

In addition to providing constructs for covering hardware at certain level, a hardware description language at any level has to provide a minimum set of constructs for describing hardware at its immediate next lower level of abstraction. Therefore, we have to cover some preliminary RT level constructs for creating a transaction level hardware language from Ada. This work focuses on TLM while providing a sufficient link to RTL.

In this paper, we introduce SystemAda as a system description language for hardware systems. The work is partitioned based on two distinct concepts: RTL and TLM. In the RTL concept, we explain different options for linking Ada to RTL and expand one of them. In the TLM concept, we describe *tlm_fifo* as the most basic TLM channel which will be used for describing other channels. The rest of this paper is organized as follows. Section 2 briefly introduces Transaction Level Modeling and TLM standards released by OSCI.

Section 3 contains an overview of Ada as a Hardware Description Language (HDL). In Section 4, major requirements for SystemAda are explained. Section 4.1 introduces existing methods for linking Ada to RTL and adopts one such method and adds several new features to it. Section 4.2 contains an overview of TLM channels and their implementation in Ada. Section 5 describes *tlm_fifo* application in a *master-slave* architecture using Ada *tasks*. In section 6 we have implemented a network on chip system using our Ada packages. Finally, in Section 7 we present the conclusion of this work.

2. Transaction Level Modeling

Recent advances in semiconductor technology enable the designers to integrate hundreds of embedded processors on a single chip which is called System on Chip (SoC). This fast-paced increasing of complexity of digital systems, on the other hand, has introduced new challenges to the design community. An important challenge is to handle the complexity of designing such systems with increased productivity and decreased time-to-market. Electronic System Level (ESL) design methodologies solve this problem by starting the design from higher abstraction levels than RTL. Transaction Level Modeling (TLM) has been proposed as the key step toward ESL methodology. In TLM point of view, a system is divided into two parts: computation parts and communication parts. The focus of the TLM is on communication and this is performed through passing high level data structures called transactions between computation parts.

SystemC that is a library on C++ has been used widely for design at transaction level. Open SystemC Initiative (OSCI) which works on the definition and standardization of SystemC as a system-level language has released two versions of the TLM standard. In TLM 1.0, channels are the basic communication elements, while in TLM 2.0 the concept of the sockets has been introduced as the communication infrastructure.

In this work, we will examine the possibility of using another language, i.e. Ada, as a TLM language. In general, having the link to RTL and the ability to describe TLM channels are two essential requirements of a TLM language. The inherent concurrency of Ada along with its object oriented features gives it potentials for satisfying these requirements. We will discuss these issues in later sections.

3. Ada as an HDL

Most hardware designers use an HDL such as VHDL or Verilog for their RT level descriptions, C or C++ for software parts of their designs, and SystemC and its TLM derivation for fast simulation of system level designs. With SigAda's project [2], RTL design, software parts of a design, and software for design test and verification can all be done in the same language and environment.

The idea of using Ada as an HDL goes back to 1980's. At that time, Ada compilation was slow and hardware design abstraction was at gate level, making Ada not the best alternative for hardware description. In 1981, Ada was used as a base language for the development of VHDL [3]. In 1995, a new version of Ada called Ada95 was defined with object oriented features that could be used for high abstraction level design such as TLM [4]. Considering Ada as the VHDL base language, a popular HDL for complex hardware description at RTL, it has the proper root to be used for system level description as a TLM language.

Because hardware systems comprise activities that are performed concurrently, an important requirement for a hardware description language is its support for concurrency. In addition to concurrency, an HDL for transaction level modeling should have some language features for object oriented modeling, genericity, and abstract communication.

Considering language features and structures of Ada, we can easily conclude that the TLM standard is implementable in this language. In addition to its capabilities for transaction level modeling, Ada has links with the lower level, i.e. RTL. The VHDL language that has been derived from Ada is similar to Ada in syntax. VHDL hardware modules that are described with entity-architecture parts are equivalent to the specification-body structures of the Ada language. Such observations led us to conclude that Ada, as it is presently defined, has potentials for description of hardware.

From the linguistics point of view, Ada compilers have the advantage of being strongly typed and being thorough in the compilation process such that most syntax and semantic errors are detected during compilation. Such features make Ada a language that is more feasible for design error detection than C/C++, their additional patches, or their SystemC derivations [5][6].

Furthermore, Ada2005 has services which provide the ability to trigger events at specific times, invoking threads, and facilitating process synchronization. Object oriented programming could be linked to real time activities using these capabilities [5][6].

4. Major requirements for SystemAda

In addition to supporting TLM abstraction level, system level languages like SystemC also provide a

strong link to RTL. In general, having the link to RTL and the ability to describe TLM channels are two essential requirements of a TLM language [17]. In this work, we consider the main concepts of the OSCI TLM 1.0 standard [7][8], and base our requirements on the main features supported by this standard.

Since communication is the main focus of the TLM and all TLM channels have been defined based on *fifo* channel (*tlm_fifo*), we have developed a *tlm_fifo* channel in this paper. We show how this channel is instantiated and used for interconnection of processing elements.

As mentioned above, a hardware description language needs a link to its most immediate lower level language; in case of our system level, the lower level is RTL. Therefore, we need to provide a link between our system level SystemAda and an existing RTL hardware description language.

4.1. Linking Ada to RTL

There are several options for linking Ada to RTL. The following sections describe the different options. Our RTL part of Ada is based on the method described in the last section. We have created an HDL package for Ada using this method.

4.1.1. Linking using Ada pragmas. An option for creating an HDL link for Ada is using the ability of Ada to communicate with a language which supports RTL. The most useful *pragma*s are *import, export,* and *convention* [9].

According to Ada95 language reference manual, the *import pragma* imports a subprogram from a foreign language into an Ada program. The *export pragma* exports an Ada subprogram to another language, and the *convention pragma* specifies that a certain type should use the conventions of a given foreign language. It is also used on subprograms if they are of the callback type [9][10][17].

Since Ada and C++ programs can interface by Ada *pragma*s (*import, export,* and *convention*), combining Ada and SystemC RTL core is possible.

4.1.2. Linking with VHDL. The second option for providing a link between Ada and RTL is using GHDL. GHDL is an open source VHDL compiler that can simulate Ada programs [11].

Command lines for this compiler provide analyzing VHDL design files, binding the design, and finally compiling the Ada program, binding it and linking it to the VHDL design. Figure 1 shows GHDL and GNAT compiler commands achieving this linking [17].

```
$ ghdl -a design.vhdl
$ ghdl --bind design
$ gnatmake my_prog -largs `ghdl -listlink
design`
```

Figure 1. GHDL and GNAT compiler commands

4.1.3. Linking by mapping VHDL to Ada. Another possibility for creating a link between Ada and an RT level HDL is mapping VHDL structures to Ada. The U.S. Air Force Research Lab has provided examples of how to describe VHDL constructs in Ada. Signals are unique objects in VHDL and their declarations and assignments are modeled in Ada. VHDL signal attributes have been modeled using an Ada record. This record is called a *signal description record* [12].

In general, the work described in [12] has mapped VHDL language defined types, signal declarations, and signal assignments to various Ada constructs. For example, for implementing VHDL signal assignment an *if* or a *case* statement is used [17].

4.1.4. Linking by an Ada HDL package. We have created an HDL package for providing a link between Ada and a hardware description language. The basic combinational components of an RT level description consist of word gates (array of gates), multiplexers, decoders, encoders, and arithmetic units. In addition, sequential components are registers and counters (Table 1).

Table 1. Digital logic basic elements

Component Type	Subtype
Elementary logic gates	AND, OR, NAND, NOR, Inverter, and their array versions
Decoder	-
Encoder	-
Multiplexer	-
Full adder	-
Basic latches	SR, D, Gated SR, Gated D
Triggered D flip-flop	-

SigAda's project in 1998 [2] proved that Ada83 could be used as a hardware description language. Since we are using Ada95 to describe hardware at TLM, we have used this version of Ada to develop a package called *UT_Ada_HDL* to provide RTL designers with the required primitives. For this purpose, we have implemented basic RTL elements mentioned above using Ada95 procedures. Based on this and the fact that Ada is an object oriented language, by taking advantage of its reusability facility, a hardware designer can use Ada procedures to develop more complex elements[16] [17].

UT_Ada_HDL package which is partially shown in Figure 2 contains the Ada script and package made

of all the digital logic basic elements shown in Table 1. The basic types and subtypes in this package are as follows:

- Subtype *input*: This type is *Boolean* and is the same as SigAda's input.
- Subtype *output*: This type is *Boolean* and is the same as SigAda's output.
- Type *bus*: This is a *Boolean* array of natural range.
- Type *state*: This is an enumeration type with high and low enumeration elements. This type indicates clock states high and low.
- Type *edge*: This type is a *Boolean* array of natural range. User can use this array to specify clock edges.
- Type *dimension2_bus*: This type is a two dimensional *Boolean* matrix of natural range. This type is introduced for more complex RTL packages.

```
package UT_Ada_HDL is

subtype input  is boolean;
subtype output is boolean;

type bus is array
(natural range <>)of boolean;

type dimension2_bus is array
(natural range<>, natural range<>) of
boolean;

type state is (high, low);

type edge is array
(natural range<>) of boolean;

type edge_not is array
(natural range<>) of boolean;
      …
procedure nand_array (x1: in out bus;
x2: in out bus; n: in out   integer;
xout: out bus);

procedure mux (en:in input;
mux_in: in input; data1: in input;
data2: in input; mux_out: out output);

procedure SR_latch (S: in out input;
R: in out input; Q_a: out output;
Q_b: out output);

procedure fulladder (input1: in input;
input2: in input; carry_in: in boolean;
carry_out: out output; sum: out output);
      …
end UT_Ada_HDL;
```

Figure 2. UT_Ada_HDL package

Included in this package are procedures *Invert, And_bit, or_bit, nand, nor*, and *mux.* These procedures use Ada's *Boolean* operations for faster simulation. Since word gates inputs use *bus* data type, direct use of *Boolean* operations is not allowed. Therefore, we have used type conversion between *bus* and *Boolean* data types. As an example, the implementation of a *nand* word gate as shown in Figure 3 uses *Boolean* type conversion for *x1* and *x2* input bus types. The invert procedure that also is shown in this figure returns complement of *xouttemp* as *nand* output.

```
procedure nand_array (x1: in out bus; x2: in
out bus; n: in out integer; xout: out bus)
is
xouttemp,xouttemp_not,x1temp,x2temp :
boolean;
begin
  for i in  reverse x1'range loop
    xouttemp:= boolean(xout(i));
    x1temp:= boolean(x1(i));
    x2temp:=boolean(x2(i));
    xouttemp:= x1temp and x2temp;
    invert(xouttemp,xouttemp_not);
    xout(i):= xouttemp_not;
  end loop;
end nand_array;
```

Figure 3. Nand_array procedure

A simple multiplexer with two data lines, one select line, and an enable control input is shown in Figure 4. As mentioned before, this *mux* is included in our *UT_Ada_HDL* package.

```
procedure mux (en:in input; mux_in: in input;
data1: in input; data2: in input; mux_out: out
output) is
begin
   if en then
      if mux_in then
         mux_out:= mux_in and data1;
      else
         mux_in_invert:= not mux_in;
         mux_out:= mux_in_invert and  data2;
      end if;
   else
      put("off");
   end if;
end mux;
```

Figure 4. Multiplexer procedure

```
procedure SR_latch (S: in out input; R: in
out input; Q_a: out output; Q_b: out output)
is
begin
    delay 0.004 ;
    Nor(S,Q_a, q_btemp);
    Nor(R,Q_b, q_atemp);
    Q_a:= q_atemp;
    Q_b:= q_btemp;
end SR_latch;
```

Figure 5. SR_Latch procedure

Figure 5 shows implementation of an SR latch. In this procedure, Ada *delay* construct has been used to imply the actual data passing delay in hardware.

4.2. Describing TLM channels using Ada

This section starts with the description of Ada *tasks* that are the main language constructs we have used for describing TLM channels. This will be followed by a subsection for describing the main TLM 1.0 channel, i.e. *tlm_fifo.*

4.2.1. Task overview. *Tasks* are the basic elements for implementing concurrency in Ada. *Task* units can communicate with each other, and each will be in progress for a specified time. The language semantics make it look like that *task*s are running on separate computer systems. *Task*s have this ability using the *entry* concept, which defines what information should be sent to a *task* when it is invoked, and what should be done. *Task*s can be sensitive to activation of one or more *entries*. *Task*s have a declaration part and a body part. A *task* body part describes the functionality of the *task* [13][14][15].

Each *task* can use Ada's *select* block and its *accept* statement. The *select* block, with the *accept* statements inside it, provides alternative entry points for messages into a *task*. The *accept* statements in the *select* block are separated by OR to allow the *task* to accept a call on each entry [13][14][15].

4.2.2. TLM_FIFO channel in Ada. Every TLM 1.0 channel can be implemented based on the *tlm_fifo* channel. This channel is *generic* in size and type. As a result, we define a *generic fifo* channel in Ada to be used later to define other TLM channels [16] [17].

```
with Ada.Text_IO;
    …
generic type fifo_element is(<>);
package fifo IS
  type fifo_node;
  type fifo_pointer is access fifo_node;
  type fifo_node is record
    …
   input : fifo_element;
   output: fifo_element;
    …
  procedure add_fifo;
  procedure rem_fifo;
    …
   task type fifo_task is
      entry add;
      entry remove;
      entry stop;
   end fifo_task;
   tlm_fifo : fifo_task;
end fifo;
```

Figure 6. Generic fifo package

The *generic* keyword in Ada has the functionality of the *template* in C++. Figure 6 shows the code of our *fifo* specification that can be used as a TLM channel. This *fifo* is a linked list that has no size limitation. Because of the *generic* element type, the type of its nodes can be of any types.

The *tlm_fifo* of Figure 6 represents a hardware component for communication between several running processes. This component that is used at transaction level must have concurrency as an essential requirement for modeling hardware systems. For this reason, we have used a *task type,* as the basic unit of concurrency in our *fifo*. The *tlm_fifo task* in this figure is an instance of the *task* type called *fifo-task*.

Our *fifo task* has the *add* and *remove* entries, whose names describe their functionalities. A *generic fifo* provides users with the opportunity to determine the type of its elements. The program shown in Figure 7 shows how we define an integer *fifo* [1].

```
with Ada.Text_IO;
use Ada.Text_IO;
with fifo;
package fifo_channel is new fifo(integer);
```

Figure 7. Generating an integer fifo

5. TLM master-slave architecture

We have modeled aTLM master-slave architecture in Ada using *task types* with two computational modules: *master* and *slave*. The block diagram of the TLM master-slave architecture is shown in Figure 8 [16] [17].

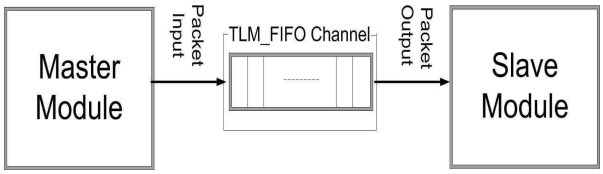

Figure 8. TLM master-slave architecture

```
with fifo_channel;
    …
procedure tlm_master_slave is
   task  slave is
    …
    fifo_channel.tlm_fifo.remove;
    …
   task  master is
    …
       fifo_channel.tlm_fifo.add;
    …
  task simulation is
    …
begin
   simulation.start;
   delay 1.0;
   simulation.stop;
end tlm_master_slave;
```

Figure 9. TLM master-slave procedure

Figure 9 shows a procedure that defines master and slave interconnections using a *tlm_fifo*. This procedure is composed of three *task*s, each containing

an infinite loop inside its body: *master, slave* and *simulation*. These *task*s have two entries *start* and *stop*. The *master task* enters integer numbers into the integer *fifo* by sending *fifo_channel.tlm_fifo.add* message to *tlm_fifo task*. On the other hand, the *slave task* removes integers from the integer *fifo* by sending *fifo_channel.tlm_fifo.remove* message to *tlm_fifo task*, and performs its operation on the obtained data.

In the main block of the program (Figure 9), *simulation.start* massage starts the *simulation task*. After a delay of 1.0, the *simulation.stop* message will stop the *simulation task*. The *simulation task* of Figure 10 by accepting the *start* message enters the loop and sends the *start* massage to *master* and *slave task*s to notify that the simulation has been started. By accepting the *stop* message, the simulation *task* exits the loop and stops the *master*, *slave*, and *tlm_fifo task*s.

```
task simulation is
      entry start;
      entry stop;
end simulation;
task body simulation is
begin
      accept start;
      loop
         select
         accept stop;
              exit;
         else
            master.start;
            slave.start;
         end select;

      end loop;
      master.stop;
      slave.stop;
    fifo_channel.tlm_fifo.stop;
end simulation;
```

Figure 10. Simulation task declaration and body

The process described above is an exact modeling of channels as expected in transaction level design. This description is more accordant to behavior of hardware than if the description were done with C++ and SystemC [1].

6. Implementation of a Network-on-Chip using SystemAda

In order to show the capabilities of SystemAda for modeling of digital systems at transaction level, we have implemented a Network-on-Chip (NoC) using SystemAda. The NoC architecture implemented in this paper is shown in Figure 11. This NoC has 7 switches and 6 processing elements. Switches are shown as squares, processors as circles, and channels between modules as straight lines. Each switch has a connected processing element except for switch4. Switches are connected to their adjacent switches by fifo channels which are named F1 to F8 respectively. In a similar way, each switch is connected to its corresponding processor by a fifo channel. Our NoC architecture uses a dynamic routing algorithm and works based on Store and Forward (SaF) packet switching method. The routing algorithm will be discussed later in this section. Processor1 and Prcoessor7 are the master processors of the NoC and input and output ports of the circuit are located at these processors respectively. Processor1 is in charge of reading data packets from an input file and sending them to Switch1. When Switch1 receives an incoming packet, sends it to appropriate output port that is decided by the routing algorithm. Upon arrival at destination, the packet is processed by corresponding processor and then, is sent toward Switch7. When Switch7 receives a packet, sends it to Processor7 which writes the packet data into some output file.

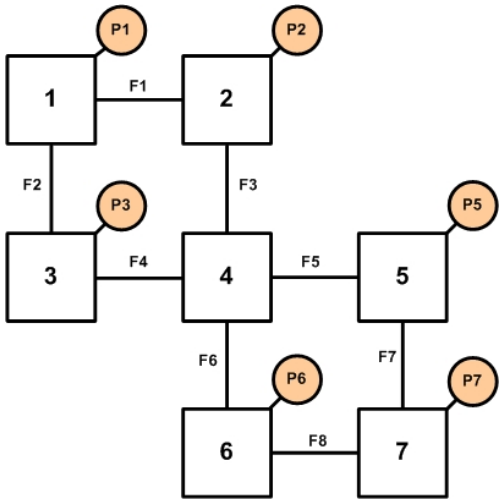

Figure 11. NoC architecture

Figure 12 shows the body of the procedure which implements the functionality of our NoC. A task is defined for each processor and switch in the NoC. Each task has two entries, start and stop, except for Switch4 and switch7. Because Switch4 and Switch7 receive packets from two neighboring switches, they have separate entries for each input port instead of a single start entry. Swicth4 has two entries named start2 and start3 and Switch7 has entries start5 and strart6. Simulation task controls the simulation of the NoC design and has two entries for start and stop the simulation.

```
with
fifo1,fifo2,fifo3,fifo4,fifo5,fifo6,fifo7,fifo
8,fifop1,fifop2,fifo3,fifo5,fifo6,fifop7;
procedure SystemAda_NoC is

    task processor1 is
        entry start;
```

```
        entry stop;
    end processor1;
    …
    task processor7 is
        entry start;
        entry stop;
    end processor7;

    task switch1 is
        entry start;
        entry stop;
    end switch1;
    …
    task switch7 is
        entry start;
        entry stop;
    end switch7;

    task simulation is
        entry start;
        entry stop;
    end simulation;
```

Figure 12. SystemAda_NoC procedure

Figure 13 shows the body of processor1 task. The packets that will be sent to the network are written in a file named input_file. At the start of the task, input_file is opened. Then, the task enters an infinite loop in which the start entry is checked first. If the end of input_file is not reached, a packet is read from input_file and is added to fifop1. Fifop1 is the fifo between processor1 and switch1. Then, start entry of swicth1 is called and switch1 starts it operation. If stop has been called, the loop terminates.

```
task body processor1 is
    input_file : file_type;
begin
    open(input_file);
    loop
        select
            accept start do

    if(not(end_of_file(input_file))) then

    get(input_file,fifop1.input);
                    fifop1.tlm_fifo.add;
                    switch1.start;
                end if;
            end start;
            or
            accept stop;
                exit;
        end select;
    end loop;
    close(input_file);
end processor1;
```

Figure 13. Processor1 task body

Figure 14 shows the body of switch1 task. A Boolean variable called flag is defined which is used in making routing decisions. At the beginning of the task, it enters an infinite loop. After accepting *start* entry, if the fifo between switch1 and processor1 (fifop1) is not empty, a packet is removed from it and is routed based on the routing algorithm that we describe here. If the switch has only one output port, the packet is sent to that port. Otherwise, two situations may arise. If the packet is targeted for an immediate neighbor of this switch, it will be sent to the corresponding port. Otherwise, the packet will be sent to one of the output ports alternatively using flag variable. As shown in Figure 14, if the packet is targeted for Switch2 or if the packet is not targeted for switch3 and flag is false, the packet is sent to Switch2 and start entry for this switch is called. Otherwise, the packet is sent to Switch3 and start entry for this Switch is called. Upon calling stop entry of Switch1 task, the loop terminates.

```
task body switch1 is
    flag : boolean := false;
begin
    loop
        select
            accept start do
                if (fifop1.empty_flag=false)
then
                    fifop1.tlm_fifo.remove;
                    if
(fifop1.output.destination=2 or
(fifop1.output.destination/=3 and flag=false))
then
                        flag:=true;

    fifo1.input:=fifop1.output;
                        fifo1.tlm_fifo.add;
                        switch2.start;
                    else
                        flag:=false;

    fifo2.input:=fifop1.output;
                        fifo2.tlm_fifo.add;
                        switch3.start;
                    end if;
                end if;
            end start;
            or
            accept stop;
                exit;
        end select;
    end loop;
end switch1;
```

Figure 14. Switch1 task body

Figure 15 shows the body of simulation task. After accepting *start* entry, the task enters an infinite loop in which the stop entry is checked first. If stop has been called, the loop terminates. Otherwise, start entry of processor1 task is called and processor1 starts its simulation. Processor1 will call the start entry of switch1 task and switch1 will call the appropriate entry of switch2 or switch3 based on the routing decision. Upon arriving at destination, the packet will be processed by the destination processor and then routed toward Switch7 where it is written into the output file. This process repeats for the rest of the packets generated at processor1. Once he stop entry of simulation task is called, the loop terminates and stop

entries for switches, processor, and fifo tasks including fifos between switches and fifos between switches and their processors are called and simulation ends.

```
task body simulation is
begin
    accept start;
        loop
            select
                accept stop;
                    exit;
                else
                    processor1.start;
            end select;
        end loop;
    switch1.stop;
    …
    switch7.stop;
    processor1.stop;
    …
    processor7.stop;
    --stop all
    …
end simulation;
```

Figure 15. Simulation task body

7. Conclusion

Since digital designs are getting more complex, the need for a system description language that supports TLM is obvious. An important requirement at this level is to be able to run hardware of a system along with its software. Ada with intrinsic concurrency, early error detection, and extensive support for multithreading and multiprocessing would be a good choice to satisfy this requirement.

Other features of an HDL at a certain level of abstraction include link to its immediate lower abstraction level. In this paper, we showed how an RTL package could be used to override Ada with a register transfer capability.

Finally, an essential feature of an HDL at the transaction level is to be able to model transaction based communication channels for concurrent communication between processes. For this purpose, we showed how Ada could be used for defining the *tlm_fifo* channel functionality and implementing a NoC system using this channel.

With the features inherent in Ada and packages added to this language, it could be a good candidate for transaction level description of hardware. Such a language is a required part of evolution of hardware design to higher levels of abstraction.

7. References

[1] S. Mirkhani, and Z.Navabi, *The VLSI Handbook*, Chapter 86, CRC Press, 2ed Edition, 2006.

[2] S. Wong, and Gertrude Levine, "Kernel Ada to Unify Hardware and Software Design", *Proc. Annual ACM SIGAda International Conference on Ada*, 1998, pp. 28-38.

[3] "Reusable Software Components for Reusable Hardware Designs", 2009-01-10, Available at: http://alpha.fdu.edu/~levine/reuse_course/columns/HDL_column.

[4] "A History of Object-Oriented Programming Languages and their Impact on Program Design and Software Development", 2009-01-10, Available at: http://jeffsutherland.com/papers/Rans/OOlanguages.pdf.

[5] R. Goering, "ESC: Ada 2005 Speaks to Real-time Embedded Applications", 2007-4-2, EE Times, Available at: http://www.embedded.com/news/embeddedindustry/198701828?_requestid=308128.

[6] J. E. Sammet, "Why Ada is not Just another Programming Language", *Communications of the ACM*, vol. 29, no. 8, August 1986, pp. 722-732.

[7] S. Swan, "OSCI SystemC TLM", 2009-01-10, Available at: http://www-ti.informatik.uni-tuebingen.de/~systemc/Documents/Presentation-13-OSCI_2_swan.pdf.

[8] "SystemC TLM1.0", 2009-01-10, Available at: http://www.systemc.org/home.

[9] Ada Reference Manual, ISO/IEC 8652:2007(E) Ed. 3, pp. 471-474.

[10] D. A. Wheeler, "Lovelace tutorial", Section 16.1-General Information on Interfacing to Other Languages, 2009-01-10, Available at: www.dwheeler.com/lovelace.

[11] GHDL guide, section 5.8.5 Linking with Ada, 2009-01-10, Available at: http://ghdl.free.fr/ghdl/Linking-with-Ada.html#Linking-with-Ada.

[12] M. Mills, and G Peterson, "Hardware/Software Codesign: VHDL and Ada 95 Code Migration and Integrated Analysis", *Proc. Annual ACM SigAda international conference on Ada*, 1998, pp. 18-27.

[13] Ada Reference Manual, ISO/IEC 8652:2007(E) Ed. 3, pp. 181-186.

[14] "Introductory Ada Concurrency Summary", 2009-01-10, Available at: http://www.seas.gwu.edu/~csci51/fall99/ada_task.html.

[15] D. A. Wheeler, "Lovelace tutorial", Section 13.2-Creating and Communicating with Tasks, 2009-01-10, Available at: http://www.dwheeler.com/lovelace/s13s2.htm.

[16] Negin Mahani, Parniyan Mokri, Mahshid Sedghi, Zainalabedin navabi, "System Ada : An Ada based Syste- Level Hardware Description Language" ACM SIGADA AdaLetters, vol. XXIX , no. 2 , August 2009 , pp. 15-19.

[17] Negin Mahani, Parnian Mokri, Zainalabedin Navabi.\, "System Level Hardware Design and Simulation with SystemAda" Proceedings of IEEE East-West Design & Test Symposium (EWDTS'08), October 2008, pp. 190-195.