Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 245 (2009) 3–22
www.elsevier.com/locate/entcs

Modeling and Analyzing the Implementation of Latency-Insensitive Protocols Using the Polychrony Framework

Bin Xue[1] Sandeep K. Shukla[2]

FERMAT Lab
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA

Abstract

As Globally Asynchronous and Locally Synchronous (GALS) based System-on-chip (SoC) are gaining importance, a special case of GALS when the global clocking is preserved, but the interconnect delays of multiple clock cycles are to be tolerated has also been proposed, and used. In some cases, such designs, known as Latency-Insensitive Protocol (LIP) based SoC integration are also general enough to work when the global clocking is not present. In either case, the protocols are complex, and many optimized implementations of such protocols need verification that indeed they work correctly with respect to the specification of the system. Usually the specification of the system is fully globally clocked with negligible interconnect delays, so that the specification can be first implemented as synchronous design with traditional tools. GALS or LIP refinements are then applied to tolerate the multi-cycle interconnect delays, or fully asynchronous interconnect communication, as the case may be. Verifying that such refinements are correctness preserving, researchers have used model checking in the past. In this paper, we present static analysis based framework for such verification. Our framework makes use of the Polychrony framework and associated semantic analysis techniques, in the form of endo-isochrony. We show a number of LIP protocols to preserve the correctness with respect to their fully synchronous specifications using our framework. We believe, designers can verify LIP implementations with clever optimizations using our framework much more readily than when using model checking.

Keywords: latency-insensitive protocols, SIGNAL language, endochrony, isochrony, synchronous data flow graph, clock hierarchy

1 Introduction

Research on Globally Asynchronous and Locally Synchronous (GALS) system is attracting a lot of attention recently, and is expected to solve many problems faced by the current IC industry [8] as well as embedded system design [4]. The problem we consider here is germane in System-on-Chip (SOC) design. Built from predesigned IP modules, the SOC design methodology dramatically shortens the design cycle

[1] Email: xbin114@vt.edu
[2] Email: shukla@vt.edu

doi:10.1016/j.entcs.2009.07.025

and facilitates the modification and customization at system level. However, clocks of predesigned modules from different manufacturers may not be optimized for the same frequency which brings forth a global synchronization problem. In this situation, each module works synchronously under their own local clocks while the communication between different modules are asynchronous, which forms a GALS system. The other issue here is that intensive integration on one chip and ever growing clock speed exacerbate the interconnected delay, making it more difficult for one signal to propagate through the entire chip within one clock cycle. Partitioning large system into several clock domains is a possible solution, which again forms a GALS system. Furthermore, the commercial world demands power efficient chips to increase the battery life for laptops, cell phones and other portable devices. GALS architecture allows fine-grained power management through functional block activation.

Latency-Insensitive Protocol (LIP) based communication is a subclass of GALS communication which maintains the methodology of classic synchronous design flow while mitigating the multi-cycle interconnected delays [7]. The idea is based on retiming that inserts latch based protocols called *relay stations* to compensate the long interconnect delays and each process is encapsulated with a wrapper for correct interaction with the relay stations. [6] [17] [22] provides a fully formal and cycle accurate description of the mechanism of LIP amenable for formal analysis. [9] simplifies the original LIP protocol by using a scheduling algorithm for the functional block activation which dramatically reduces routing resources, area and complexity of the gating structures. A practical implementation is Jacobson's synchronous interlocked pipeline [14] which pipelines the handshaking signals *valid* and *stall* with the transferred data to avoid a global propagation of stall signals. Cortadella [10] presents a simple LIP protocol called Synchronous Elastic Flow (SELF) which inherits the ideas of the original LIP and synchronous interlocked pipelines and proposes the implementation of Elastic Buffers (EB). You et. al. [25] report the performance of their new LIP network fabric protocols called the Phase Synchronous Elastic Flow (pSELF) which is similar to SELF but compatible with both synchronous and asynchronous interfaces.

Implementation of GALS calls for innovate verfication methods, since the verification tools for synchronous design can't be directly applied to the asynchronous system. However, few verification tools have prevailed for asynchronous and GALS design though there are some attemps. Rostislav et. al. [12] propose a way of converting signal transition graphs of asynchronous protocols into PSL statements and employ assertion based verification tool to do complete verification. In [16] , a verification framework based on process space is employed, where a new data transition model representing the implicit relationship between clock and data validity events is proposed and a comprehensive implementation models for the asynchronous wrapper and the asynchronous communication scheme is constructed; In [11], Dasgupta et. al. propose deadlock tolerant GALS ring architecture using Petri Net specification and used model checking tool for reachability analysis and deadlock check. [21] verifies the asynchronous hardware designs specified in CHP, a VLSI programming

language based on process calculi, using exiting model checking tool CADP that are based on exploration of LTSS (labeled transition system). Specially for LIP, [22] propose a framework to validate the families of LIP using Spin model check for latency equivalence. It also model the LIP with SML to validate the functional correctness by a programming based simulation technique. Most of the techniques mentioned above use model checking technique which verifies the protocol by exploring all of the reachable states.

On the other hand, the distributed embedded software community has been working on GALS in a slightly different setup for at least two decades [3] [2] [5]. Synchronous language SIGNAL is developed to allow people describe their designs with concurrency specification and concentrate on the functional correctness without considering the timing issue arising in the real implementation, especially for a distributed environment where the concept of GALS applies. The task of generating and verifying an implementation independent from communication delay, is handled by the SIGNAL compiler Polychrony and a verification tool SIGNALI. In [3] [2], Benveniste et. al. formally model the synchronous process, asynchronous process, desynchronization, resynchronization and composition of processes under synchronous or asynchronous environment. Based on these works, formal definition of endochrony and isochrony are proposed. Endochrony refers to the property of a process which enforces unique scheduling and deterministic interaction with its asynchronous environment. Isochrony refers to a delay-independent communication between a pair of processes. Endochrony and isochrony together forms endo-isochrony which is a sufficient property for correct GALS implementation [3] [2] for distributed embedded software. In [20], the property of weakly endochrony, is proposed to describe less strict deterministic processes which have multiple valid schedules. In [24], Talpin et. al. formalize the functionality of LIPs and transform the synchronous modules of a multiclock synchronous specification into weakly endochronous modules, for which simple and efficient wrappers exists. In [23], Talpin unifies the verification of endochrony, isochrony and weakly endochrony by checking the formal structures Synchronous Data Flow Graph (SDFG) [18] and Clock Hierarchy (CH) [1]. The theory concerning GALS combined with the formal tools provide a possible solution of checking correct communication for a given system. However, most of these works stay in the embedded software domain. To the best of our knowledge, few work has been done for formalizing realistic GALS implementations, especially for the designs described at the RTL or gate level. This paper aims at bridging the gap between the work on latency-insensitive hardware design and the work on GALS concept in the context of embedded software.

The paper presents a framework of modeling and analyzing the communication for LIPs based on the Polychrony framework. The Polychrony framework consists of the SIGNAL language and various analysis tools and software synthesis tools. The static analysis of various properties of a SIGNAL program are done using formal structures of SDFG and CH which we use here as well. In our framework for analysis of LIP based system, a given LIP is firstly modeled by the SIGNAL syntax which is further transformed to get its SDFG and CH. Next, endo-isochrony is checked

based on the obtained SDFG and CH and their respective composition. A case study of modeling and analyzing Phase Elastic Buffer (pEB) is done to demonstrate the flow and validate the proposed framework. The prominent distinction between our framework and other verification techniques for GALS is that we are using static analysis which is an efficient technique. Also, this work brings the concept of Polychrony to real hardware level LIP based SOC communication.

The paper is organized in the following order. Section 2 reviews some background knowledge required for later discussions. Basic syntax of SIGNAL including the operators, clock equations are introduced. Formal structures SDFG and CH and their construction from SIGNAL are briefly discussed. Moreover, the concepts of endo-isochrony and its verification are introduced. Section 3 demonstrates our attempt to model basic hardware components such as combinational logics, latches and registers as well as finite state machine using SIGNAL which shows how to model LIP with SIGNAL. We present our framework at the beginning of Section 4 and demonstrate the flow through a case study of pEB in Section 4 and Section 5. Additional analysis results of other LIPs are given as well. In the end, this paper concludes with discussion on future work in Section 6.

2 Background

2.1 SIGNAL language

SIGNAL is a synchronous programming language for real time applications [13], which allows describing a system with concurrency specification and verify the functional correctness without considering the actual implementation. In a SIGNAL process, signals like x,y can update at different frequencies determined by their "clocks", denoted as $\widehat{x}$ and $\widehat{y}$. The clock of a signal indicates the set of instants at which this signal is absent or present. The relations of clocks are implicitly included in SIGNAL operations shown in Table 1.$[C]$ refers to the set of instants when boolean variable $C = true$. Clocks can also be explicitly related using clock equations shown in Table 2. One can refer to [5] for the detailed explanation of SIGNAL syntax shown in Table 1 and Table 2.

SIGNAL operator	SIGNAL expression	Clock relation	Conditional dependency
Function	$Y = f(X_1, X_2, \ldots, X_N)$	$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_N}$	$X_i \xrightarrow{\widehat{X_i}} Y$
Downsampling	$Y = X\ when\ C$	$\widehat{Y} = \widehat{X} \wedge [C]$	$X \xrightarrow{\widehat{X} \wedge [C]} Z$
Merge	$Y = X\ default\ Z$	$\widehat{Y} = \widehat{X} \vee \widehat{Z}$	$X \xrightarrow{\widehat{X}} Y \xleftarrow{\widehat{Y}/\widehat{X}} Z$
Delay	$Y = X\ \$\ init\ y_0$	$\widehat{Y} = \widehat{X}$	$X(i-1) \rightarrow Y(i), i > 0, Y(0) = y_0$

Table 1
Conditional dependencies for kernel operations of SIGNAL language [5]

2.2 SDFG and CH

One of the tasks of the SIGNAL compiler is to transform the concurrent specification written in SIGNAL language to a sequential implementation in C language.

Therefore a valid schedule of the concurrent specification must be found. SDFG is a formal structure used by the compiler to find scheduling. Each basic operation of SIGNAL has its own *conditional dependency* [19] shown in Table 1. SDFG is constructed by composing all of the conditional dependencies [18]. A simple SIGNAL process and its SDFG(bottom left) is shown as in Figure 1. A valid schedule forbids *combinational loop* in SDFG. Combinational loop in SDFG refers to that

```
process A= (? integer a,b;  ! integer c,d,e;)
   (| c := a - b
    | d := a + 1 when c > 0
    | e := 2 * b when c <= 0 |)
```

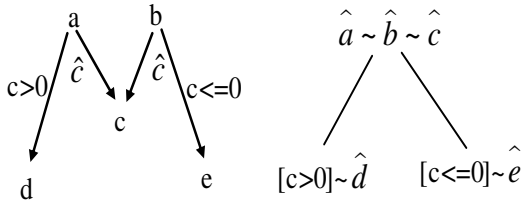

Fig. 1. SIGNAL description of process A and its SDFG and CH

1. The loop without buffers and which is expressed by "$" in a SIGNAL process. For example, in `BX := X $ init 0`, signal `BX` takes the previous value of `X` and which can be treated as a buffer of `X`.

2)The loop with effective conditions. If there is a loop in the path, forexample, $a_1 \xrightarrow{c_1} a_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} a_n \xrightarrow{c_n} a_1$, it must satisfy $\bigwedge_{i=1}^{n} c_i$ to avoid a combinational loop. In other words, all the conditions along this loop should not be true at the same time. Figure 2 shows two processes B, C and their SDFG. Process B has a combinational loop between signal c and d. On the other hand, process C with a slight change has no combinational loop, because there are buffers `zc` and `zd` in the loop.

Another important aspect of the compiler is to synthesize the clock relations by formal structure CH. In a given CH, any child node clock is a down-sample of its ancestors. All of the synchronized clocks are grouped as one node in CH. [1] illustrates the algorithm to construct the CH for a given SIGNAL process. The CH of process A is shown in Figure 1(on the right) as well. Root node $\hat{a} \sim \hat{b} \sim \hat{c}$ means that signal a, b, c are synchronized and have the fastest clock. $[c > 0] \sim \hat{d}$ as a

Clock relations	Clock equations	Explanation
Synchronization	$Y\hat{}= X$	signal Y is synchronized with signal X
Union	$Y\hat{}=X\hat{}+Z$	values of signal Y are on the instants of signal X or signal Z
Intersection	$Y\hat{}=X\hat{}*Z$	values of signal Y are on the same instants of signal X and signal Z
Deference	$Y\hat{}=X\hat{}-Z$	values of signal Y are on the instants of signal X but not signal Z

Table 2
Explicit clock relations using clock equations

```
process B= (? integer a,b;
            ! integer c,d;)
   (| c := a + b
    | d := b + c |)
```

```
process C= (? integer a,b;
            ! integer c,d;)
      (| c := a + zd
       |zc := c $ init 0

       | d := b + zc
       |zd := d $ init 0 |)
where
    integer zc, zd;
end
```

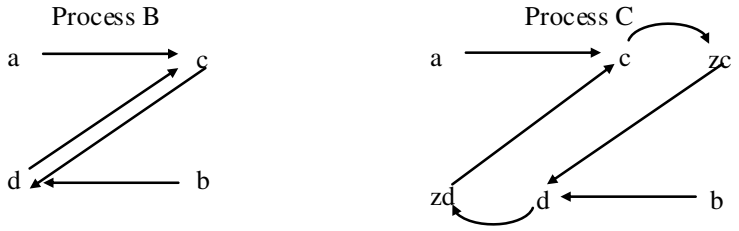

Fig. 2. Process B with a combinational circle and Process C without combinational circles.

child node indicates that clock of d is synchronized with the clock $[c > 0] = true$, and they are downsamples of clock c. Sometimes, there might be contradict clock relations within one CH, and which is formally defined in Talpin's paper [23] as the ill-formed hierarchy.

Definition 2.1 *[23] A CH is ill-formed if and only if:*
1. There exists any boolean signal x, that $\widehat{x}$ is a childnode of either $[x]$ or $[\neg x]$, where $[x]$ represents "when $x = true$" and $[\neg x]$ represents "when $x = false$".
2. There exist signal $b1$, $b2$ and $\widehat{b1}$ is a ancestor of $\widehat{b2}$, where $b1 = c1 fc2$ and $\widehat{c1}$, $\widehat{c2}$ are childnodes of $\widehat{b2}$, $f = \{\vee, \wedge, \backslash\}$.

In 1, $\widehat{x}$ is a childnode of its downsamples $[x]$ or $[\neg x]$. In 2, since $b1 = c1 fc2$, $b1$ should be a childnode of the common ancestor of $\widehat{c1}$ and $\widehat{c2}$. Here the common ancestor is $b2$, then $b1$ is a childnode of $b2$, and it contradicts that $\widehat{b1}$ is a ancestor of $\widehat{b2}$. Therefore, ill-formed hierarchy results from contradict clock inclusion, and which is explained by an example given in Section 5. SDFG and CH are used to check communication correctness between interconnected processes.

2.3 Endochrony, Isochrony and GALS distribution

Endochrony and Isochrony are properties to ensure correct operation of synchronous processes working in an asynchronous environment. The formal definitions and proofs of endochrony and isochrony can be found in [4] and [3]. Limited by the space in this paper, we will only talk about their application for GALS and their verification, which is the foundation for our framework.

Endochrony ensures that a process interacts with an asynchronous environment correctly. An endochronous process can uniquely resynchronize a group of

dataflow to synchronous state transition [4] without external information. An endochronous process has a unique sequential scheduling.

Isochrony enables correct communication between two processes P_1 and P_2. If two processes P_1 and P_2 are isochronous, their communication is independent from the channel delay. In other words, they can behave equivalently as they are interacting through synchronous channels.

GALS refers to a globally asynchronous system made of locally synchronous components, communicating via asynchronous communication media. Endochrony and isochrony provide a sufficient solution for mapping a synchronous program onto a group of distributed processors [2].

Property 1: Suppose we have a system P which consists of a finite collection of processes $P_i(i = 1, 2, 3, \ldots, n)$, P is endo-isochronous if

1. each P_i is endochronous;
2. each pair (P_i, P_j) is isochronous.

If P is endo-isochronous, when each P_i works in an asynchronous environment and communicates with others asynchronously, then P will have the equivalent behavior as when all P_is communicate synchronously. Endo-isochrony provides a sufficient solution for GALS design.

Verification of endo-isochrony is employed from Talpin's paper [23] and we come up with our framework. Talpin has pointed out that

Property 2: A process is endochronous iff the process is:

1. acyclic;
2. without ill-formed clock;
3. its CH has a unique root.

Condition 1 can be easily verified by checking its SDFG and conditions 2 and 3 can be verified during the procedure of construction of the CH. Since it is possible to get the SDFG and CH from a given SIGNAL process, they can be used to check endochrony for any system modeled by SIGNAL. Talpin also pointed out the way of checking isochrony [23]

Property 3: (P, Q) is isochronous, if two P and Q satisfy,

1. $P \| Q$, the synchronous composition of P and Q is acyclic;
2. P and Q have no ill-formed clocks;
3. $P \| Q$ has no ill-formed clock.

Similarly, condition 1 can be easily checked by analyzing the composition of the SDFG of $P \| Q$, condition 2 can be checked by observing CHs of P and Q. We can compose the CH of P and Q to see if there is contradicting clock relations to verify 3. Analyzing the clock relations of two processes is fully described in another paper of ours [15]

3 Modeling Register Transfer Level (RTL) elementary hardware using the SIGNAL language

It is well known that SIGNAL is developed for modeling embedded software. However, most LIPs' implementation are described at lower level such as gate or RTL level. In this section, we describe how the SIGNAL language can effectively model hardware described at gate or RTL level as well. Based on this fact, it is reasonable for us to model more complicated LIP implementation. Generally speaking, the basic components of hardware design include combinational logic, sequential logic such as registers and latches, and sometimes Finite State Machine (FSM) to generate control signals. In the following context, we will use SIGNAL to describe combinational logic, latches and registers and FSM respectively.

3.1 Modeling combinational logic

Combinational logic is a type of logic circuit whose output is a pure function of the present input only, which can be expressed as a triple $\{I, O, F\}$. I is the set of inputs, O is the set of outputs, F is the set of functions that can be formed by $\{AND, OR, NOT\}$ or their variants. For $\forall o \in O$, $\exists f \in F$ and $\exists i_1, i_2, \ldots, i_m \in I$, $o = f(i_1, i_2, \ldots, i_m)$. Modeling combinational logic with SIGNAL is direct. Each varable in $\{I, O\}$ can be modeled as a boolean or integer signal. Each function in F can be modeled using boolean or arithmetic functions.

3.2 Modeling latches and registers

Latches and registers can store their previous value until the trigger conditions are satisfied. Here we use latch to denote level-triggered storage and register for edge-triggered storage. In Figure 3, a register triggered at positive edge and its SIGNAL description is shown. Internal signal `ZDout` is defined as the previous value of `Dout` to model the storage. "`when (clk and not zclk)`" expresses the positive edge-triggered condition, where `zclk` is the previous value of `clk`. One can use "`when clk`" ("`when not clk`") to express a high (low) level-triggered condition. "`Dout := Din when (clk and not zclk) default ZDout`" assigns `Dout` to `Din` when the trigger condition is true, otherwise it will take its previous value `ZDout`. `Din^= Dout^= clk` synchronizes the three signals. (i.e, as clk changes value, the value of Din is sampled, and the value of Dout is computed).

3.3 Modeling FSM

The finite state machine (FSM) is generally used for control logic, which can be implemented by combinational logic and registers. However, it will be much simpler to express FSM directly from its state transition diagram. Figure 4 shows an FSM with three states IDLE, REQ and WAIT. In its SIGNAL description, `ack` and `v` are inputs and `req` is the output, `NS` represents next state of FSM and `S` denotes the current state.

```
process register= (? integer Din; boolean clk;
                   ! integer Dout;)
   (| Dout := Din when (clk and not zclk)
      default ZDout
    | ZDout := Dout $ init 0
    | zclk  := clk $ init 0
    | Din ^= clk ^= Dout |)
where
    integer ZDout; boolean zclk;
end
```

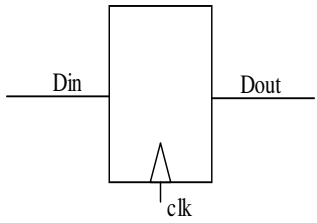

Fig. 3. A register and its SIGNAL description

```
% State transition of a FSM %
% S:current state, NS:next state %
% S = 0:IDLE, S =1:REQ, S = 2:WAIT %
process FSM = ( ? boolean ack, v;
                ! boolean req; )
 (| NS := 1 when S = 0 when v default
          2 when S = 1 when ack default
          0 when S = 2 when not ack
          default S
  | S := NS $ init 0
  | req := 1 when S = 1 default 0
where
    integer S, NS;
end
```

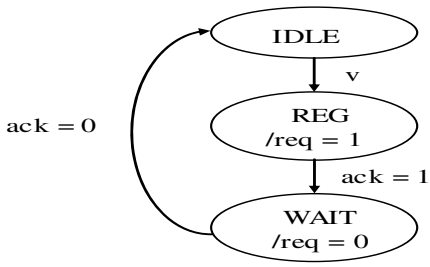

Fig. 4. A finite state machine and its SIGNAL description

4 Modeling LIP SIGNAL

Now we are going to resend our methodology to model and verify the implementation of an LIP. The methodeology has the following steps:

1. Model each component Pi using the SIGNAL language;
2. Construct SDFG and CH of each Pi;
3. Check endo-isochrony of the design by the sets of SDFG and CH;
4. If endo-isochrony is satisfied, the design has correct communication.

We will demonstrate this methodology by modeling a particular latency insensitive design in Section 4 and analyzing its endo-isochrony in Section 5. In the end, the analysis results of other latency-insensitive designs will be provided as well.

In this section, we choose to model pSELF proposed by You, et. al. pSELF is a handshaking protocol used for a communication fabric and interface between Intellectual Property (IP) cores, and it traditionally applies to clocked systems. pSELF supports communication with both asynchronous and clocked logics [25]. pEB is one of pSELF protocols as shown in Figure 5 from [25]. One pEB consists of two phase elastic half buffers(pEHB) implemented by latches, which are enabled on different clock phases. In this figure, the clock signal is not explicitly shown for

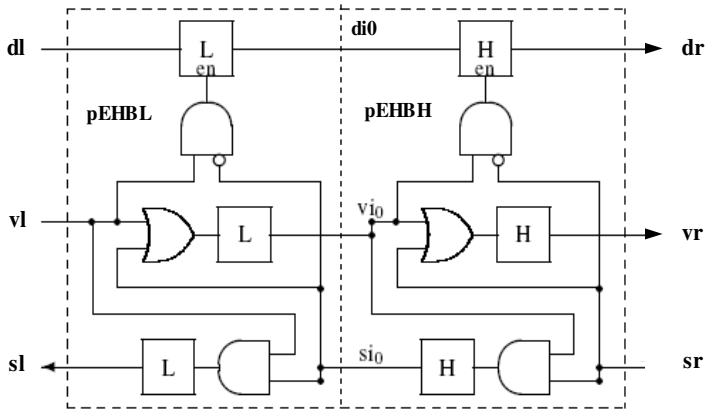

Fig. 5. Phase synchronous elastic buffer [25]

simplicity. The pEHB with a L-latch is named as pEHBL and the pEHB with a H-latches is named as pEHBH. vl is the valid signal from previous stage to indicate an effective value of input dl, sr is the stall from the next stage to indicate filled buffers and to stop the current pEB generating new outputs. Both vl and sr can propagate through the current pEB. This pEB is modeled as a SIGNAL process pEB and shown in Figure 6. Process pEB calls two subprocesses pEHBL, pEHBH with interconnected signals `di0`, `vi0` and `si0`. Due to space limitation, we only give the code of pEHBL. pEHBH is same as pEHBL, except for latches. In subprocess pEHBL, it firstly synchronizes input signals `Vl, Sr, Clk` and `Dl` by clock equation `^=` since it's a clocked design. Then it generates the outputs of the control signals `Vr` and `Sl` as is quite obvious from the circuit diagram. (`Zvr, Vr`) and (`Zsl, Sl`) model behaviors of the latches in the control path. After that, the enable signal of the data path is generated by the logic. At last, the data path is modeled by `DL` which represents the characteristic of the latch.

5 Analysis of Endochrony and Isochrony of LIP using SDFG and CH

In this section, we will show the flow of analyzing endochrony and isochrony by a case study of pEB modeled in Section 4. In addition, we will provide analysis results of other LIP protocols.

5.1 Analysis of Endochrony for pEB

Using the SIGNAL compiler description we can extract the SDFG (Figure 7) and the CH(Figure 8) for process pEB. We need to analyze SDFG to check out combinational loops and CH for ill-formed clocks and single root respectively. To ease the expression, vll and srl are used to represent the internal signals of pEHBL and enl is the enable signal of pEHBL's latch and vlh, srh and enh are named for pEHBH in the same way. Notice that in this SDFG, loops between DL and ZDL, DH and

```
process pEB= (? boolean vl, sr, clk; integer dl;
              ! boolean vr, sl; integer dr;)
   (| (vi0, sl, di0) := pEHBL(vl, si0, clk, dl)
    | (vr, si0, dr)  := pEHBH(vi0, sr, clk, di0) |)
where
    boolean vi0, si0;
    integer di0;
    %subprocesses EHBM, EHBL%
    process pEHBL = (..., ...);
    process pEHBH = (..., ...);
end;

%subprocess EHBL%
process pEHBL = ( ? boolean Vl, Sr, Clk; integer Dl;
                  ! boolean Vr, Sl; integer Dr;)
    %synchronize input signals' clocks%
   (| Vl ^= Sr ^= Clk ^= Dl
    %Compute Vr and Sl%
    | Vll := Vl or Sr
    | Vr  := Vll when (not Clk) default Zvr
    | Zvr := Vr $ init false
    | Vr ^= Vl
    | Srl := Vl and Sr
    | Sl  := Srl when (not Clk) default Zsl
    | Zsl := Sl $ init false
    | Sl ^= Sr
    %Date path control%
    | Enl  := Vl and (not Sr)
    %Data path%
    | DL  := Dl when Enl when (not Clk) default DL $ init 0
    | Dr  := DL
    | DL ^= Clk |)
where
    boolean Vll, Zvr, Zsl, Zrl, Enl;
    integer DL;
end
```

Fig. 6. SIGNAL description of pEB

ZDH, sl and zsl, vr and zvr, si0 and zsi0, vi0 and zvi0 are not combinational loops. Another loop vll→vi0→srh→si0→vll, is also not an effective combinational loop, because the path between vll to vi0 and the path between srl to si0 can not be effective at the same instant. Above all, there is no effective combinational loop in this SDFG. Next, let's move to its CH. As a clocked design, all of the signals in pEB are synchronized with signal Clk and therefore its CH has a special form with only one node. This node is also the root node. It's easy to see that this CH has no ill-formed clock and has a unique root. According to Property 2 in Section 2.3, process pEB is endochronous.

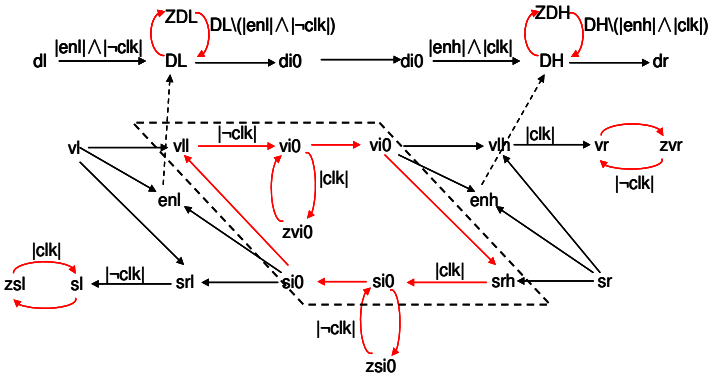

Fig. 7. SDFG of one pEB

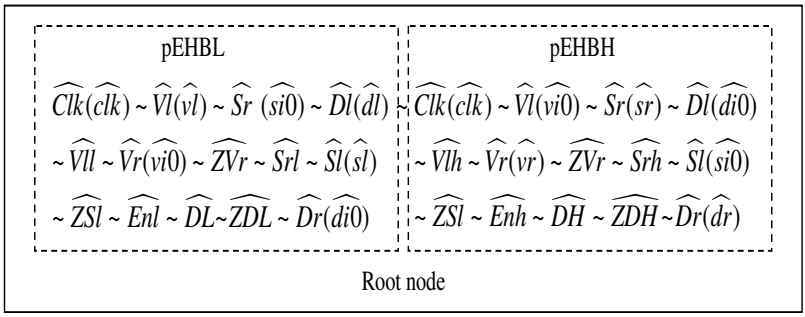

Fig. 8. CH of one pEB

5.2 Analysis of composition for two pEBs

Let's move to two pEBs, denoted as pEB1 and pEB2. We can construct their CHs and SDFGs in the same manner and which are named as CH1 and CH2, SDFG1 and SDFG2 respectively. Since both CH1 and CH2 have only one node, the composition of the two will not give any ill-formed clocks, see Figure 9. This case is too special and we would like to give readers another counter example to see how ill-formed clock could be generated when composing two different processes. See processes D and E and the composition of their CH's are given in Figure 10. s1 and s2 are communication signals shared by both processes. In CH(D) of process D, s2 is a downsampling of s1 and is present when $s1 > 0$. On the other hand, in CH(E) of process E, s1 and s2 are synchronized such that s2 is present when s1 is present. Therefore, composition CH(D) and CH(E) leads to the contradiction that s1 is present when $s1 > 0$ which gives a ill-formed clock.

Next, we look at the composition of SDFG1 and SDFG2 in Figure 11. We only need to observe the interface, because we already know from the previous discussion that both SDFG1 and SDFG2 have no combinational loops. Similarly, we can observe that loop vlh→vr→vl→srl→sl→ sr→vlh is not effective and therefore the composition SDFG is acyclic. Above all, we can conclude that the composition of pEB1 and pEB2 are isochronous and therefore a system composed by several pEBs

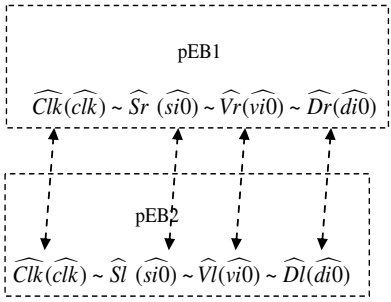

Fig. 9. The composition of CH1 and CH2

```
process D= (? integer a,b;                process E= (? integer s1,s2;)
            ! integer s1,s2;)                        ! integer c;)
   (| s1 := a + b                           (| c := s2 + s1 |)
    | s2 := a + 1 when s1 > 0 |)
```

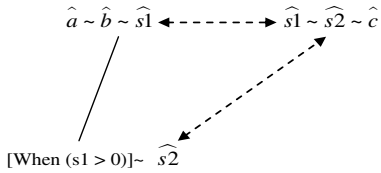

Fig. 10. Composition of process D and E has ill-formed clocks

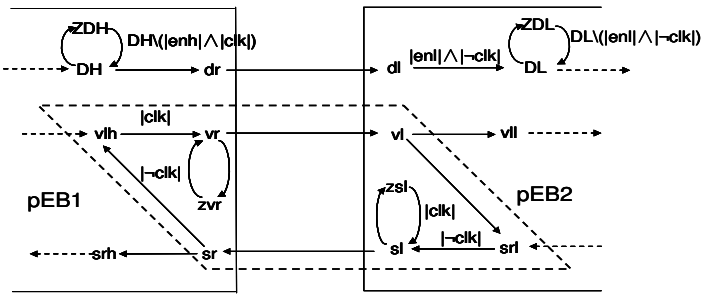

Fig. 11. composition of SDFG1 and SDFG2 of pEB1 and pEB2

is endo-isochronous and have correct function as if they are working in synchronous environment.

5.3 Other analysis results

Apart from the example given in Section 5.1 and 5.2, we have also modeled and analyzed several other LIP protocols listed below. One can refer to the Appendix for the detailed SIGNAL codes. The analysis results are given in Table 12

1. Relay station(RS),wrapper(W) from Carloni et. al. [7];
2. Synchronous interlocked pipelines(SIP) from Jacobson et. al. [14];

3. SELF, elastic half buffer master/slave(EHBM, EHBS), fork, join, eager fork from Cortadella et. al. [10];

4. pSELF(pEHBL, pEHBH), interleaving to synchronous protocol(int2syn), synchronous to interleaving protocols(syn2int) from [25].

The results are as expected. All of these LIP listed above are clocked designs and the communication between them are synchronized, therefore isochrony is easily satisfied with the same reason as mentioned in Section 5. On the other hand, RS, W, SIP, SELF are based on Elastic Buffer(EB) with handshaking signals *valid* and *stall*, and their structures have much in common. At the same time, EB and pEB are similar. Therefore, it's not surprising to get the same results as Section 5.

Protocols	Endochronous?
RS	yes
Wrapper	yes
SIP	yes
EHBM	yes
EHBS	yes
fork	yes
join	yes
eagerfork	yes
pEHBL	yes
pEHBH	yes

Compositions	Isochronous?
RS——RS	yes
RS——Wrapper	yes
Wrapper——Wrapper	yes
SIP——SIP	yes
SELF(EHBM——EHBS)	yes
pSELF(pEHBL——pEHBH)	yes
int2syn(pEHBL——EHBM)	yes

Fig. 12. Analysis results of other LIPs

6 Conclusion and future work

In this paper, we have shown that the SIGNAL language and its related formal structures can be used to model LIPs and to verify their implementation using static analysis avoiding state space generation based methods such as model checking. With the SIGNAL syntax, the elementary hardware combinational logic, sequential logic and FSM can be correctly described. It enables the modeling of more complicated LIP. On the other hand, formal structures SDFG and CH constructed from SIGNAL description, can be used to analyze endo-isochrony and therefore verify the correct communication. These facts provide us an appropriate framework to model and verify the communication of LIP and which is demonstrated and validated through a case study. The next consideration is how to do this better. We are going to improve the data structures for SDFG and CH and their composition and an automation algorithm will be developed later. In addition, the protocol we analyzed in this paper is LIP, which is still a clocked design. The global synchronization simplifies the analysis of CH and CH's composition. However, without global synchronization the analysis of CH will become much more difficult. Therefore, further consideration will be given on checking ill-formed clocks based on CH.

Acknowledgement

We thank Kenneth Stevens (University of Utah), Mathew W. Heath (Intel Corp.), Marly E. Roncken (Intel Corp.), Jean-Pierre Talpin (INRIA), and Dimitrou Potop (INRIA) for various fruitful discussions regarding this work. This work was partially supported by NSF fund CCF- 0702316, and SRC task 1818.

References

[1] Amagbégnon, P., L. Besnard and P. L. Guernic, *Implementation of the data-flow synchronous language signal*, in: *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (1995), pp. 163–173.

[2] Benveniste, A., *Some synchronization issues when designing embedded systems from components*, in: *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software* (2001), pp. 32–49.

[3] Benveniste, A., B. Caillaud and P. L. Guernic, *From synchrony to asynchrony*, in: *Proc. of Concurrency Theory*, 1664 (1999), pp. 162–177.

[4] Benveniste, A., B. Caillaud and P. L. Guernic, *Compositionality in dataflow synchronous languages: Specification and distributed code generation*, Information and Computation **163** (2000), pp. 125–171.

[5] Bernard, *The synchronous programming language signal a tutorial*, Espresso project, HOUSSAIS IRISA (2004).

[6] Boucaron, J., J. V. Millo and R. D. Simone, *Another glance at relay stations in latency insensitive design*, , **146**, 2006, pp. 41–59.

[7] Carloni, L., K. McMillan and A. Sangiovanni-Vincentelli, *Theory of latency-insensitive design*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions **20** (2001), pp. 1059–1076.

[8] Carloni, L. P. and A. L. Sangiovanni-Vincentelli, *Coping with latency in soc design*, IEEE Micro **22** (2002), pp. 24–35.

[9] Casu, M. R. and L. Macchiarulo, *A new approach to latency insensitive design*, in: *DAC '04: Proceedings of the 41st annual conference on Design automation* (2004), pp. 576–581.

[10] Cortadella, J., M. Kishinevsky and B. Grundmann, *Synthesis of synchronous elastic architectures*, in: *DAC '06: Proceedings of the 43rd annual conference on Design automation* (2006), pp. 657–662.

[11] Dasgupta, S. and A. Yakovlev, *Modeling and verification of globally asynchronous and locally synchronous ring architectures*, in: *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe* (2005), pp. 568–569.

[12] Dobkin, R., T. Kapshitz, S. Flur and R. Ginosar, *Assertion based verification of multiple-clock gals systems*, Technical report, VLSI Systems Research Center, Technion Israel Institute of Technology (2008).

[13] Guernic, P. L., M. Borgue, T. Gauthier and C. Marie, *Programming real time applications with SIGNAL*, Proc. of the IEEE **79** (1991), pp. 1321–1335.

[14] Jacobson, H. M., P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer and C. J. Myers, *Synchronous interlocked pipelines*, in: *In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (2002), pp. 3–12.

[15] Jose, B. A., B. Xue and S. K. Shukla, *An analysis of the composition of synchronous systems*, in: *Formal Methods for Globally Asynchronous Locally Synchronous Design*, 2009.

[16] Kong, X., R. Negulescu and L. W. Ying, *Refinement-based formal verification of asynchronous wrappers for independently clocked domains in systems on chip*, in: *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (2001), pp. 370–385.

[17] Li, C.-H., R. Collins, S. Sonalkar and L. Carloni, *Design, implementation, and validation of a new class of interface circuits for latency-insensitive design*, in: *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2007.

[18] Maffeis, O. and P. L. Guernic, *Distributed Implementation of Signal: Scheduling & Graph Clustering*, Springer-Verlag (1994), pp. 547–566.

[19] Nowak, D., *Synchronous structures*, Inf. Comput. **204** (2006), pp. 1295–1324.

[20] Potop-Butucaru, D., B. Caillaud and A. Benveniste, *Concurrency in synchronous systems*, Formal Methods in System Design **28** (2006), pp. 111–130.

[21] Salaun, G., W. Serwe, Y. Thonnart and P. Vivet, *Formal verification of chp specifications with cadp illustration on an asynchronous network-on-chip*, in: *ASYNC '07: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems* (2007), pp. 73–82.

[22] Suhaib, S., D. Mathaikutty, D. Berner and S. Shukla, *Validating families of latency insensitive protocols*, IEEE Transactions on Computers **55** (2006), pp. 1391–1401.

[23] Talpin, J.-P., J. Ouy, L. Besnard and P. L. Guernic, *Compositional design of isochronous systems*, in: *DATE '08: Proceedings of the conference on Design, automation and test in Europe* (2008), pp. 928–933.

[24] Talpin, J.-P., D. Potop-Butucaru, J. Ouy and B. Caillaud, *From multi-clocked synchronous processes to latency-insensitive modules*, in: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software* (2005), pp. 282–285.

[25] You, J., Y. Xu, H. Han and K. S. Stevens, *Performance evaluation of elastic gals interfaces and network fabric*, Electron. Notes Theor. Comput. Sci. **200** (2008), pp. 17–32.

APPENDIX

A Additional SIGNAL codes for other LIPs

A.1 Wrapper consists of an Equalizer, an Extended Relay Station(ERS), a Stalling Signal Generator(SSG) from Carloni et. al. [7]. In this process, we use a simple core process ADDSUB to be wrapped. It computes the summation and subtraction of the two inputs.

```
process Wrapper = ( ? boolean vl1, vl2, sr1, sr2;
                      integer Sin1, Sin2;
                    ! boolean sl1, sl2, vr, vl3, Sp, Sr1, Sr2;
                      integer Sout1, Sout2;)
(| (SPI1, SPI2, sl1, sl2, vr):= Equalizer(Sin1, Sin2, vl1, vl2, vl3)
 | (SPO1, SPO2, Pvr) := ADDSUB(SPI1, SPI2, Pvl)
 | (Sout1, vr1) := ERS(SPO1, sr1, Pvr)
 | (Sout2, vr2) := ERS(SPO2, sr2, Pvr)
 |  Sr1 := sr1 $ init false
 |  Sr2 := sr2 $ init false
 |  Sp  := SSG(Sr1, Sr2)
 |  vl3 := not Sp |)
 |  Pvl := vr
where
     integer SPI1, SPI2, SPO1, SPO2; boolean Pvl, Pvr, vr1, vr2;
    %module of Equalizer%
     process Equalizer = ( ? integer Sin1, Sin2; boolean vl1, vl2, vl3;
                           ! integer Sout1, Sout2; boolean sl1, sl2, vr;)
     (| vr := vl1 and vl2 and vl3
      | clkin1 := vl1 and (not sl1)
      | clkin2 := vl2 and (not sl2)
      | nsl1 := vl1 and (not vr)
      | nsl2 := vl2 and (not vr)
      | sl1 := nsl1 $ init false
      | sl2 := nsl2 $ init false
      % Datapath %
      | Sin1 ^= when clkin1 = true
      | Sin2 ^= when clkin2 = true
      | E1 := Sin1 when (vl1 = true and sl1 = false)
              default E1 $ init 0
      | E2 := Sin2 when (vl2 = true and sl2 = false)
              default E2 $ init 0
      | Sout1 := E1 when vr = true default E1 $ init 0
      | Sout2 := E2 when vr = true default E2 $ init 0
      | nsl1 ^= nsl2 ^= vr
      | E1 ^= E2 ^= vr ^= vl1
      | Sout1 ^= Sout2 ^= vl1 |)
     where
          boolean nsl1, nsl2, clkin1, clkin2; integer E1, E2;
     end;
     %Module of SSG%
     process SSG = ( ? boolean Sr1, Sr2; ! boolean Sp;)
     (| Sp := Sr1 or Sr2 |);
     %Module of ERS%
     process ERS = ( ? integer Sin; boolean Sr, Vl;
                     ! integer Sout; boolean Vr;)
     (| (Sl, Vr, S, Sout) := EBFSM(Sin, Sr, Vl)
      |  Sr1 := Sr $ init false
      |  Sr1 ^= Vl |)
     where
          integer S; boolean Sl, Sr1;
          % This is a module of relay station %
          process EBFSM = ( ? integer Sin; boolean Sr, Vl;
                            ! boolean Sl, Vr; integer S, Sout;)
          (% clk generation
           | clk := not (clk $ init false)
           | Vl ^= when clk
           % State transition of FSM %
           | case1 := true when (S = 1) when ((not Vl) and (not Sr))
                      default false
           | case2 := true when S = 0 when Vl
                      default true when S = 2 when (not Sr)
                      default false
           | case3 := true when S = 1 when (Vl and Sr)
                      default false
```

```
            | case1 ^= case2 ^= case3 ^= Vl
            | NS := ( 0 when case1 = true)
                   default ( 1 when case2 = true)
                   default ( 2 when case3 = true)
                   default S
            | S := NS $ init 0
            | NS ^= Vl
            % Computation of Vr and Sl%
            | Vr := false when S = 0 default true
            | Sl := true when S = 2 default false
            | Vr ^= Vl ^= Sl ^= Sr
            % computation of Em and Es %
            | Em := true when S = 0 when Vl
                   default true when S = 1 when Vl
                   default false
            | Es := true when S = 0 when Vl
                   default true when S = 1 when ( Vl and (not Sr))
                   default true when S = 2 when not Sr
                   default false
            | Em ^= Es ^= Vl
            % datapath %
            | DL := Sin when Em default ZDL
            | DH := ZDH $ init 0
            | ZDH := DL when Es default ZDH $ init 0
            | DL ^= DH ^= Vl
            | Sin ^= Vl
            | Sout := DH when Es default ZDH
            | Sout ^= Sin |)
          where
               boolean case1, case2, case3, Em, Es;
               integer NS, ZDH, DL, DH;
          end
     end;
     %Module of stallable process: the core processor%
     process ADDSUB = ( ? integer Sin1, Sin2; boolean vl;
                        ! integer Sout1, Sout2; boolean vr;)
     (| Sout1 := Sin1 + Sin2
      | Sout2 := Sin1 - Sin2
      | vr := vl |);
end
```

A.2 One stage of Synchronous interlocked pipelines(SIP) from Jacobson et. al. [14].

```
process SIP = ( ? boolean validin, stallin, gclk; integer Sin;
                ! boolean stallout, validout; integer Sout;)
(% clock relation among inputs%
 | stallin ^= gclk
 | Sin ^= gclk
 | validin ^= gclk
 % valid signal %
 | valid1 := validin when not gclk when not rstall1 default rvalid1
 | rvalid1 := valid1 $ init false
 | validout := valid1
 | valid1 ^= validin ^= validout
 % stall signal %
 | stall1 := (rvalid1 and stallin) when gclk default rstall1
 | rstall1 := stall1 $ init false
 | stallout := stall1
 | stall1 ^= stallin ^= stallout
 % Data path %
 | Sout := Sin when validin when (not gclk) when (not stall1)
default L1
 | L1 := Sin $ init 0
 | L1 ^= Sin ^= Sout |)
where
     boolean stall1, valid1, rstall1, rvalid1;
     integer L1;
end
```

A.3 SELF, including Elastic Buffer(EB)(EHBM, EHBS), lazyfork, join, eagerfork. [10]

```
process EB = ( ? boolean Vl, Sr, Clk; integer Sin;
               ! boolean Vr, Sl; integer Sout;)
(| (Vm, Sl, Dm) := EHBM(Vl, ZSm, Clk, Sin)
 | (Vr, Sm, Sout) := EHBS(ZVm, Sr, Clk, ZDm)
 | ZVm := Vm $ init false
 | ZDm := Dm $ init 0
 | ZSm := Sm $ init false |)
where
     boolean Vm, Sm, ZVm, ZSm;
     integer Dm, ZDm;
     process EHBM = ( ? boolean Vl, Sr, Clk; integer Sin;
                      ! boolean Vr, Sl; integer Sout;)
     (% Relate input signals %
      | Vl ^= Sr ^= Clk ^= Sin
       % Compute Vr %
      | Vl1 := Vl or ZSl
      | Vr  := Vl1 when (not Clk) default ZVr
      | ZVr := Vr $ init false
      | Vr ^= Vl
      % Compute Sl %
      | Sr1 := ZVr and Sr
      | Sl  := Sr1 when Clk default ZSl
      | ZSl := Sl $ init false
      | Sl  ^= Sr
      % Date path Control %
      | Em  := Vl and (not Sl)
      % Data path %
      | DL  := Sin when Em when (not Clk) default DL $ init 0
      | Sout := DL
      | DL  ^= Clk |)
     where
          boolean Vl1, ZVr, ZSl, Sr1, Em; integer DL;
     end;
     process EHBS = ( ? boolean Vl, Sr, Clk; integer Sin;
                      ! boolean Vr, Sl; integer Sout;)
     % The description of EHBS is similar to EHBM except for the latches%
     (| ...., .... |)
end;

process Join = ( ? boolean Vl1, Vl2, Sr;
                 ! boolean Vr, Sl1, Sl2;)
(| Vr := Vl1 and Vl2
 | Vm := Vr and (not Sr)
 | Sl1 := Vl1 and (not Vm)
 | Sl2 := Vl2 and (not Vm) |)
where
     boolean Vm;
end

process Lazyfork = ( ? boolean Vl, Sr1, Sr2;
                     ! boolean Vr1, Vr2, Sl;)
(| Sl := Sr1 or Sr2
 | Vr1 := Vl and (not Sl)
 | Vr2 := Vl and (not Sl) |)

process Eagerfork = ( ? boolean Vl, Sr1, Sr2;
                      ! boolean Vr1, Vr2, Sl;)
(| V1 := Sr1 and ZV2
 | V5 := Sr2 and ZV4
 | Sl := V1 or V5
 | V3 := Vl and Sl
 | V2 := V1 or (not V3)
 | V4 := V5 or (not V3)
 | ZV2:= V2 $ init false
 | ZV4:= V4 $ init false
 | Vr1:= ZV2 and Vl
 | Vr2:= ZV4 and Vl |)
where
     boolean V1, V2, V3, V4, V5, ZV2, ZV4;
end
```

A.4 pSELF, including interleaving to synchronous protocol(int2syn), synchronous to interleaving protocols(syn2int) from [25].

```
process int2syn = ( ? boolean vl, sr, clk; integer dl;
```

```
                    ! boolean vr, sl; integer dr; )
( | (vi0, sl, di0) := pEHBL(vl, zsi0, clk, dl)
  | (vr,  si1, dr) := EHBM(zvi1, sr,  clk, zdi0)
  %compute vi1 %
  | vi1 := vi0
  | zvi1 := vi1 $ init false
  %compute si0 %
  | sl0 := vi1 and si1
  | si0 := sl0 when clk default zsi0
  | zsi0:= si0 $ init false
  | zdi0:= di0 $ init 0
  | )
where
    boolean vi0, vi1, zvi1, si0, zsi0, si1, sl0;
    integer di0, zdi0;
    process pEHBL = ( ? boolean Vl, Sr, Clk; integer Din;
                      ! boolean Vr, Sl; integer Dout; )
    % The detailed description is shown in section 4%
    (| ...., .... |)
    process EHBM = ( ? boolean Vl, Sr, Clk; integer Sin;
                     ! boolean Vr, Sl; integer Sout; )
    % The detailed description is shown in A.3 %
    (| ...., .... |)
\end

process syn2int = ( ? boolean vl, sr, clk; integer dl;
                    ! boolean vr, sl; integer dr; )
(| (vi0, sl, di0) := EHBM(vl, zsi0, clk, dl)
 | (vr,  si1, dr) := pEHBL(zvi1, sr, clk, zdi1)
 %compute si0 %
 | si0 := si1
 | zsi0 := si0 $ init false
 %compute vi1 %
 | vr0 := vi0 or si0
 | vi1 := vr0 when clk default zvi1
 | zvi1:= vi1 $ init false
 %data path %
 | en := vi0 and (not si1)
 | di1 := di0 when en when clk default zdi1
 | zdi1 := di1 $ init 0 |)
where
     boolean vi0, vi1, zvi1, si0, zsi0, si1, en, vr0;
     integer di0, di1, zdi1;
     process EHBM = ( ? boolean Vl, Sr, Clk; integer Sin;
                      ! boolean Vr, Sl; integer Sout;)
      % The detailed description is shown in A.3 %
     (| ...., .... |)
     process pEHBL = ( ? boolean Vl, Sr, Clk; integer Din;
                       ! boolean Vr, Sl; integer Dout; )
      % The detailed description is shown in section 4%
     (| ...., .... |)
end
```