CESNET technical report number 3/2004

Verification Results in Liberouter Project [1]

Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, and Pavel Šimeček

September 16, 2004

1 Abstract

This technical report presents current results of the formal verification of *VHDL* design of *Liberouter* and *Scampi* hardware accelerator card for packet routing, originating from the *Liberouter* project. We use the symbolic model checker **Cadence SMV** [SMV] to prove desired properties of separate units of the design. We have verified many properties of the number of units. Moreover, we have also gained precious experiences concerning the fight with the state explosion problem.

2 Introduction

The aim of the *Liberouter* project [LibWWW] is to design and develop the hardware accelerated router. The most important part of this project is development of the *Combo6* and *Scampi* hardware accelerator card [Nov04] allowing to route the most of traffic of *Gigabit Ethernet* in the hardware.

More introduction into the problematics of the verification in this project can be found in [VerifVHDLCombo6].

After finishing the technical report [VerifVHDLCombo6] we have found significant failures in our verification. Unfortunately, our previous definition of LATRS register entity was not correct and also further registers (e. g. DFFRS and DFFERS) were translated in the wrong way (by **vl2smv** utility).

For this reason we have created new models of these registers. We have also created new models of the registers in the cases with more than one clock signal, where it is necessary to have more detailed descriptions (see [FormalVHDL]).

After corrections of our translation process we have focused our effort to the real model checking of separate units of the design. We have gained precious

[1]This work is supported by the FP5 project No. IST-2001-32603, the CESNET project 02/2003, and the GACR grant No. 201/03/0509.

experience. Now we are able to formulate very interesting formulas and preconditions in *LTL*, furthermore we can manually model (on the very high level of abstraction) the behavior of certain parts of the design. And this way we are able to simulate the behavior of the units which create inputs (environment) of the verified component (see the verification of Statistic Unit).

In this report we would like to present results and experiences with the model checking of *VHDL* source in the *Liberouter* project in the form of description of verification of single units. Techniques used in verification slightly vary according to the particular verified unit.

3 Current state of translation process from VHDL to Cadence SMV

In the technical report [VerifVHDLCombo6] we have introduced our approach to the translation of *VHDL* to *Verilog* and then to the **Cadence SMV**. After releasing the report we have found significant errors in the definition of LATRS entity and we have also found out that further entities representing registers were wrong. The main reason of these mistakes is based on the fact that **vl2smv** is not able to correctly translate `always` blocks in a *Verilog* source code.

The first step which have to be done to verify source codes is downloading them from CVS[2]. You can download them directly via web interface of our CVS or you can use anonymous account and CVS client to download the entire CVS.

The process of translation is almost the same as in [VerifVHDLCombo6]. The difference is in definitions and counts of replaced entities. The process of translation and substitutions for entities was strongly automatized. We have created scripts called **vhd2v** (written in *Bash* scripting language) and **v2smv.pl** (written in *Perl*), which can be found in our CVS[3]. **vhd2v** calls the synthesizer on the design given in the command line with some optional parameters:

```
vhd2v [-vhd] DESIGN TO_OPTIMIZE [MOD ..] SRC
```

where `DESIGN` is the name of the synthesized design, `TO_OPTIMIZE` is the name of the part of the design to optimize (value `-` stands for the optimization of the entire design), `MOD` is the *VHDL* module used for synthesis and `SRC` is the *VHDL* module with top-level design.

v2smv.pl calls **vl2smv** and do few other transformations (e.g. substitutions for the modules). This is the syntax of its usage:

[2] *http://www.liberouter.org/cgi-bin2/cvsweb.cgi*

[3] *http://www.liberouter.org/cgi-bin2/cvsweb.cgi/liberouter/ver/scripts/*

```
v2smv.pl [-async/-sync] SRC
```

where `SRC` is the *Verilog* source (preferably the one produced by **vhd2v**) and parameters have the following meaning:

- `-async` means that for default substitutions we want to use definitions of modules determined for designs with multiple clocks placed in `$LIBEROUTER_PATH/ver/smv_codes/ASYNC`
- `-sync` means that for default substitutions we want to use definitions of modules determined for designs with single clock placed in `$LIBEROUTER_PATH/ver/smv_codes/SYNC`

`LIBEROUTER_PATH` is the environmental variable (it should represent the root of local copy of CVS).

3.1 Required Software

This is the list of software required for translation, verification, presentation of our results:

- **LeonardoSpectrum/Precision** for *VHDL* synthesis. (http://www.mentor.com/[4])
- **Cadence SMV** for model checking. (http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/[5])
- **saxon** for XSLT transformations of Verification reports. (http://saxon.sourceforge.net/[6])

Of course, these applications require some additional libraries and applications installed. For example **Cadence SMV** needs Tk Interface eXtension **TIX** and **saxon**, written in *Java*, requires Java Runtime Environment (available from web page of Sun).

3.2 Scampi project translation

This section is about the synthesis of *Scampi* project and its translation to **Verilog** and **Cadence SMV**.

Working directory for *Scampi* project is

[4] *http://www.mentor.com/*
[5] *http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/*
[6] *http://saxon.sourceforge.net/*

$LIBEROUTER_PATH/vhdl_design/combo6/projects/scampi_ph1

In this working directory the content of the file *Modules.tcl* determines which modules will be included in the translation. Translation of *top_level.vhd* to *top_level.v* and then to *top_level.smv* is performed using *gmake top_level.smv* or just *make top_level.smv*.

This translation is provided by **v2smv.pl** script and uses the following files:

$LIBEROUTER_PATH/vhdl_design/combo6/projects/scampi_ph1/Makefile
$LIBEROUTER_PATH/vhdl_design/base/Makefile.fpga.inc (included)

And Leonardo spectrum depends on the following files:

$LIBEROUTER_PATH/vhdl_design/combo6/projects/scampi_ph1/...
...Leonardo.ver.tcl
$LIBEROUTER_PATH/vhdl_design/combo6/projects/scampi_ph1/Modules.tcl
$LIBEROUTER_PATH/vhdl_design/base/Leonardo.inc.tcl (included)

The translation to **Cadence SMV** can produce empty modules. They can be for example:

DCM, RAM16X1D, RAM32X1D, RAM64X1D, RAMB16_S18_S18, RAMB16_S18_S36, RAMB16_S9_S18

Keeping these modules empty we abstract from their behavior and so all possible outputs of these modules are taken in consideration. The reason is that saving every state of memory module (e. g. RAM16X1D, RAM32X1D) to the computer memory is unfeasible.

4 Obtaining Assertions

One of the biggest problems is the way of obtaining properties which should be verified. There are three main sources of assertions:

- Assignment from *VHDL* designer: verbal description or assertions in comments in *VHDL* codes (for more details see [VC]).
- Documentation of components contains a large amount of specifications.
- Analysis of *VHDL* source codes and Applications notes from various sources (e. g. Xilinx).

5 Cadence SMV on Different Platforms

Cadence SMV model checker is freely available on the following platforms: *HP/UX*, *MIPS/IRIX*, *i386/Linux*, *Sparc/Solaris* and *Windows*. The latest version of **Cadence SMV** is only available on the *i386/Linux*, *Windows* and *Sparc/Solaris* platforms. Ports to the *HP/UX* and *MIPS/IRIX* platforms are no longer being maintained.

We want to choose the best platform for fast verification process. For this purpose the test verification was created and the time and space consumed by **Cadence SMV** with the test verification as input was analysed. The time and space consumption on different platforms is shown in logarithmic scale in Figure 1. We found difference between **Cadence SMV** versions on different platforms in memory consuming with the same input. On *Windows XP* the test verification never finished. It is unusual because on other platforms the space consumption was at most 151 MB and on *Windows XP* the 600 MB of free physical memory is not enough.

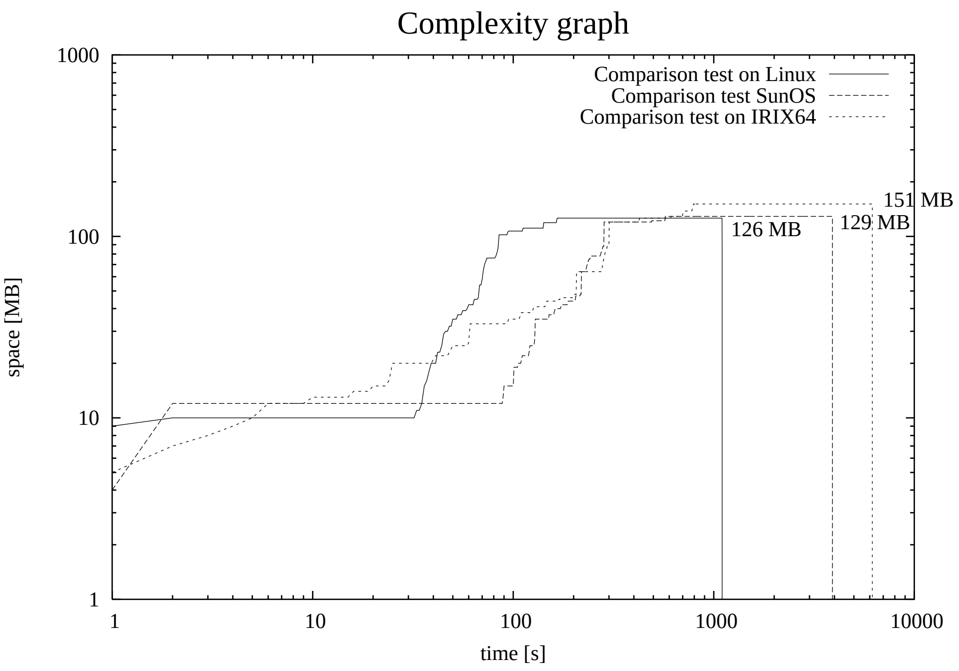

Figure 1: On *Windows XP* the test verification not finished (memory exceeded).

Running time of **Cadence SMV** with the testing input on several different machines is in Figure 2. These times are the average from two running times. On every row there is a different computer although it may seem that *Pentium 4 2,6 GHz, Linux* and *Pentium 4 3 GHz, Linux* are only identical copy.

For efficient usage of **Cadence SMV** we recommended computer with the lat-

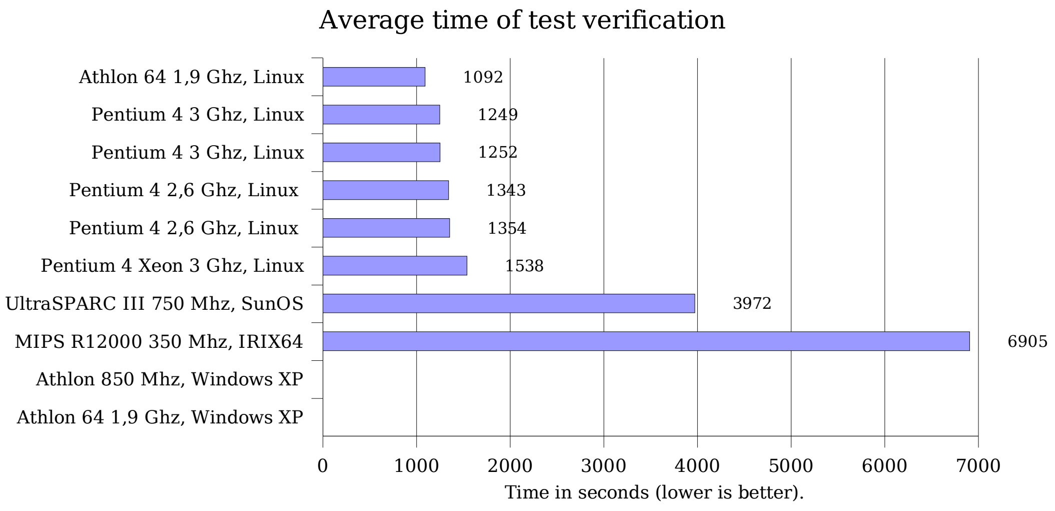

Figure 2: On *Windows XP* the test verification not finished (memory exceeded).

est Pentium or Athlon with some Linux distribution installed. The psychical memory size should be 2 GB for huge state spaces.

6 Components Under Verification

We will present selected components from *Liberouter* and *Scampi* designs which we currently verify according to several properties each time the new version of design appears in CVS. The basic outline of components is depicted in Figure 3. The highlighted components are currently under verification. Numbers under components are counts of unique verified properties. The similar properties are counted as one (see examples of verification to understand what ”similar” means).

Using the number of examples we would like to present the techniques used to achieve sufficiently efficient model checking in **Cadence SMV**.

6.1 Edit Engine

Basic documentation of *Edit Engine* (*EE*) is in the *Liberouter* CVS. [EditEngine] *EE* is a block for output packet processing. The record from *Priority Queues* is the first input information. It contains pointer to DRAM memory, where the packet data is stored, pointer to the parameters data structure, and the information on how to process the packet (so called *edit parameters*).

When *EE* gets record from *Priority Queues* it loads all parameters important for further processing. Then *EE* just loads packet data from DRAM memory

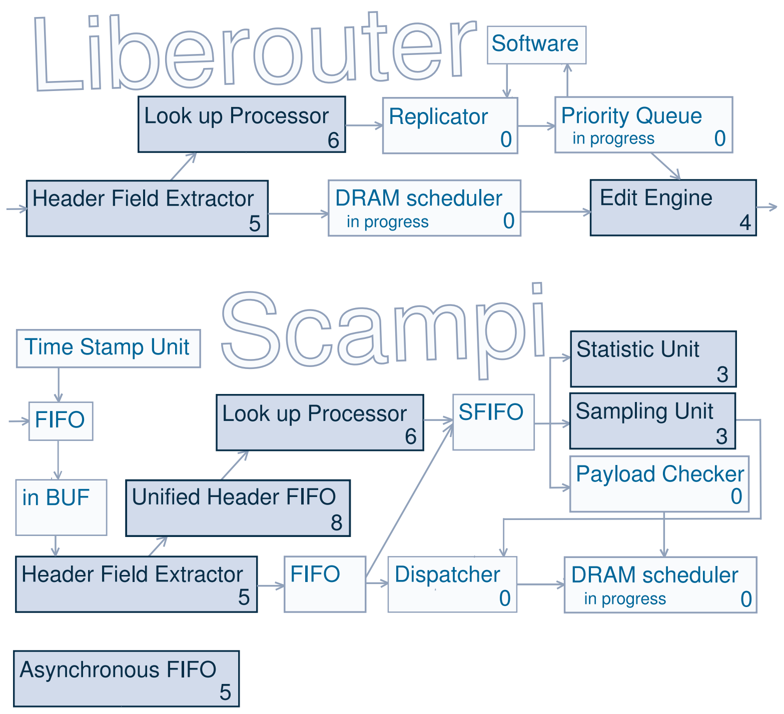

Figure 3: Parts of design under verification with number of properties verified:

and proceeds its processing. The example of *EE* work is changing L2 header, decrementation of Hop Limit entry (Time To Live in IPv4), processing Routing Header, control of ICMPv6, or Encapsulation/Decapsulation of packet in a tunnel.

1. G (! (plx_wr and plx_rd));
2. G ((!DRAM_COM_U.RESET and DRAM_COM_U.START) -> (X DRAM_WR_REQ));
3. G ((!DRAM_COM_U.RESET and DRAM_COM_U.START) -> (X DRAM_ACK));
4. G ((!DRAM_COM_U.RESET and DRAM_COM_U.START) -> (X DRAM_RD_REQ));
5. G ((!DRAM_COM_U.RESET and DRAM_COM_U.START) -> (X DRAM_DEC_REQ));
6. G ((!DRAM_COM_U.RESET and DRAM_COM_U.START) -> (X CTRL_U.DRAM_FINISH));
7. G (!DRAM_COM_U.RESET and !((DRAM_COM_U.INP_PTR + 1) = DRAM_COM_U.OUT_PTR));

The first assertion is simple mutual exclusion of signal `plx_wr` and `plx_rd`. This assertion is true and verification takes only 0.62 seconds.

All other assertions are the matter of *DRAM_COM_U* (DRAM Scheduler Communication Unit). It is assumed that at the beginning of the run the reset signal *DRAM_COM_U.RESET* resets the DRAM Scheduler Communication Unit and then the signal *DRAM_COM_U.RESET* is supposed to be disabled (to logical value 0).

The second assertion for *WR REQ FSM* (Write Request Finite State Machine) only ensures that after the *START* signal, which starts communication of DRAM Scheduler, the signal *DRAM_WR_REQ* for writing request will be enabled in one step.

Similarly for the other assertions. We considered these assertions as one unique property.

Some of them are even isomorphic and *Cadence SMV* primarily skips the model checking process when it finds a model checking problem which is isomorphic to some that has been already solved in the past. (To prevent this behaviour, use *-force* argument, or remove *.smv_history*.)

The fourth assertion is false. The reason is not the bug in design but a very imperfect specification. More preconditions would have to be added to make this assertion valid.

The last assertion is very important. DRAM scheduler coordination of data transfer into *PD* block using *CNT CE FSM* (Counter Clock Enable Finite State Machine). If the condition "(InputPointer + 1) = OutputPointer" is not satisfied, then the data that was still not processed will be rewritten. This situation should not happen.

No matter which abstraction we have used, the state space was always larger than 4 GB memory. The limit of 4 GB memory is given by our precompiled binary version of **Cadence SMV**.

6.2 Sampling Unit

VHDL structure of *Scampi* [VHDLScampi] project contains 16 *Sampling Units* (*SAU*).

Each can be configured to do:

- **probability sampling** – the packet is passed through the unit with the probability *1/n*.
- **deterministic sampling** – each *n*-th packet is passed through.

- **byte deterministic sampling** – the packet containing each *n*-th byte is passed through.

The packet could be processed simultaneously in more than one *SAU*. If the packet is not passed through any of these units, it is discarded. The required processing of the packet is coded in the control word. First half of 32-bit control word is *SAMASK* – sampling unit assignment. Each *SAU* has one bit in this bit array and '1' means that packed will be processed by corresponding *SAU*. The *SAMASK* field is masked by the result of *SAU* processing. The '1' is remained only if the packet is passed through the corresponding *SAU*.

Special configuration of *SAU* is the case of deterministic sampling with *n=1*. this case represents the situation, when all specified packets are required to pass to the application.

6.2.1 Assignment from hardware developers

Assertions obtained from hardware developer:

1. Mutual exclusion of control_flag and data_flag.
2. Mutual exclusion of mode_wr and init_wr_vec.
3. Mutual exclusion of signals CONTROL_A, CONTROL_B, CONTROL_C, and CONTROL_D.

In the first assertion both signals have been synthesized into `NOT_control_flag` and `NOT_data_flag` in component *SAU_INS_notri*. These signals are undefined when one of the resets `SAU_INS_notri.RESET` or `LRESET` is enabled. Hence the complete formula looks like:

```
G ( ! (
        (!SAU_INS_notri.RESET) and
        (!LRESET) and
        SAU_INS_notri.NOT_control_flag and
        SAU_INS_notri.NOT_data_flag
      )   );
```

The second assertion is a mutual exclusion of signal `mode_wr` and vector of 16 signals `init_wr_vec` in component `SAU_INS_notri` again.

Signal `mode_wr` is synthesized into signal `nx72` and vector `init_wr_vec` is synthesized into 16 signals `nx57, nx58, .., nx71, nx73`. No bit of vector `init_wr_vec` should be enabled, when `mode_wr` is enabled, hence the formula of mutual exclusion should be *G (! (SAU_INS_notri.nx72 and SAU_INS_notri.nx57 and SAU_INS_notri.nx58 and ... and SAU_INS_notri.nx73);*

The third assertion was not verified. It is due to the size of the state space generated by the resulting formula:

```
G ( ! ( SAU_INS_notri.sau_cores_0_U_SAU_CORE.CONTROL_A and
        SAU_INS_notri.sau_cores_0_U_SAU_CORE.CONTROL_B and
        SAU_INS_notri.sau_cores_0_U_SAU_CORE.CONTROL_C and
        SAU_INS_notri.sau_cores_0_U_SAU_CORE.CONTROL_D ) );
```

In each tested abstraction the state space was larger than 4 GB memory. Same as it was for the last assertion in *Edit Engine* component.

6.3 Header Field Extractor

Header Field Extractor (*HFE*) processes the instructions stored in block RAM. Its purpose is to perform packet analysis and store packets into DRAM. HFE is implemented as a full scope processor that is controlled by binary instructions. These instructions form a program that may be described by an assembler. If the parsing program has to be modified or extended in the future, only the parsing program will have to be updated without changing the hardware design.

6.3.1 Basic assignment

We have chosen the subcomponent *HFE_CORE*, because in due time the top-level design was in the continuous development. Below are assertions from developer and from our mind using developer's description of the design:

1. G F (loop_cntr_0_ or loop_cntr_1_ or loop_cntr_2_ or loop_cntr_3_ or loop_cntr_4_ or loop_cntr_5_ or loop_cntr_6_ or loop_cntr_7_);
2. G ((indd_inc -> (not indd_dec)) and (indd_dec -> (not indd_inc)))
3. G ((set_int -> (not clear_int)) and (clear_int -> (not set_int)))
4. ((not CLK) U G (not ((shortcut_en and sel_gpr) or (shortcut_en and sel_din) or (shortcut_en and sel_din_i) or (sel_gpr and sel_din) or (sel_gpr and sel_din_i) or (sel_din and sel_din_i))));
5. G ((ds_ren -> F (not ds_ren)) and ((not ds_ren) -> F ds_ren))

The first assertion is devised by the verifier. It can be rewritten into the assertions described in [VC] as:

loop_cntr is always alive

This assertion is true, when we specify some additional preconditions about the instruction input, as it will be shown in the next subsection.

The second and the third were assigned by the developer in this form (which is described in [VC]):

- *exclusion indd_inc, indd_dec*
- *exclusion set_int, clear_int*

Here we can see that for a developer it is very easy to understand what the exclusion of signals is.

Unfortunately it was not the case of the fifth assertion that was given by this assignment: *alive ds_ren*

`ds_ren` is the signal, which value determines whether it is possible to read data. Of course it is very natural to think about it in such a way that we would like to be always able to see the signal `ds_ren` enabled (equal to 1) in the future. But is it true to specify it in a way the designer did? No, because this requirement strongly depends on instructions which are processed by the processor (coming from processor input). But we do not have instructions for the processor encoded in the design. Therefore if it will always come *NOP*, then `ds_ren` signal will be 0 forever. And this is only one of many possible counterexamples falsifying activeness of `ds_ren`.

The fourth assertion was given by developer in this form:

exclusion shortcut_en, sel_gpr, sel_din, sel_din_i

It is valid, but it is also very special. This is the fragment of code of HFE_CORE (`hfe_core.vhd`, version `1.10`):

```
-- Data in bus address decoder
din_addr_decoder : process(clk,src_addr_i, dst_addr_i,
                           res_wen_reg, acc_wen_reg)
begin
   if clk'event and clk = '1' then
      sel_gpr <= '0';
      sel_din <= '0';
      sel_din_i <= '0';
      shortcut_en <= '0';
      if ( (src_addr_i = dst_addr_i) and
           (res_wen_reg = '1' ) ) or
         -- "shortcut" enable process
      . . .
```

Here you can see that signals `sel_gpr`, `sel_din`, `sel_din_i` and `shortcut_en` are set by the edge of clock. But they are not set by signal `reset`. Nevertheless it is not the fault, because it suffices to initialize the signal at the first tick of clock. Suppose that formula *Frm* expresses mutual exclusion of those four signals. Then the validity of *Frm* can be falsified by the run where all four signals are initialized to 1 (it is possible, because they are not reset at the beginning of the run). Therefore we have to create new formula, that exactly corresponds to the requirement of exclusivity after the first tick of clocks:

(not CLK) U Frm

This formula assumes that signal *CLK* representing clock is initialized to 0. This is given as a precondition:

not CLK

In this place it is also proper to list some basic preconditions we made about the system:

1. *RESET and X G not RESET*... It means that signal *RESET* is 1 only at the first state and then it is 0.
2. *G ((F CLK) and (F not CLK))*... It means that clock is always changed in future.
3. *assert (not CLK)*... It means that clock is initiated to zero (we do not have to use this precondition always)

As we have continued in trying to prove the assertions, we were continuously persuaded by model checker, that these assertions are false, if we use such a small count of preconditions. Therefore we had to enlarge the set of preconditions according to the counterexample we had got from the model checker for the given *LTL* formula.

6.3.2 More preconditions

In the case of *LTL* formulas number 2, 3 and 4 only first two preconditions from the given above suffice to show validity of assertions. In the case of the first *LTL* formula we get validity only if we add many additional (but valid) preconditions.

First we have to assign the precondition saying that instruction REPI comes infinitely times. In general it has not to be true, but we want to verify, that there is no obstacle to correctly increase and decrease loop_cntr in future. The obstacle should not be the absence of REPI instructions, because we know that REPI instructions are the only instructions working with loop_cntr and therefore these instructions are the only possible source of liveness of loop_cntr.

> G F ((INSTR_IN1[17..10] = 30 and not reg_pc_sel) or
> (INSTR_IN2[17..10] = 30 and reg_pc_sel))

The instruction REPI has one argument. If this argument was always zero, then the REPI instruction would not cause the increase of loop_cntr and the assertion would not be true. Therefore we have to add one more precondition:

> G ((INSTR_IN1[17..10] = 30 -> INSTR_IN1[7..0]>=2) and
> (INSTR_IN2[17..10] = 30 -> INSTR_IN2[7..0]>=2))

Unfortunately it does not suffice for validity of the first assertion. The reason is the asynchronism of the instruction input (signal buses INSTR_IN1 and INSTR_IN2). This piece of code is the only piece, that modifies loop_cntr:

```
if reset = '1' then
   loop_cntr        <= (others => '0');
   ...
elsif clk'event and clk = '1' then
   ...
   if lc_wen='1' then
      loop_cntr <= ALU_RES(7 downto 0) - 1;
   end if;
   if from_loop_ld = '1' then      -- write to loop_cntr
      loop_cntr <= instruction(7 downto 0)-1;
   elsif loop_in_progress = '1' and lc_gtz='1'
         and stop='0' then
      loop_cntr <= loop_cntr - 1;
   end if;
end if;
```

Signal from_loop_ld is dependent only on the instruction input (when REPI instruction occurs). Instruction input is not explicitly synchronized with clock and hence REPI can cause enabling of from_loop_ld for so short time that it is not noticed by the above code synchronized with clock.

This would cause invalidity of the first *LTL* formula. Therefore we have to add the set of preconditions, which set the synchronism of instruction input with clock:

G (X CLK -> CLK)->(INSTR_IN1[0]<->X INSTR_IN*i*[*bit*])

where *i* is the number of the instruction bus (it can be 1 or 2) and *bit* is the number of the single bit (it can be from 0 to 17). Therefore the set of preconditions setting synchronism of instruction bus with clock contains 36 preconditions.

All assertions except for the liveness of `ds_ren` has been found true. To be able to finish the model checking of single formulas we had to make the abstraction. We have deleted the content of *HFE_ALU*. This abstraction has broken the relation between the present content of the most of registers and the far history, because *HFE_ALU* lies on the circular way of the data through the registers of *HFE*.

6.4 Unified Header FIFO

Unified Header FIFO (*UHFIFO*) is the data queue between *HFE* and *LUP* (Look up Processor). *UHFIFO* was verified continuously during the development. The verifications from year 2003 are not fully reliable because there were some mistakes in the verification procedures and verifications were not repeated. The more reliable and interesting verification was the verification of *UHFIFO* exported on 22/02/2004 and 18/04/2004.

6.4.1 Export from 22/02/2004

This export corresponds to the version 1.1 of `uh_fifo.vhd` in the new CVS repository. It is the old version of *UHFIFO* that had to be redesigned because of problems connected with two clocks in one design. But from the simple point of view, when we work on the level of abstraction where these problems do not occur, this model works well.

<h4>Basic assignment</h4>

We have verified this set of assertions:

1. *alive ready*
2. *globally if (ready(conv_integer(unsigned(lup_block)))) then ((hfe_block <> lup_block) or (not write_i))*
3. *(hfe_block does not change after HFE_RDY=1 until hfe_is_producing=1) or (hfe_block does not change forever after HFE_RDY=1)*
4. *(uh_valid=0 after HFE_RDY=1 until HFE_VALID=1) or (uh_valid=0 forever after HFE_RDY=1)*
5. *HFE_RDY=1 infinitely-times*
6. *hfe_is_producing=1 infinitely-times*

First two assertions are given by the developer. The others are devised by the verifier. Rewriting to *LTL* was not so easy as it could seem. The first assertion is rewritten to the set of 32 *LTL* formulas:

- 16 formulas of the form *not (F G ready_i_)* . . . it means that *i*-th bit of `ready` does not stay constant 1 in the future.
- 16 formulas of the form *not (F G not ready_i_)* . . . it means that *i*-th bit of `ready` does not stay constant 0 in the future.

The transcription to the 32 formulas instead of the single one is necessary mainly for the memory complexity reasons.

For the purpose of formal verification the second assertion is simplified so that `lup_block` is substituted by 0 (therefore we verify the assertion only for one of sixteen possible values of `lup_block`). It is probably not wrong because cases of the verification for different values of `lup_block` look very similar. The simplified assertion is this one:

> *globally if ready(0) and LUP_ADDR[8..5]=0 then (hfe_block or not write_i)*

It can be easily rewritten to *LTL* (some variables are substituted by their real counterparts in the *SMV* code):

> *G ((ready_0_ and LUP_ADDR[8..5]=0) -> (hfe_block_0_ or hfe_block_1_ or hfe_block_2_ or hfe_block_3_ or not (HFE_WEN and hfe_allocated)))*

Third assertion is rewritten to *LTL* using 8 formulas:

- 4 formulas of the form

  G ((HFE_RDY and hfe_block_i_)->
  ((hfe_block_i_ U hfe_is_producing) or *(G hfe_block_i_)))*

  where *i* is the number of bit (from 0 to 3)

- 4 formulas of the form

  G ((HFE_RDY and not hfe_block_i_)->
  (((not hfe_block_i_) U hfe_is_producing) or *(G not hfe_block_i_)))*

  where *i* is the number of bit (from 0 to 3)

The rest of assertions is rewritten to *LTL* in this way:

- *G (HFE_RDY -> ((G not uh_valid) or ((not X uh_valid) U hfe_is_producing)))*
- *G F HFE_RDY*
- *G F hfe_is_producing*

The assertions 3 and 4 need only the precondition restricting the signal `RESET` to be valid:

> *RESET and X G (not RESET)*. . . This means that signal `RESET` is 1 only at the first state and then it is 0.

More preconditions

The rest of assertions needs many more preconditions. These preconditions model the behavior of inputs (sometimes with regard to outputs). There is the list of preconditions used in verification of the rest of assertions.

- *G ((F HFE_CLK) and (F not HFE_CLK))*. . . `HFE_CLK=1` infinitely-times and also `HFE_CLK=0` infinitely-times
- *G ((F LUP_CLK) and (F not LUP_CLK))*. . . `LUP_CLK=1` infinitely-times and also `LUP_CLK=0` infinitely-times
- *G F (HFE_REQ)*. . . `HFE_REQ=1` infinitely-times
- *G F (LUP_SR_CLEAN)*. . . `LUP_SR_CLEAN=1` infinitely-times
- *G (HFE_RDY -> F not HFE_REQ)*. . . After enabling of `HFE_READY`, `HFE_REQ` will be disabled
- *G (LUP_SR_VALID -> F LUP_SR_CLEAN)*. . . There comes `LUP_SR_CLEAN=1` after `LUP_SR_VALID=1`
- This formula is so big that it is better to write it structured:

```
G (
   ( (LUP_ADDR[5] xor (X LUP_ADDR[5])) or
     (LUP_ADDR[6] xor (X LUP_ADDR[6])) or
     (LUP_ADDR[7] xor (X LUP_ADDR[7])) or
     (LUP_ADDR[8] xor (X LUP_ADDR[8]))  ) ->
   (
    (X not LUP_SR_CLEAN) and
    X (
       ( not ( (LUP_ADDR[5] xor (X LUP_ADDR[5])) or
               (LUP_ADDR[6] xor (X LUP_ADDR[6])) or
               (LUP_ADDR[7] xor (X LUP_ADDR[7])) or
               (LUP_ADDR[8] xor (X LUP_ADDR[8]))  ))
       U LUP_SR_CLEAN
      )
   )
  );
```

After change of LUP_ADDR there is no further change of LUP_ADDR until LUP_SR_CLEAN=1. The only magic about this formula are the positions of *X* operators.

- *G F ((LUP_ADDR[5] xor (X LUP_ADDR[5])) or (LUP_ADDR[6] xor (X LUP_ADDR[6])) or (LUP_ADDR[7] xor (X LUP_ADDR[7])) or (LUP_ADDR[8] xor (X LUP_ADDR[8])))* ... `LUP_ADDR[8..5]` is not constant
- *LUP_ADDR[8..5]=0* ... `LUP_ADDR[8..5]` is initialized to zero
- *hfe_block_0_=0 and hfe_block_1_=0 and hfe_block_2_=0 and hfe_block_3_=0* ... `hfe_block` is initialized to zero
- *G ((LUP_ADDR[8..5]*=x*) -> X((LUP_ADDR[8..5]*=x*) or (LUP_ADDR[8..5]*=y*)))*, where *x* is a value from 0 to 15 and *y*=*x*`+1 mod 16`... `LUP_ADDR[8..5]` can only be the constant or it can increase (except for the overflow, of course)
- *G (((X HFE_CLK) -> HFE_CLK) -> ((HFE_REQ<->X HFE_REQ)))*... `HFE_REQ` is synchronized with clock of *HFE*
- *G (((X LUP_CLK) -> LUP_CLK) -> ((LUP_SR_CLEAN<->X LUP_SR_CLEAN)))* ... `LUP_SR_CLEAN` is synchronized with clock of *LUP*

6.4.2 Export from 18/04/2004

This is absolutely new implementation of *UHFIFO* implemented using component *DP_REGFLAGS = Dual Ported Register of Flags*. We want to verify the same properties as in the case of the previous implementation. Fortunately inner signals used in formulas remained unchanged. Therefore it is possible to simply run the verification with the same formulas and preconditions as in the previous implementation.

But if you simply repeat the verification in the same way as above it will not finish because of high memory complexity. There is not any other way than that of making changes inside of *VHDL* code. We have decided to shrink the length of FIFO to only 4 items. Of course, this modification may cause that the formula which is valid for the modified code would not be valid for the original one, but it is the matter of good reasoning of the verifier to choose formulas, which are valid equivalently in both designs. Formal proof of this equality would be hard and a lot of time consuming.

There are *LTL* formulas for the shrunk *UHFIFO*:

1. - 4 formulas of the form *not (F G ready_i_)* . . . it means that *i*-th bit of `ready` does not stay constant 1 in the future.
   - 4 formulas of the form *not (F G not ready_i_)* . . . it means that *i*-th bit of `ready` does not stay constant 0 in the future.
2. *G ((ready_0_ and (LUP_ADDR[8..7]=0)) -> (hfe_block_0_ or hfe_block_1_ or not (HFE_WEN and hfe_allocated)))*
3. - 2 formulas of the form *G ((HFE_RDY and hfe_block_i_)->((hfe_block_i_ U hfe_is_producing) or (G hfe_block_i_)))* where *i* is the number of bit (from 0 to 1)
   - 2 formulas of the form *G ((HFE_RDY and not hfe_block_i_)->(((not hfe_block_i_) U hfe_is_producing) or (G not hfe_block_i_)))* where *i* is the number of bit (from 0 to 1)
4. *G (HFE_RDY -> ((G not uh_valid) | ((not X uh_valid) U hfe_is_producing)))*
5. *G F HFE_RDY*
6. *G F hfe_is_producing*

There are preconditions for the shrunk *UHFIFO*:

- *RESET and X G (not RESET)*
- *G ((F HFE_CLK) and (F not HFE_CLK))*
- *G ((F LUP_CLK) and (F not LUP_CLK))*
- *G F (HFE_REQ)*
- *G F (LUP_SR_CLEAN)*
- *G (HFE_RDY -> F not HFE_REQ)*
- *G (LUP_SR_VALID -> F LUP_SR_CLEAN)*

-
  ```
  G (
      ( (LUP_ADDR[7] xor (X LUP_ADDR[7])) or
        (LUP_ADDR[8] xor (X LUP_ADDR[8]))  ) ->
      (
       (X not LUP_SR_CLEAN) and
       X (
          ( not ( (LUP_ADDR[7] xor (X LUP_ADDR[7])) or
                  (LUP_ADDR[8] xor (X LUP_ADDR[8]))  ))
          U LUP_SR_CLEAN
         )
      )
     );
  ```
- *G F ((LUP_ADDR[7] xor (X LUP_ADDR[7])) or (LUP_ADDR[8] xor (X LUP_ADDR[8])))*
- *LUP_ADDR[8..7]=0*
- *hfe_block_0_=0 and hfe_block_1_=0 and hfe_block_2_=0 and hfe_block_3_=0*
- *G ((LUP_ADDR[8..7]=*x*) -> X((LUP_ADDR[8..7]=*x*) or (LUP_ADDR[8..7]=*y*)))*, where *x* is a value from 0 to 3 and *y*=*x*+1 `mod 4`
- *G (((X HFE_CLK) -> HFE_CLK) -> ((HFE_REQ<->X HFE_REQ)))*. . . `HFE_REQ` is synchrinized with clock of *HFE*
- *G (((X LUP_CLK) -> LUP_CLK) -> ((LUP_SR_CLEAN<->X LUP_SR_CLEAN)))* . . . `LUP_SR_CLEAN` is synchronized with clock of *LUP*

The *VHDL* code of the shrunk *UHFIFO* is placed in the **Appendix A** The model checking for the shrunk *UHFIFO* runs relatively fast.

By the process of verification there have not been found any mistakes in the design of *UHFIFO*, but surprisingly the mistakes have been found in *LUP*, which was under development at that time. *LUP* did not control the part of signals leading to *UHFIFO*. This mistake was found during the search for the possible preconditions for *UHFIFO*. It was not found by the model checker, but by verificators during studying of the source code of *LUP*.

6.5 Statistic Unit

Currently there have been verified only properties of the part *STU_LENGTH* of the *Statistic Unit*, because the part for the processing of time stamps has not been implemented yet. The verification of *STU_LENGTH* has been very specific, because we have focused our attention to the mechanism of addressing in this unit. *LBCONN_MEM* is accessing the same memory at the same time as the finite state machine inside *STU_LENGTH*. Therefore, it is necessary that the highest bit of the address used by *LBCONN_MEM* and the finite state machine of *STU_LENGTH* has to be different. The highest bit of the address is called a bank. *LBCONN_MEM* sets the bank used by the finite state machine of *STU_LENGTH* in the following way:

1. *LBCONN_MEM* sends the special address value via the address bus.
2. Concurrently with the special address value *LBCONN_MEM* sends a new flag register value and the new bank value is the part of the flag register.

It is wanted to verify that the bank stored in *STU_LENGTH* is always the inverse of the bank used by *LBCONN_MEM* for an access to the memory.

6.5.1 The different approaches to the verification

The first approach to the verification of this property was very simple. We took the component *STU_LENGTH* together with the component *LBCONN_MEM* and translated the code to *SMV*. But the verification of even the simple properties did not finish in the time limit. The reason was the complex structure of *LBCONN_MEM*.

This situation is now solved by the abstraction. The component *LBCONN_MEM* has been be deleted. This simple abstraction would cause invalidity of the property. First we were trying to put additional preconditions on the signals leading from *LBCONN_MEM*, but the set of preconditions was too large, complex and finally even incorrect. We were not capable to set the sufficiently detailed behavior of *LBCONN_MEM* only using the preconditions written in *LTL*. Therefore we have created the abstract model of *LBCONN_MEM* (it has not been proved that it is correct abstract model, but we believe it) - the source code of this abstract model can be found in **Appendix B**.

Furthermore it has been given by developer that the clock of *LBCONN_MEM* can be at most twice as fast as the clock of *STU_LENGTH* . This precondition has played the important role in the verification. Again it seems too hard to write the formula expressing such a property of clock. Therefore the system of clocks has been modeled in *SMV*:

```
init(LBCLK) := 0;
init(lbclk_count) := 0;
init(CLK) := 0;

if ((LBCLK=0) && (lbclk_count=2))
```

```
  --no lbclk edge when the count of lbclk is high
  aux_lbclk := 0;
else
  aux_lbclk := {0,1};

next(LBCLK) := aux_lbclk;

if (CLK=0 & lbclk_count=0)
  --no clk edge when the count of lbclk is low
  aux_clk := 0;
else
  aux_clk := {0,1};

next(CLK) := aux_clk;

--computation of lbclk_count:
if (CLK=0 & aux_clk=1)
 {
  if (LBCLK=0 & aux_lbclk=1)
    next(lbclk_count) := lbclk_count;
  else
    next(lbclk_count) := 0;
 }
else
 {
  if (LBCLK=0 & aux_lbclk)
    next(lbclk_count) := lbclk_count + 1;
  else
    next(lbclk_count) := lbclk_count;
 }
```

The developer also states that READY is enabled at most each 5 ticks of clock. It has been modeled in *SMV* this way:

```
init(READY) := 0;
init(delay_ready) := 0;

delay_ready_decrease := {0,1};
if (CLK=0 & aux_clk=1)
 {
  if (delay_ready~=0)
   {
    if (delay_ready_decrease)
```

```
      { next(delay_ready) := delay_ready - 1; }
     else { next(delay_ready) := delay_ready; }
     next(READY) := 0;        --while waiting producing 0
    }
   else
    {
     --after at most 5 ticks it decides to produce 1:
     next(delay_ready) := 5;
     next(READY) := 1;
    }
  }
else { next(READY) := READY; }
```

The *LTL* property, which has been verified in the system with the above modifications, is the following:

> *G (not en_reg_5 and (reg_phase_1_ or reg_phase_2_ or reg_phase_3_ or reg_phase_4_) ->*
> *(not reg_addr_out_8_ <-> br_addr_lb_9_))*

To prove it we only need these two basic preconditions:

- *RESET and (X G not RESET)*... This means that signal *RESET* is 1 only at the first state and then it is 0.
- *(G F CLK) and (G F not CLK)*... This means that clock is always changed in future.

Take notice that the source code of abstract model of *LBCONN_MEM* in **Appendix B** contains the variable `delay`, which sets the speed of abstract model of *LBCONN_MEM*. The `delay` is set to 8 ticks of `LB_CLK`. If it was less than 8 ticks, the design would be incorrect, because *LBCONN_MEM* would be able to read from the address with new bank earlier than *STU_LENGTH* would be able to finish the writing operation to the address with the old bank. Therefore the conflict would be possible. In fact this way it has been shown that *LBCONN_MEM* may change its outputs at most each 8 ticks of `LB_CLK`. There arises the new task for a verification of *LBCONN_MEM*.

There has been also verified the property of the phase register, which says, that any of bits of the phase register does not remain a zero forever. Expressed in *LTL* as:

> *not F G reg_phase*_i_ *= 0* ... where *i* is the index of the bit in the phase register.

There is only needed one more precondition (together with two above):

G F READY . . . `READY` is infinitely-times 1

7 Conclusion

In this paper it has been shown that in the complex formal verification we have to use techniques which are not absolutely formal. For example in the verification of *UHFIFO* a shrunk model was used without any proof of equality of *LTL* formulas in the shrunk and in the original model. This is because of the lack of time for the verification. If the verification was finished one year after the design it would be useless.

It has been also shown that the fight with the state explosion problem may be not always successful. When it is successful, it is often in spite of the price of large changes in the model. It was found out that the space complexity is not directly dependent on the length of source code or the count of registers. By the virtue of *cone of influence* it more depends on the complexity of an *LTL* formula and the variables which appear in the formula (including the variables which affect the value of these variables).

On one hand we have gained the pessimistic results, because we had to do many changes in the design and write large amounts of preconditions ourselves. It took a lot of time and human resources. On the other hand we have shown that even using such simple freeware verification tool as **Cadence SMV** it is possible to verify the large amount of complex properties of hardware components.

8 Appendix A – the source code of shrunk UHFIFO

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library unisim;
use unisim.all;

entity UH_FIFO is
   port(
      -- HFE interface
      HFE_DOUT   : in std_logic_vector(15 downto 0);
      HFE_ADDR   : in std_logic_vector(5 downto 0);
      HFE_RDY    : out std_logic;   -- Control signals
      HFE_REQ    : in std_logic;
      HFE_WEN    : in std_logic;
      HFE_CLK    : in std_logic;

      -- LUP interface
        -- whether cell contains unfied header:
      LUP_SR_VALID    : out std_logic;
         -- clean addressed cell:
      LUP_SR_CLEAN    : in  std_logic;
      LUP_DATA        : out std_logic_vector(31 downto 0);
      LUP_ADDR        : in  std_logic_vector(8 downto 0);
      LUP_CLK         : in std_logic;

      RESET      : in std_logic
   );
end UH_FIFO;

architecture behavioral of UH_FIFO is

signal hfe_block        : std_logic_vector(1 downto 0);
signal hfe_block_aux    : std_logic_vector(1 downto 0);
signal ready            : std_logic_vector(3 downto 0);
signal reg_ready        : std_logic_vector(3 downto 0);
signal hfe_allocated    : std_logic;
signal addra_i          : std_logic_vector(9 downto 0);
signal write_i          : std_logic;
signal uh_valid         : std_logic;
signal hfe_is_producing : std_logic;
signal hfe_rdy_i        : std_logic;

signal lup_block        : std_logic_vector(1 downto 0);

signal gnd_bus          : std_logic_vector(31 downto 0);
signal gnd              : std_logic;
signal pwr              : std_logic;

component RAMB16_S18_S36
   port (
      ADDRA: IN std_logic_vector(9 downto 0);
      ADDRB: IN std_logic_vector(8 downto 0);
      DIA:   IN std_logic_vector(15 downto 0);
      DIB:   IN std_logic_vector(31 downto 0);
      DIPA:  IN std_logic_vector(1 downto 0);
      DIPB:  IN std_logic_vector(3 downto 0);
      WEA:   IN std_logic;
      WEB:   IN std_logic;
      CLKA:  IN std_logic;
      CLKB:  IN std_logic;
      SSRA:  IN std_logic;
      SSRB:  IN std_logic;
      ENA:   IN std_logic;
      ENB:   IN std_logic;
      DOA:   OUT std_logic_vector(15 downto 0);
      DOB:   OUT std_logic_vector(31 downto 0);
      DOPA:  OUT std_logic_vector(1 downto 0);
      DOPB:  OUT std_logic_vector(3 downto 0));
END component;

component dp_regflags
   generic(
      EXADDR   : integer
   );
    port(
      RESET    : in  std_logic;

      -- SET part
      CLK_SET  : in  std_logic;
      SET      : in  std_logic;
      ADDR_SET : in  std_logic_vector(EXADDR-1 downto 0);
      DO_SET   : out std_logic;

      -- CLR part
      CLK_CLR  : in  std_logic;
      CLR      : in  std_logic;
      ADDR_CLR : in  std_logic_vector(EXADDR-1 downto 0);
      DO_CLR   : out std_logic;

      DO_ALL   : out std_logic_vector((2**EXADDR)-1 downto 0)
    );
end component;


begin


gnd_bus <= ”00000000000000000000000000000000”;
```

```
gnd <= '0';
pwr <= '1';

lup_block <= LUP_ADDR(8 downto 7);  --PP

addra_i <= hfe_block & "00" & HFE_ADDR; --PP
write_i <= HFE_WEN and hfe_allocated;

HFE_RDY <= hfe_rdy_i;

hfe_communication:process(HFE_CLK, RESET)
begin
   if reset = '1' then
      hfe_rdy_i        <= '0';
      hfe_allocated    <= '0';
      hfe_is_producing <= '0';
      hfe_block        <= "00"; --PP
      hfe_block_aux    <= "00"; --PP
      uh_valid         <= '0';
   elsif HFE_CLK'event and HFE_CLK = '1' then
      uh_valid  <= '0';
      hfe_rdy_i <= '0';
      if HFE_REQ='1' then
         if hfe_is_producing='1' then
            uh_valid         <= '1';
            hfe_block        <= hfe_block+1;
            hfe_is_producing <= '0';
            hfe_allocated    <= '0';
         elsif reg_ready(conv_integer(unsigned(
                                 hfe_block)))='0' then
            hfe_rdy_i     <= '1';
            hfe_allocated <= '1';
            hfe_block_aux <= hfe_block;
         end if;
      elsif hfe_allocated='1' then
         hfe_is_producing<='1';
      end if;
   end if;
end process;

reg_ready_proc: process(HFE_CLK,RESET)
begin
   if reset='1' then
```

```
      reg_ready <= (others => '0');
   elsif HFE_CLK'event and HFE_CLK='1' then
      reg_ready <= ready;
   end if;
end process;

block_ram: RAMB16_S18_S36 port map(
      ADDRA => addra_i,
      ADDRB => LUP_ADDR,
      DIA   => HFE_DOUT,
      DIB   => gnd_bus,
      DIPA  => gnd_bus(1 downto 0),
      DIPB  => gnd_bus(3 downto 0),
      WEA   => write_i,
      WEB   => gnd,
      CLKA  => HFE_CLK,
      CLKB  => LUP_CLK,
      SSRA  => gnd,
      SSRB  => gnd,
      ENA   => pwr,
      ENB   => pwr,
      DOA   => open,
      DOB   => LUP_DATA,
      DOPA  => open,
      DOPB  => open
);

flags: dp_regflags
generic map(
   EXADDR => 2 --PP
)
port map(
   RESET       => RESET,

   -- SET part
   CLK_SET     => HFE_CLK,
   SET         => uh_valid,
   ADDR_SET    => hfe_block_aux,
   DO_SET      => open,

   -- CLR part
   CLK_CLR     => LUP_CLK,
   CLR         => lup_sr_clean,
   ADDR_CLR    => lup_block,
   DO_CLR      => LUP_SR_VALID,

   DO_ALL      => ready
);


end behavioral;
```

9 Appendix B – the source code of abstract model of LBCONN_MEM

```
module \lbconn_mem_524288_13_notri (
\ADDR ,\nx1653 ,\LBLAST ,\nx1797 ,\nx1603 ,\NOT_lb_oen ,
\LBFRAME_modgen_select_12 , \data_in_reg_0_ ,\DATA_IN ,
\data_in_reg_1_ ,\data_in_reg_2_ ,\data_in_reg_3_ ,
\data_in_reg_4_ ,\data_in_reg_5_ ,\data_in_reg_6_ ,
\data_in_reg_7_ ,\data_in_reg_8_ ,\data_in_reg_9_ ,
\data_in_reg_10_ ,\data_in_reg_11_ ,\data_in_reg_12_ ,
\data_in_reg_13_ ,\data_in_reg_14_ ,\data_in_reg_15_ ,
\DATA_OUT ,\LBAD ,\RW ,\LBAS_modgen_select_11 ,\LBRW ,
\EN ,\LBCLK , LBCLK_NEXT ,\RESET ,\ARDY_modgen_select_6 ,
\SEL ,\DRDY ){
output \ADDR  : array 12..0 of boolean resolve;
\nx1653  : boolean resolve;
input \LBLAST  : boolean resolve;
\nx1797  : boolean resolve;
\nx1603  : boolean resolve;
\NOT_lb_oen  : boolean resolve;
input \LBFRAME_modgen_select_12:array 9..9 of boolean resolve;
\data_in_reg_0_  : boolean resolve;
input \DATA_IN  : array 15..0 of boolean resolve;
\data_in_reg_1_  : boolean resolve;
\data_in_reg_2_  : boolean resolve;
\data_in_reg_3_  : boolean resolve;
\data_in_reg_4_  : boolean resolve;
\data_in_reg_5_  : boolean resolve;
\data_in_reg_6_  : boolean resolve;
\data_in_reg_7_  : boolean resolve;
\data_in_reg_8_  : boolean resolve;
```

```
\data_in_reg_9_  : boolean resolve;
\data_in_reg_10_  : boolean resolve;
\data_in_reg_11_  : boolean resolve;
\data_in_reg_12_  : boolean resolve;
\data_in_reg_13_  : boolean resolve;
\data_in_reg_14_  : boolean resolve;
\data_in_reg_15_  : boolean resolve;
output \DATA_OUT  : array 15..0 of boolean resolve;
input \LBAD  : array 15..0 of boolean resolve;
\RW  : boolean resolve;
input \LBAS_modgen_select_11:array 9..9 of boolean resolve;
input \LBRW  : boolean resolve;
\EN  : boolean resolve;
input \LBCLK  : boolean resolve;
input LBCLK_NEXT : boolean resolve;
input \RESET  : boolean resolve;
input \ARDY_modgen_select_6:array 9..9 of boolean resolve;
\SEL  : boolean resolve;
input \DRDY  : boolean resolve;

gener_data: array 8..0 of boolean resolve;
bank: boolean resolve;
aux_bank: boolean resolve;
action: array 2..0 of boolean resolve;
aux_action: array 2..0 of boolean resolve;
delay: array 4..0 of boolean resolve;

 init(bank) := 1;
 init(action) := 0;
 init(gener_data) := 0;
 init(DATA_OUT) := 1;
 init(RW) := 0;
 init(delay) := 0;

 if (LBCLK=0 && LBCLK_NEXT=1)
  {
   if (delay~=0)
    {
     next(delay) := delay - 1;
     next(DATA_OUT) := DATA_OUT;
     next(action) := action;
     next(gener_data) := gener_data;
     next(bank) := bank;
     next(RW) := RW;
    }
   else
    {
     next(delay) := 5;
     next(DATA_OUT[0]) := {0,1};
     next(DATA_OUT[2])  := {0,1};
     next(DATA_OUT[3])  := {0,1};
     next(DATA_OUT[4])  := {0,1};
     next(DATA_OUT[5])  := {0,1};
     next(DATA_OUT[6])  := {0,1};
     next(DATA_OUT[7])  := {0,1};
     next(DATA_OUT[8])  := {0,1};
     next(DATA_OUT[9])  := {0,1};
     next(DATA_OUT[10]) := {0,1};
     next(DATA_OUT[11]) := {0,1};
     next(DATA_OUT[12]) := {0,1};
     next(DATA_OUT[13]) := {0,1};
     next(DATA_OUT[14]) := {0,1};
     next(DATA_OUT[15]) := {0,1};
     aux_action[0] := {0,1};
     aux_action[1] := {0,1};
     aux_action[2] := {0,1};
     if (aux_action > 5) { next(action) := 0; }
     else { next(action) := aux_action; }
     if (aux_action = 5)
      {
       next(gener_data) := 0;
       aux_bank := {0,1};
       next(bank) := aux_bank;
       next(DATA_OUT[1]) := !aux_bank;
       next(RW) := 1;
      }
     else
      {
       next(gener_data[0])  := {0,1};
       next(gener_data[1])  := {0,1};
       next(gener_data[2])  := {0,1};
       next(gener_data[3])  := {0,1};
       next(gener_data[4])  := {0,1};
       next(gener_data[5])  := {0,1};
       next(gener_data[6])  := {0,1};
       next(gener_data[7])  := {0,1};
```

```
        next(gener_data[8])  := {0,1};
        next(DATA_OUT[1])  := {0,1};
        next(bank) := bank;
        next(RW) := 0;
       }
     }
   }
  else
   {
    next(DATA_OUT) := DATA_OUT;
    next(bank) := bank;
    next(RW) := RW;
    next(gener_data) := gener_data;
    next(action) := action;
   }

ADDR := action::(bank & !RW)::gener_data;
EN := 1;
SEL := 1;

}
```

References

[FormalVHDL] J. Holeček, T. Kratochvíla, V. Řehák, D. Šafránek, and P. Šimeček: *How to Formalize a FPGA Hardware Design.*
Technical report number 4/2004, CESNET, 2004.

[VerifVHDLCombo6] T. Kratochvíla, V. Řehák, P. Šimeček: *Verification of COMBO6 VHDL Design.*
Technical report number 17/2003, CESNET, 2003.

[VHDLScampi] Vladimir Smotlacha: *Design of the VHDL structure of SCAMPI adapter.*
`http://www.liberouter.org/cgi-bin2/cvsweb.cgi/liberouter/vhdl_design/projects/scampi_ph1/doc/VHDL_structure.ps`

[EditEngine] Tomáš Martínek: *Basic documentation of Edit Engine.*
`http://www.liberouter.org/cgi-bin2/cvsweb.cgi/liberouter/vhdl_design/units/ee/doc/`

[LibWWW] Liberouter: *Liberouter Project WWW Pages.*
`http://www.liberouter.org/`

[Nov04] Novotný J., Fučík O., Antoš D.: *Project of IPv6 Router with FPGA Hardware Accelerator*. In Proceedings of Field-Programmable Logic and Applications: 13th International Conference FPL 2003, page 964-967, Springer Verlag, 2003. ISBN 3-540-40822-3.

[SMV] Cadence SMV: *Cadence SMV WWW Pages*. `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`

[VC] Tomáš Kratochvíla: *Verification cookbook (Liberouter policy WWW Pages)*. `http://www.liberouter.org/policy.html`

[VHDL] Ashenden Peter J.: *The VHDL Cookbook*. `http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf`