SoC Trust Validation Using Assertion-Based Security Monitors

Khitam Alatoun, Bharath Shankaranarayanan, Shanmukha Murali Achyutha and Ranga Vemuri
Digital Design Environments Laboratory, University of Cincinnati, Cincinnati, OH USA
alatoukm@mail.uc.edu, shankabh@mail.uc.edu, achyutsi@mail.uc.edu and ranga.vemuri@uc.edu

Abstract—Modern SoC applications include a variety of sensitive modules in which data must be protected against malicious access. Security vulnerabilities, when exercised during the SoC operation, lead to denial of service or disclosure of protected data. Hence, it is essential to undertake security validation before and after SoC fabrication and make provisions for continuous security assessment during operation. This paper presents a methodology for optimized post-deployment monitoring of SoC’s security properties by migrating pre-fab design security assertions to post-fab run-time security monitors. We show that the method is scalable for large systems and complex properties by optimizing the hardware monitors and applying it to a large SoC design based on a OpenRISC-1200 SoC. About 40 security assertions were specified in System Verilog Assertions (SVA). Following formal verification, the assertions were synthesized into finite state machines and cross optimized. Following code generation in Verilog, commercial logic and layout synthesis tools were used to generate hardware monitors which were then integrated with the SoC design ready for fabrication.

Keywords—System-on-Chip, Assertion Based Verification, System Verilog Assertions, Property Specification Language, Security and Hardware Monitors.

I. INTRODUCTION

System-on-Chip (SoC) designs are pervasive among modern computing systems. It is the dominant approach for embedded systems, wearable, mobile devices and Internet-of-Things (IoT) applications. Complicated and connected IoT devices are increasingly ubiquitous in our daily life, and they are recording, processing, and communicating financial, medical, social, and personal information. The importance of validation SoC’s security properties significantly raises after adopting SoCs in IoT devices. Many of these devices deal with sensitive data (assets), which should be protected against malicious or unauthorized access [1].

Modern SoCs integrate a variety of embedded elements and intellectual property (IP) blocks with complex communication infrastructure and protocols. A typical SoC includes hardware elements, software elements, programmable elements, clock and timing, test structures, and complex bus architectures. Some IPs are developed independently by untrusted third-party vendors. Untrusted IP blocks, malicious firmware and software with insecure communication threaten the trustworthiness of the SoC.

Existing post-manufacturing testing and functional validation techniques are insufficient to achieve comprehensive security assurance in the presence of vulnerable components. Often, an executed vulnerability manifests as an observable event after millions of cycles of operation, posing challenges to both pre-fab and post-fab validation processes. In addition, with the growing complexity of the modern SoC integration process, verification becomes a significant challenge. In some cases, verification tasks account for 70% of the total development activities [2].

Assertion Based Verification (ABV) is widely used in the SoC design flow. Assertions can be embedded into the design under verification to capture the design behavior. Security assertions verify security policies during the static as well as the dynamic verification of SoC design. Many specification languages are used to write assertions including System Verilog Assertions (SVA) [3] and Property Specification Language (PSL) [4].

Assertions can be synthesized and used as hardware monitors in emulation, simulation acceleration, post-silicon debugging, and runtime monitoring. Synthesizing assertions and generating hardware monitors has been proposed by many researches [5]–[10] based on two main approaches: modular and automata-based. In both methods, one monitor is generated for each assertion.

Assertion-based runtime monitors provide a practical method for continual monitoring of security properties during operation. Producing a resource-efficient assertion-based circuit is crucial if ABV is to be adopted for runtime monitoring. Typically, to monitor one security property, several assertions are developed and...
converted into hardware monitors.

This paper presents a methodology for the enforcement of security policies by the efficient translation of the security assertions into online monitoring circuits. Second, we develop an automated tool that translates SVA assertions into synthesizable Verilog modules. The tool can read SVA assertions and group them based on the underlying security property and generate only one monitor for each property. Finally, we demonstrate the feasibility of the method by applying the approach to a OpenRISC-1200 based SoC [11]. Our method reduces the area and power overheads of the monitoring hardware.

The rest of this paper is organized as follow: a brief introduction of the SVA language and a related work are discussed in Section 2. In Section 3 we show our proposed method and in Section 4 we present the case study. Section 5 shows the results and Section 6 concludes this paper.

II. BACKGROUND

Assertion Based Verification (ABV) is widely used in various phases of the SoC design flow, from specification to verification. In ABV, formal assertions are used to capture violations of the behavior in a design. Assertions are high-level expressions based on temporal logic that can capture the expected behavior of a design under verification (DUV). During the verification phase, hardware engineers develop assertions to capture the security policies of the DUV and verify security requirements of the design. We make use of the security assertions developed for pre-fab verification to generate online security monitors. SVA [3] is a part of SystemVerilog which is widely used to write assertions and is adopted in this research.

A. System Verilog Assertions (SVA)

SVA is a standardized assertion language to verify design behavior using assertions. In addition, assertions can be utilized to produce testbenches for validation. SVA assertions can be written inline in SystemVerilog modules. SVA contains two types of assertions: immediate assertions and concurrent assertions. Immediate assertions use simple Boolean operators to specify a design behavior independent of time. An example of immediate assertion is:

```
assert (Req != Ack)
```

Concurrent assertions use temporal operators to specify a design event that spans across time, therefore it is operated relative to a clock edge. An example of concurrent assertion is:

```
property P1;
 @(posedge clk)
   Req |-> ##4 Ack;
endproperty
assert property (P1);
```

Property P1 means that a Req should be followed by an Ack occurring four clocks after the Req is asserted.

SVA properties are built from Boolean expression and sequences. The implication operators ( |− > and |=>) are frequently used in this type of assertion. The left side of the operator is called the antecedent and the right side is called the consequent. SVA provides the parametrized assertions feature that allows reusing assertions. Assertions may be adapted to use in different situations. Example 1 shows a parameterized assertion.

Example 1: SVA Parametrized Assertion

```
property P2( clk, rst, except_trig, i_d, PC, i_f, except_prefix, hgr_bound, lwr_bound);
 @(posedge clk) disable iff (rst)
   (((except_trig[hgr_bound:lwr_bound] ==’d0) && except_trig[i_d])) ##1
      !except_prefix ) |-> (PC == i_f);
endproperty
```

Assertion_a can be instantiated as:

```
assertion_a: assert property ( P2( clk, rst,except_trig, 1, PC, 32’h800, except_prefix, 13, 2))
```

Synthesizing SVA or PSL assertions and generating hardware monitors has been proposed by several researchers [5]–[8]. The proposed methods fall into two categories: modular method and automata-based method. In both methods, one monitor is generated for each assertion. In this section both methods are discussed.

B. Modular Method

In the modular method, each operator in the assertion is implemented in a separate sub-module, and connected to other sub-modules to generate synthesizable HDL code for an entire assertion. Das et al. [5] deals with the generation of a SVA checker. In this work, sequence operators are implemented as sub-modules that inter-connect with input and output wires labeled “start” and “match” respectively. To synthesize SVA property, the tool decomposes the property into the basic sequence expression submodules and translates them into Verilog.
Katell Morin-Allory and others [6] have proposed a method called SyntHorus2 to synthesize assertions. SyntHorus2 is a modular approach to synthesize PSL assertions where each PSL operator is implemented in a dedicated module (hardware component). Then, all modules are produced and interconnected according to the property being implemented. Assertions generate two signals that indicate the assertion status. SyntHorus2 is a new version of an earlier SyntHorus tool [12].

Omar A. and others [13] designed an SVA compiler by implementing each SVA operator in a separate Verilog module. After parsing the assertion, it is compiled into two levels, sequence level where all required modules for sequence operators are invoked and property level where property modules are invoked. Finally, a merging unit matches all produced modules and connects them based on the assertion’s format.

C. Automata-Based Method

In this method, assertions written in PSL or SVA are converted into an automata representation before being compiled into HDL code. The construction of automata for assertions in dynamic verification has been explored by several researchers. In [14], E. Florian and others converted PSL formulas into symbolically represented Nondeterministic Generalized Büchi Automata (NGBA). The work was based on the Linear Temporal Logic (LTL) operators and Sequential Extended Regular Expressions (SEREs) in PSL. A normal form named Suffix Operator Normal Form (SONF) separates the LTL components and the SERE components, where each component is translated to a symbolic representation of an automaton.

Boulé and Zillic [15] present an automata-based approach for synthesizing sequence assertions described in PSL. Several automata construction algorithms are developed for a small number of operators called “base cases” and then a set of rewrite rules are produced and applied for the remaining operators.

The SynPSL tool [7] uses a similar method as the one described by Boulé and Zillic [16]. First, PSL formulas are reduced into base cases called PSL\text{min} and then a Nondeterministic Finite Automaton (NFA) is generated for each base case. NFA are converted to Deterministic Finite Automaton (DFA) before producing VHDL code. However, the SynPSL tool processes only simple Boolean expressions. Example 2 shows automaton generation from a SVA assertion.

Example 2: Assertion\_a in Example 1 can be converted to the automaton as shown in Figure 1. Where q0 and q2 represent the start state and failed state respectively. The q2 state is triggered when the security policy is violated.

 Automata-based method allows assertions to be combined and merged at the automata level. Moreover, at the automata level, the states can be minimized. In our work an automaton is defined by the four-tuple $A = (Q, I, \delta, F)$ where:

1) $Q$: a non-empty set of states
2) $\delta$: transition relation is set of triples $\{(s, \sigma, d), s \in Q, \sigma \in B, d \in Q\}$. B is a Boolean expression.
3) $I$: a start state
4) $F$: a final state

Boulé and Zillic [17] defined two modes of automata, conditional-mode to denote the acceptance of sequences and obligation-mode to denote the acceptance of properties. A sequence “holds”, if the occurrence of sequences is detected while a property “holds”, if no failure is detected. Figures 2 and 3 represent the automaton that accepts a sequence and an automaton that accepts a property, respectively.

III. PROPOSED APPROACH

Run-time monitors are utilized to monitor the behavior of a design where some requirements should be satisfied during operation. Typically, a security property includes several security policies. Synthesizing security property, SP, involves designing a monitor Mp for each policy.
$P \in SP$. The purpose of the monitor is to “observe” the behavior of $P$ and output 1 if it is violated and 0 otherwise. Any violation in any $P$ means $SP$ deviates from its intended behavior. Instead of producing one monitor for each $P$, all $M_P$ monitors can be combined and produce only one monitor to “check” $SP$ behavior while reducing the area and power.

We build on the work done by Boulé and Zilic [15] and the tool developed by Bilzor [16] to generate automata for individual assertions. After generating the automata, assertions in obligation mode can be combined with each other. Once an automata is generated for assertions, the combining process can be initiated. When the assertions contain an implication operator, the similarity of antecedent and consequent parts of assertions control the combining process. When the antecedents of combined assertions are not the same, non-deterministic combination automata is generated, and if the consequent parts are similar, the equivalent states are merged as shown in Example 3.

**Example 3:** In this example, two assertions with nothing in common are combined.

Assertion_1: $A \implies (c \#1 d)$  
Assertion_2: $B \implies (e \#1 f)$

Figure 4 shows the automaton for Assertion_1 on the left and that for Assertion_2 on the right. Figure 5 shows the combined automaton for both assertions. The combined automaton is a NFA because the edges are Boolean expressions which are not completely mutually exclusive. In such automata, one state could transit to multiple success states. These automata are different from conventional automata and conversion to a DFA is not required [18] for the following reasons:

- The monitor checks for the violation of assertions, if one of them fails, the final state is reached regardless of the status of the remaining assertions.
- One flip-flop per state is used in the implementations of the automata. Multiple flip-flops can be 1’s at the same time. This facilitates tracing multiple paths and activate two or more states at the same time. Each active state represents a run through the automaton. Therefore, any failure activates the final state.
- Producing DFA increases the number of states [19], Hence, increasing the resources consumed by the monitor.

Algorithm 1 describes the process of combining two assertions. First, the initial and the final states of both assertions are unified. Then, the similarity of the antecedents of the input assertions determines which part of the if statement (line 9) should be executed. For the assertions in Example 3, the else part (line 14) is executed. Combining algorithm results in the worst case of an automaton with $(x+y-2)$ states, where $x$ and $y$ are the number of states of automaton $A_1$ and $A_2$ respectively.

**Algorithm 1 Combine Automata-based Assertion**

<table>
<thead>
<tr>
<th>Input</th>
<th>Automaton $A_1 = {Q_1, \delta_1, I_1, F_1}$, $A_2 = {Q_2, \delta_2, I_2, F_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Create new automaton $A = {Q_1 \cup Q_2, \delta_1 \cup \delta_2, I_1, F_1}$</td>
<td></td>
</tr>
<tr>
<td>2. for each edge $(s_1, \sigma_1, d_1) \in \delta_1$ and $d_1 = F_1$ do</td>
<td></td>
</tr>
<tr>
<td>3. Update edge $d_1=F_2$</td>
<td></td>
</tr>
<tr>
<td>4. end for</td>
<td></td>
</tr>
<tr>
<td>5. for each edge $(s_1, \sigma_1, d_1) \in \delta_1$ and $s_1 = I_1$ do</td>
<td></td>
</tr>
<tr>
<td>6. Update edge $s_1=I_2$</td>
<td></td>
</tr>
<tr>
<td>7. end for</td>
<td></td>
</tr>
<tr>
<td>8. delete $I_1$ and $F_1$</td>
<td></td>
</tr>
<tr>
<td>9. if precondition ($A_1$) = precondition ($A_2$) then</td>
<td></td>
</tr>
<tr>
<td>10. NFA_to_DFA(A)</td>
<td></td>
</tr>
<tr>
<td>11. MergeEqualState(A)</td>
<td></td>
</tr>
<tr>
<td>12. Minimize(A)</td>
<td></td>
</tr>
<tr>
<td>13. else</td>
<td></td>
</tr>
<tr>
<td>14. MergeEqualState(A)</td>
<td></td>
</tr>
<tr>
<td>15. end if</td>
<td></td>
</tr>
<tr>
<td>16. Return Automaton A</td>
<td></td>
</tr>
</tbody>
</table>

**Example 4:** In this example, two assertions with similar consequents are combined.

Assertion_3: $A \implies (c \#1 d)$  
Assertion_4: $B \implies (c \#1 d)$

Figure 6 shows the automaton for Assertion_3 on the left and that for Assertion_4 on the right. When two
assertions have the same consequent part, the similar states are merged. As shown in Figure 7, state (q3) is the merged state for the equivalent states (q5) in both assertions.

To unite equal states, MergeEqualState algorithm is developed, it is described in Algorithm 2, where the outgoing edge is the input value of the transition and the destination state, and the incoming edge is the input value of the transition.

**Algorithm 2 MergeEqualStates.**

**Input**: Automaton A  
1: for each state \( s \in Q \) do  
2: \hspace{1cm} for each state \( t \in Q \) do  
3: \hspace{2cm} if (outgoing edge of \( t = \) outgoing edge of \( s \)) and (Incoming edge of \( t = \) Incoming edge of \( s \)) then  
4: \hspace{3cm} for each edge \( (s, \sigma, d) \in \delta \) and \( I = t \) do  
5: \hspace{4cm} Update edge \( I = s \)  
6: \hspace{3cm} end for  
7: \hspace{2cm} for each edge \( (s, \sigma, d) \in \delta \) and \( d = t \) do  
8: \hspace{3cm} Update edge \( d = s \)  
9: \hspace{2cm} end for  
10: delete state \( t \)  
11: end if  
12: end for  
13: end for  
14: Return Automaton A

IV. CASE STUDY

In order to investigate our method, we chose a large SoC design OpenRISC-SOC [11].

OpenRISC-SOC: is an open source SoC available at opencores.org, the design includes the OR1200 processor and various modules including Interrupt Controller, Tick Timer, Power Management Unit and Debug Unit. The block diagram of an OpenRISC SOC is shown in Figure 10.

After security analysis of the OpenRisc-SoC is done and security assertions are written, several security policies were monitored by developing 17 parametrized SVA security properties. 52 assertions were formulated for various IPs. 40 of them are related to 6 security properties, therefore they were combined to 6 groups. The first 28 of the assertions that were grouped are shown in Table I along with the group number, the others are omitted. Figure 11 shows the automaton generated for the assertions in Group 5. Where \( a_1, a_2, a_3, b_1, b_2, \) and \( b_3 \) are:

\[
\begin{align*}
\text{a1:} & \quad (\text{except}\_\text{trig}[13:3] == 'd0) \&\& \text{except}\_\text{trig}[2] \\
\text{a2:} & \quad (\text{except}\_\text{trig}[13:9] == 'd0) \&\& \text{except}\_\text{trig}[8] \\
\text{a3:} & \quad (\text{except}\_\text{trig}[13:6] == 'd0) \&\& \text{except}\_\text{trig}[5] \\
\text{b1:} & \quad (\text{except}\_\text{type} == 4'h9) \&\& \text{!except}\_\text{prefix} \&\& (\text{icpu}\_\text{adr}_o - (\text{ex}_\text{branch}\_\text{taken} | \text{spr}\_\text{pc}\_\text{we})) != 32'h900) \\
\text{b2:} & \quad (\text{except}\_\text{type} == 4'h3) \&\& \text{!except}\_\text{prefix} \&\& (\text{icpu}\_\text{adr}_o - (\text{ex}_\text{branch}_\text{taken} | \text{spr}_\text{pc}_\text{we})) != 32'h300) \\
\text{b3:} & \quad (\text{except}\_\text{type} == 4'h9) \&\& \text{!except}\_\text{prefix} \&\& (\text{icpu}\_\text{adr}_o - (\text{ex}_\text{branch}_\text{taken} | \text{spr}_\text{pc}_\text{we})) != 32'h900)
\end{align*}
\]
In this work, we used various tools to generate, compile, and verify the online-monitors. We used Python to develop all the algorithms needed to translate SVA assertion to Verilog and implement the combining method. Mentor Graphics Modelsim was used to simulate the generated monitors, Cadence JasperGold was used for formal verification of OpenRISC SoC along with the online-monitors. Synopsys Design Compiler with 90nm cell library was used to synthesize the design and compute the area and power results. Synopsys IC Compiler II was used to generate the layout of the SoC and the online-monitors. Figure 12 shows the layout of the Group 5 monitor.

The area and power were computed for the monitors before and after combining. For the OpenRISC test case, Figures 13 and 14 show the total area and power consumed by combined and individual monitors in each group. As noticed, the area is reduced by 45% and the power is reduced by 34%.

This paper proposed an optimized method to migrate pre-fab design security assertions to post-fab run-time security monitors. The method includes combining of automata-based runtime monitors to minimize the area and power consumed by online monitoring. The goal is to assess SoC’s security properties continuously during operation. To demonstrate this method we applied it to the OpenRISC based SoC. The test case results proved the efficiency and feasibility of the method.

We thank Padmaja Bhamidipati, Subashree Raja, Suriya Srinivasan, and Sanjay Siddharth who are members of the FACETS team for their comments and feedback. This work has been funded by the Air Force Research Laboratory Under Case Number AFRL-2020-0443.
Table I: A Set of OpenRISC Assertions Combined into Groups

<table>
<thead>
<tr>
<th>Assertion ID</th>
<th>Assert Property (p); where p is:</th>
<th>Group ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>`dltb_done and ((!dcpu_we_t &amp; !sr[0] &amp; !dltb_ure) or (dcpu_we_t &amp; sr[0] &amp; !dltb_ure))</td>
<td>GROUP 1</td>
</tr>
<tr>
<td>2</td>
<td>`dltb_done and ((!dcpu_we_t &amp; !sr[0] &amp; !dltb_ure) or (dcpu_we_t &amp; sr[0] &amp; !dltb_ure))</td>
<td>GROUP 1</td>
</tr>
<tr>
<td>3</td>
<td>`(((except_trig)[3] == ‘d’0) &amp;&amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>4</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>5</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>6</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>7</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>8</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>9</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>10</td>
<td>`(((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[1])) #1 (except_prefix)</td>
<td>GROUP 2</td>
</tr>
<tr>
<td>11</td>
<td>`(rst</td>
<td>-&gt; <code>lsbu_stb_o &amp;</code>lsbu_cyc_o)</td>
</tr>
<tr>
<td>12</td>
<td>`(rst</td>
<td>-&gt; <code>lsbu_stb_o &amp;</code>lsbu_cyc_o)</td>
</tr>
<tr>
<td>13</td>
<td>`(reset</td>
<td>-&gt; <code>m_stb_o &amp;</code>m_cyc_o)</td>
</tr>
<tr>
<td>14</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>15</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>16</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>17</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>18</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>19</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>20</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>21</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>22</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>23</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>24</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>25</td>
<td>`((!(en &amp; Srs(e)(wb_cyc_o) &amp; Srs(e)(wb_stb_o) &amp; (wb_we_o == 0)) #2 ((wib_ack_i) &amp; &amp; (s_SEL_one_hot == 4’d1))</td>
<td>GROUP 4</td>
</tr>
<tr>
<td>26</td>
<td>` (((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[2])) #1 (except_type == 4’hd) &amp; &amp; !except_prefix)</td>
<td>GROUP 5</td>
</tr>
<tr>
<td>27</td>
<td>` (((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[2])) #1 (except_type == 4’h3) &amp; &amp; !except_prefix)</td>
<td>GROUP 5</td>
</tr>
<tr>
<td>28</td>
<td>` (((except_trig)[3] == ‘d’0) &amp; &amp; except_trig[2])) #1 (except_type == 4’h9) &amp; &amp; !except_prefix)</td>
<td>GROUP 5</td>
</tr>
</tbody>
</table>
REFERENCES


