Modelisation and Validation of a Chip Embedded Architecture for Secure Applications

CHAVET Cyrille
Student Id. : 97000501

MSc Thesis

Advisor : Mme BORRIONE Dominique

MSc Thesis
Acknowledgements

I want to thank TIMA laboratory and Dominique BORRIONE, Professor and leader of the VDS group (Verification & modeling of Digital Systems), who offered me good conditions to carry out this project.

I also address particular thanks to Diana TOMA, Emil DUMITRESCU and Julien SCHMALTZ for their valuable assistance and advices.

I thank also Menouer BOUBEKEUR and Pierre OSTIER for their valuable technical advices, as well as Alexandre CHAGOYA for his availability and his good mood, and Emmanuel Zarpas (IBM Haïfa) who made available the FoCs and RuleBase software.

Lastly, a very particular thank to all the VDS team to have welcome me, and for the good mood in the team:

Ghiath AL SAMMANE, Claude LE FAOU and Bénédicte FLUXA, VDS group secretary, for her kindness and her availability.

May all be ensured of my deep gratitude, in particular those with whom I could hold these discussions which often make work so pleasant and stimulating... they will recognized themselves.
Modelisation and Validation of a Chip Embedded Architecture for Secure Applications

CHAVET Cyrille

Abstract:

The development of increasingly secure, and relatively complex, embedded architectures leads the designers to use formal verification tools in order to guarantee that these circuits fulfill a set of properties ensuring their security level.

During this master project the SHA-1, a standardized secure hash algorithm, was studied. This circuit was specified using a VHDL subset for synthesis, also standardized.

Thereafter, the formal checking of this circuit design was carried out using two tools: FormalCheck and RuleBase. The second one uses a particular formalism to express the properties: Sugar, a specification language combining temporal operators, logical and regular expressions, developed by researchers at IBM laboratory in Haïfa.

We aimed at evaluating what this language brings to the formulation of circuits properties, as well as the performances of the checking tools developed to use it. Thus, an original approach to model-checking was tested: evaluation of the potentials of the FoC's automatic tool for monitor generation, in order to use monitors to facilitate the expression of properties for the checking tools.

Key words:

Model-checking, formal specification language, monitor, Sugar, RuleBase, FoC's, FormalCheck.

TIMA Laboratory
46, avenue Félix Viallet 38031 GRENOBLE Cedex France
Ph: +33 4 76 57 46 15  Fx: +33 4 76 47 38 14
# Table of contents

*Introduction* ................................................................. 6

I.  *Context* .................................................................................. 8
   A.  *The SHA-1 algorithm* ......................................................... 8
       1.  *Standard specification* .................................................. 8
       2.  *A modeling language: VHDL* ....................................... 10
       3.  *Context of use* ........................................................... 10
       4.  *Detailed description* .................................................... 12
   B.  *Properties to be verified* ................................................... 17
       1.  *Properties to be verified by the SHA-1* .......................... 17
       2.  *A model simplification* ............................................. 18
   C.  *A formal specification language: Sugar* ............................ 19
       1.  *Sugar 2.0* ..................................................................... 19
       2.  *Sugar operators* ........................................................ 19
       3.  *Sugar Extended Regular Expressions* ............................ 21

II.  *SHA-1 validation with FormalCheck and RuleBase* ................ 23
   A.  *Verification tools* .......................................................... 23
       1.  *FormalCheck* ............................................................. 23
       2.  *RuleBase* .................................................................... 24
   B.  *Properties to verify* ......................................................... 25
   C.  *Experimental results* ....................................................... 27

III.  *Verification using monitors* .................................................. 28
   1.  *The FoCs tool* ............................................................... 28
   2.  *Properties* ..................................................................... 29

IV.  *Experimental results analysis* ............................................... 30

*Conclusion* ................................................................................ 31

*Bibliographical sources* .............................................................. 32

*Appendix* .................................................................................. 34
Table of figures

Fig. 1 : Using the SHA-1 with the DSA ........................................................................................................ 8
Fig. 2 : Last 512 bits block of a message .................................................................................................. 9
Fig. 3 : External view of the circuit ........................................................................................................... 11
Fig. 4 : Control automaton of the padding process ............................................................................... 13
Fig. 5 : Control automaton of the hash process ..................................................................................... 15
Fig. 6 : Control automaton of the sampling process .............................................................................. 16
Fig. 7 : FormalCheck Verification procedure ......................................................................................... 23
Fig. 8 : Results obtained on the reduced version of the model .............................................................. 27
Fig. 9 : FoCs environment ....................................................................................................................... 28
Fig. 10 : State encoding for the automata of the two main processes ................................................. 68
Introduction

This project takes place in the heart of the design of micro-electronic circuits, and more precisely at the level of their formal checking. The aim is to specify a circuit implementing a secure hash algorithm, and to carry out its checking by exploring the possibilities offered by a new formalism: Sugar. Before presenting with more details the subject and the organization of this document, we start by giving an overall picture of the fields in which this work takes place.

The ever growing level of complexity (up to several million gates) of electronic systems, and the reduction of their time to market, makes it impossible to guarantee an exhaustive cover of the errors with the traditional validation by simulation techniques. For this reason, any alternative or complementory checking approach receives a considerable interest. Formal methods for the verification of digital systems bring an effective answer, and for this reason are subject of a keen interest from the companies. There are mainly three families of tools: equivalence checkers, proving that two models of a circuit are equivalent; model checkers, which consist in the proof of a set of properties, expressed in a temporal logic, on a model of the circuit; and finally, theorems provers which, starting from a model of the circuit in an axiomatic theory, aim at showing theorems. Because of the level of expertise required to guide the demonstrator towards the proof, the latter are mainly used to guarantee the correctness of critical systems.

In order to check properties, it is necessary to be able to express them starting from elementary propositions. To do so, various temporal logics have been defined (CTL or LTL for example). The enumerative CTL model-checking algorithm, primarily due to Queille, Sifakis, Clarke, Emerson and Sistla [12, 2] has a fundamental place in the formal verification research field. Coupled to a binary decision diagram model representation, and with the symbolic approach developed by MacMillan [19], model-checking algorithms made it possible to check automatically system models such as bus protocols. The two tools we are studying here are based on this symbolic approach. However, writing complex properties using CTL or μ-calculus is painful and error-prone according to circuit designers. To solve this problem, two approaches were proposed.

- The authors of FormalCheck have limited the subset of CTL recognized by FormalCheck. This simplifies the writing of properties, by forcing the designer to use an intermediate rigid, but very intuitive, interface.
- Another solution is to preserve CTL expressiveness, and to offer a more elaborate language to express complex properties. This second approach was taken by the IBM research Laboratory in Haifa. Sugar [13], a formal specification language, has been defined to be used in conjunction with the RuleBase model-checker. All the software processing needed for Sugar have also been developed: parser, automatic generation of monitors, ... The last evolution of this formalism, Sugar2.0, was donated to Accelera’s "Formal Checking Tool Committee" as a basis to develop a standard property specification language, PSL.

The current requirements of the embedded systems markets, such as smart card, leads to the development of products that are increasingly protected. A way of protecting oneself from intrusions is to ensure that the applications which are executed on the circuit are authorized. To tackle this problem, there exist algorithms for the automatic generation of signatures, e.g. the Digital Signature Algorithm, where one of the key steps is to produce a condensed form of the message to be identified. The SHA-1 [11], a standardized secure hash algorithm, produces a 160 bits condensed representation for a message of arbitrary size less than $2^{64}$. It is essential, for any designer wishing to use it, to make sure that the available model of a circuit correctly implements
the standard. To this aim, traditional model-checking techniques are a solution. During this work, a more original model-checking approach was also explored. The underlying idea is to benefit from Sugar and FoCs [5] (an automatic monitor generation tool). The circuit model is composed with the monitor generated for the property at hand (the property being written in Sugar). It is then straightforward to express that the monitor never enters an error state. To evaluate the performances of this approach, the checks were done with FormalCheck [1], and the tool for which Sugar was developed, RuleBase [9]. The first part of this report will present Sugar, as well as the SHA-1 algorithm, its description and the properties it must verify. Then, the various verification steps results will be presented and finally analyzed.
I. Context

A. The SHA-1 algorithm

The Secure Hash Algorithm is a standard, which computes a condensed message representation. For a message length lower than \(2^{64}\) bits, the SHA-1 produces a 160 bit output called "digest". This can be used as input to a Digital Signature Algorithm (cf Fig. 3) which generates or checks the signature of a message.

We first detail the SHA-1 specification before describing the version of this algorithm that we implemented.

1. Standard specification

The SHA-1 is said to be secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. The least modification of the message will result, with a very strong probability, in a different digest.

The SHA-1 is used with the DSA (Figure 1), within the framework of identification of e-mails, secured funds transfers, and other applications which require data integrity assurance and data origin authentication. The SHA-1 may also be used whenever it is necessary to generate a condensed version of a message.

<table>
<thead>
<tr>
<th>Signature Generation</th>
<th>Signature Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message</strong></td>
<td><strong>Received Message</strong></td>
</tr>
<tr>
<td><strong>SHA-1</strong></td>
<td><strong>SHA-1</strong></td>
</tr>
<tr>
<td><strong>Message Digest</strong></td>
<td><strong>Message Digest</strong></td>
</tr>
<tr>
<td><strong>Private</strong></td>
<td><strong>Digital</strong></td>
</tr>
<tr>
<td><strong>DSA Sign Operation</strong></td>
<td><strong>DSA Verify Operation</strong></td>
</tr>
<tr>
<td><strong>Key</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td><strong>Key</strong></td>
</tr>
</tbody>
</table>

Yes - Signature Verified  
No - Signature Verification Failed

Fig. 1 : Using the SHA-1 with the DSA
The standard gives a computation algorithm of the digest. But before executing it, it is necessary to format the data (Padding). Indeed, the SHA processes the data only per 512 bits blocks according to precise criteria: if the message is less than 448 bits, it is stored in the first 448 bits of the block, with the end bit added to the message. The last 64 bits of the block store the message length (Fig. 2).

With 64 bits for the length, the longest message the SHA can take has a maximum of \(2^{64}\) bits. The block on which the SHA works is exactly 512 bits. Now, in general, the message is longer, and must be divided in several 512 bits blocks. When the remaining number of bits is lower than 448, the last block is completed as shown on Figure 2.

![Diagram of last 512 bits block of a message](image)

**Fig. 2: Last 512 bits block of a message**

As depicted on Fig. 2, after the last bit of the message, it is necessary to put an end bit, supplemented with zeros up to the 448th bit, and finally the length of the message is stored on the last 64 bits.

We note that there are four possible situations according to the message length:

- **Message length greater than 512 bits**: a 512 bits block is recovered from the first 512 bits of the message and processed by the hash algorithm, then the rest of the message is processed.
- **Message length lower than 447 bits**: there is enough space in a block to hold all the message, plus the end bit, plus the length.
- **Message length between 448 and 511 bits**: in this case, the message, plus the end bit, encroach on the zone reserved for the length. The remaining free bits in this block are filled with zeroes, and the block is transmitted to the hash algorithm. A last block containing 448 zeroes, and the message length on its last 64 bits, has to be added.
- **Message length is exactly 512 bits**: this case is, in fact, similar to the former one, with the difference that the end bit, which cannot be catenated to the message, is then added in an additional block, which has a “1” as first bit, and then 447 zeroes and finally the message length on its last 64 bits.
The modeling effort for the SHA-1 relates mainly to making the padded message. The hash algorithm being provided by the specification in pseudo code, it is enough to implement it just as is. The properties that are expressed later in this document state that the padding is made correctly and that the various processes of the VHDL model communicate as expected. Once again, the hash algorithm is not detailed here, the interested reader can refer to the standard [11].

2. A modeling language: VHDL

Initially supported by a contract of the american "Department of Defense", VHDL was standardized by the IEEE in 1987, and then was revised in 1993 and 2000. It became quickly essential for hardware systems design, and shares today a dominant position with another standardized language, VERILOG.

The VHDL language is characterized by its simulation semantic. A VHDL system is a set of processes which communicate using signals. A process is a program made up of traditional sequential instructions, as well as suspension instructions (specifying a duration, a condition...) and signal assignments. The reference manual [17] also defines an event driven simulation algorithm and a procedure which, starting from the description of such a system, builds the data structures which will be used to simulate it. All the remainder is only "syntactic sugar".

The VHDL language is defined in a very prosaic English, sometimes obscure, but it is sufficiently rich in constructs to make it possible for the designer to describe his circuit in various design levels (architectural level, data flow level or register transfer level). However, only the VHDL subset for synthesis is accepted by verification tools. This subset has been standardized [18]. Thus, for formal checking, we must limit ourselves to the register transfer level.

3. Context of use

The circuit is described so that the padding, "digestion" and "digest" delivery phases are separated. As a result, the input/output interface of the model is :

- **Inputs:** Data – message, cut in 8 bits words, NbValid – number of valids bits on the message bus (data), FIN – end-of-message signal, RESET – reset signal, TOK – clock signal.

- **Outputs:** Digest – message condensed version, on a 16 bits bus, Digest_OK – digest ready signal, STOP – suspension of the transfer, in order to process the 512 bits already recovered.
The circuit running principle is as follows: the message is transmitted to the SHA-1 on the Data bus, when END is low. During the same clock cycle, the NbValid bus says how many bits are relevant on the Data bus. If the number of bits sent to the SHA-1 is higher than 512, the STOP signal requires the suspension of message transfer, when raised to high, to let the already recovered 512 bits be processed. Then, this signal falls, enabling the transfer of the remainder of the message. The message end is announced by the signal FIN set to high.

At this point, it is already possible to define the communication between our circuit and its environment in more details:

- The number of valid bits (port: NbValid) corresponds to the number of bits belonging to the message on the Data bus during the same clock cycle (port: TOK).
- On the Data bus, the valid bits starts at the high-order bit.
- The signal FIN becomes true only when all the message has been delivered.
- If the signal FIN is true, it cannot fall (new message) as long as the STOP signal is true (SHA-1 is computing a hash).
- If the signal STOP becomes true and if we are transferring a message (FIN is false), the signal FIN remains low, but we don't transfer any more data until STOP becomes false again (SHA-1 is computing the hash algorithm on a block of the message).
- The signal Digest_OK becomes true one clock cycle before digest transfer starts.

Nota bene: since a digest is 160 bits long and the output Digest bus of our SHA is only 16 bits wide, we need 10 clock cycles to recover the complete digest.
4. **Detailed description**

The circuit description (cf appendix 1 and 2 for source codes) is composed of three distinct parts, three processes. Each process carries out a part of the work: padding, computation of the digest and sampling of the result. In fact, the design recovers the message in a 512 bits block, according to the various cases identified previously. This block is then digested by the hash algorithm. Last, if the message is finished, the "digest" is sampled, otherwise the hash algorithm results are stored and will be used for the next block hash computation.

**a. Recovery and padding of the message**

The first process implementation, which carries out the padding, is described here as an automaton. A different state is entered for each situation (normal, crushing the length reserved zone, overflowing the block). The general ideas are:

- As long as there remains space in the first 448 bits of the block, valid message bits are stored there.
- If the message is finished, the next step consists in shaping the block. This can cause a particular case (the end bit is put on the 449th bit, which belongs to the 64 reserved for the length), then the control is passed to the second process.
- If the message is not finished, the incoming data are written in the length zone. Then, either the message finishes before all the reserved zone is used, in this case we finish the padding and the control is passed to the second process; either all the reserved zone is used and the message is not finished, in this case the message transfer is suspended before passing the control to the hash process.

When the signal STOP is true, no data may be transmitted. In fact, this is necessary in two quite distinct situations: either, the message is more than 512 bits and the transfer is temporarily suspended, or the message was completely transferred and we have to wait until computations are finished and the digest sampled, before a new message transmission can be authorized.

The automaton represented on Figure 6 reuses these principles:

- The No_Problemo, Ecrase and Depasse states represent the three situations explained above. The states Trais and Former deal with the padding. The other states are states in which we wait until the end of message processing, or in which another message is awaited.
- The StopSig, busy1 and busy_ended signals (cf Fig. 7) are used to compute the signal STOP.
- The NbBlok is an integer variable which makes it possible to know where we are in the current block.
- The Traiter, Sblok_null and Sblok_null_un signals are intended for the hash process. The first one requests the processing of a 512 bits block; the others announce that, following the padding step, it is necessary to finish the digest computation with an additional block: either a null block (only zeros and the length), or a null block with one “1” as first bit, followed by 447 zeros and the length. These cases are differentiated by the condition.
- Finally, Fin_Phase1 and Traite_Fin are signals coming from the second process.
Fig. 4: Control automaton of the padding process
b. Computation of the hash function on a block

The second process implements the hash function according to the standard. It is described in the shape of an automaton too. Here, it is enough to follow the standard algorithm, the only additions are signals enabling the communication with the two other processes.

The hashing results for the successive blocks of a message are progressively stored in five 32 bits words (H0, H1, ..., H4). In fact, these Hi are initialized with a given value [11] for each new message, and are used in the hashing process. When the first message block has been processed, the results are in the Hi if the message is finished. Otherwise, the current value of the Hi is used as new inputs for the hash process on the next message block.

The signals used by this automaton are:

- Fin_phase1 tells the first process that the 512 bit block has been fully copied, and that it can be updated (the first process goes to state RaZ as a result).
- Traite_Fin indicates the end of hashing block computation.
- Signal S_OK is used to ensure that a null block (or a null block with a “1” as first bit) is computed only once. Indeed, if such a block is required, it is created in state et5 (after the hash step of the transmitted block), and its digest is directly calculated (transition towards the state et1), therefore the automaton will return in this state et5 and will not have to process a particular block again.
- When the computation and the message are finished, the signal Termine takes value true to notify the sampling process that it can deliver the "digest".

In the standard, two versions of this hash algorithm exist: the first version works on an eighty 32 bit word table (2560 bits); the other version deals with sixteen 32 bit words, the block’s 512 bits, and carries out the same computations by realizing a modulo 16 crossing of the table. This version is designed for limited memory applications. The two versions were implemented during this work. We discuss the verification of only one of them, as they only differ in the data path (traversing a 80-word table once vs a 16-word table several times with a modulo 16 indexing).
Fig. 5: Control automaton of the hash process

MSc Thesis
TIMA-CMP groupe VDS, 2002/2003
c. Result sampling on output bus

The last process only samples the digest. This last process resumes when the hash process indicates that the \(H_i\) contain their final values. The results are put on the bus called DIGEST.

![Control automaton of the sampling process](image)

*Fig. 6: Control automaton of the sampling process*

The data are transmitted in 16 bit blocks, starting with \(H_0\) which contains the digest high order bits. Signal DIGEST_OK is set to true to tell the environment that it must recover the results at the next cycle. When this step is finished, Fin_delivre is set to a high state during one clock cycle, to notify the other process.
B. Properties to be verified

1. Properties to be verified by the SHA-1

The properties which should be checked relate first to the overall design. But it is also necessary to check properties on the internal processes.

The general properties we can express are:

- As soon as all the message contents have been transmitted (FIN becomes true), the signal indicating that we can recover the digestion results is true (DIGEST_OK becomes true).
- If a message digest is available (DIGEST_OK becomes true), no message is being transmitted (FIN is true).
- After a message has been completely transmitted (FIN becomes true), the internal signal requiring the hash process to work must be set (TRAITER becomes true).

The processes must display, during their execution, the following properties:

- If the padding process is in reset state (RaZ), it means that the second process no longer needs the block data (Fin_Phase1 is true), then we can reset it safely.
- If data transfer suspension is required (STOP becomes true), the final states of the processes are never reachable as long as the data transfer is not authorized again (STOP becomes false).
- When we ask to process a block which has just been recovered (TRAITER becomes true), we expect this process to finish (FIN_TRAITER eventually becomes true).
- If data transfer is suspended (STOP becomes true), then it is necessary to process the already received block (TRAITER becomes true).
- If a block hashing is finished (TRAITE_FIN becomes true), either the message is not finished (FIN is false) and the first process is in the initial state, or the message is finished (FIN is true) and in this case the first process is in its final state.
- If we are shaping a block (add the end bit, complete with zeroes...), i.e. the first process is in state Forme1 or Forme2, the message must be finished (FIN is true).
- If a message requires a particular block (null, or null with “1” as first bit), i.e. the signal SBLOK_NULL or signal SBLOK_NULL_UN is true, the message must be finished (FIN is true).
- When a message appears (FIN becomes false), the first process must go through the state No Problemo.
- If the second process is in state et6, that means that we are working on a piece of a message whose transfer has been suspended (STOP_SIG is true)
- When the message has been completely transmitted (FIN becomes true), processes end up going in their final states, and they remain there until a new message is transmitted.

These properties are expressed with the adequate formalism for the checking tool being used. In particular, within the framework of Rulebase model-checking and monitor generation, it is necessary to use Sugar.
2. A model simplification

The system, as presented and modelled, and as specified in the standard, requires many resources; the model is too complex to be checked. Model-checking tools regard any memorizing element as a state variable. Moreover, even for the SHA simplified version, we need 512 memorizing bits, i.e. $2^{512}$ possible configurations. Tools limits are exceeded. To be able to check the implemented model, it was thus necessary to simplify the circuit, which means reduction of the data path.

The functionality of the circuit does not change, only the data structure size is modified. This only influences the operations carried out in states $et1$, $et2$ and $et4$. The remainder of the process does not change. In fact, block dimension does not have any influence on the control part of the circuit. It can be regarded as a parameter, since no property relates to it. The internal communications between processes, and the communication between the SHA-1 and its environment which are the subject of our property checking, do not depend on the size of data, and this modification preserves the results obtained on the properties.

The reduced version of the SHA-1 has the following data structure size:

- The 512 bits block is replaced by but on a 6 bit block, including 2 bits for the length.
- The data bus is reduced from 8 to 1 bit, and bus DIGEST is reduced from 16 to 5 bits.
- All the constants are reduced to 1 bit, and the different subfunctions used for computations also work on 1 bit.
- The eighty 32 bit word table becomes an eight 2 bit word table.

The remainder of the design does not change.
C. A formal specification language: Sugar

The purpose of Sugar is to enable fast writing of formal properties. In particular, the complex expressions of CTL formulas are difficult to read and write. Sugar provides a set of operators to simplify the writing of such properties. This language offers traditional CTL operators plus a set of mechanisms and particular constructors.

1. Sugar 2.0

Sugar is a formal specification language designed for hardware. It has been developed by the IBM research laboratory as specification language for RuleBase. The schedule of conditions was as follows:

- Easy to learn, read and write.
- Concise syntaxe.
- Rigorously well-defined formal semantics.
- Expressive power, allowing the specification for a large class of real world design properties.
- To be usable with simulation tools.

Initially, Sugar was purely "syntactic sugar" to express CTL properties in a simple and readable way. Then, it evolved to allow the expression of LTL properties, and to make it possible to write regular expressions, SERE (sequence), however the syntactic dimension of sugar remains important.

Lately, Sugar was used as a basis for the definition of a standard specification language [8] by the Accelera’s “Formal Checking Tool Commitee”.

Sugar consists of 4 layers:

- **Boolean layer**: consists of boolean expressions, in a syntax determined by the flavor of Sugar being used (VHDL, Verilog or EDL).
- **Temporal layer**: consists of temporal properties which describe the relationships between boolean expressions over time. Sugar can be used to work on synchronous (one or more clocks) or asynchronous designs.
- **Verification layer**: consists of directives which describe how the temporal properties should be used by verification tools (assert, assume, restrict, …).
- **Modeling layer**: provides semantics to model the behavior of the inputs and auxiliary variables of a model.

2. Sugar operators

Sugar offers boolean logic operators (!f, f1 and f2...), as well as CTL operators: AX, EX, AG, EG... But the main interest of this language is that it provides a lot of additional operators.
Let \( n, i, j \) be integers and \( f, f_1 \) and \( f_2 \) some properties.

- **next\([n]\) f**: short cut to write \( n \) times \( \text{AX} \)
  
  \[
  \text{next}[3] f \_ \_ AX AX AX f
  \]

- **next\(_e\)[i..j](f)**: forces \( f \) to be true in a future state between the \( i^{th} \) and the \( j^{th} \), starting from the current state.
  
  \[
  \text{next}_e[2..3](f) \_ (AX \text{ not } f) \text{ and } (AX AX f \text{ or } AX AX AX f)
  \]

- **next\(_a\)[i..j](f)**: forces \( f \) to be true in all future states between the \( i^{th} \) and the \( j^{th} \), starting from the current state.
  
  \[
  \text{next}_a[2..3](f) \_ (AX \text{ not } f) \text{ and } (AX AX f \text{ and } AX AX AX f)
  \]

**CTL** operators exist in a strong and a weak version. The strong version means that a terminating condition must be fulfilled. For example:

- **f\(_1\)** before **f\(_2\)**: the first **f\(_1\)** must be observed before, or at the same time as the first **f\(_2\)**, if **f\(_1\)** is true some day.

- **f\(_1\)** before! **f\(_2\)**: strong version, same as above, but in addition **f\(_1\)** must be observed.

The use of the exclamation point means that we use the strong version of the operator. There also exists, for some operators, the possibility of stating that there is an overlap (symbol: \( _\) ) between various signals:

- **f\(_1\)** until **f\(_2\)**: this syntax has the same semantic as the following CTL formula:
  
  \[
  A \ f_1 W f_2
  \]

- **f\(_1\)** until \( _f_2\)**: the semantics are the same as above, but **f\(_1\)** remains true in the first cycle where **f\(_2\)** is true. In this case, there is an overlap between **f\(_1\)** and **f\(_2\)**.

Sugar also offers, under all the possible version and overlapping combinations, several other operators such as for example:

- **next_event\(_e\)(f\(_1\))(f\(_2\))**: if **f\(_1\)** is true some day, then in the next state **f\(_2\)** is true.

- **next_event\(_a\)(i..j)(f\(_1\))(f\(_2\))**: during \( i^{th} \) occurrence of **f\(_1\)** (and the \( i+1^{th} \) ... and the \( j^{th} \) occurrence) **f\(_2\)** is true too.

**Example:**

We wish to express the following property: if we detect a read request with a tag equal to \( i \), then on the next 4 data transfers we expect to see tag \( i \).

\[
\text{Forall } i \text{ in } 0..7
\]

\[
\text{Always } ((\text{Req}_\text{Lec} \& \text{tag}[2:0] == i) \_ \_ \text{next}_\text{event}_\text{a}(\text{donnée})[1..4] (\text{tag}[2:0] == i))
\]
Finally, Sugar offers two other operators allowing to express properties over time period defined with respect to other expressions:

- **within**($f_1, f_2$)($f$): $f$ is true during the time interval starting at the cycle when $f_1$ becomes true and ending at the cycle when $f_2$ becomes true (not included).

- **whilenot**($f_1$)($f_2$): means that in every computation, formula $f_2$ is true now and remains true at least until one clock cycle before $f_1$ is true.

**Examples:**

We wish to express the following property: after a request signal, there is an acknowledge, and between these two signals busy is true.

Always ( within(req, ack)(always busy) )

An other example: whenever a high priority request is received (req_prio is asserted), then the next two grants (assertion of signal grant) must be sent to a high priority requester ($dst = plus_prio$).

Sugar:

$$\text{Always} (\text{req\_prio\_next\_event\_e} (\text{grant})[1..2] (\text{dst = plus\_prio}))$$

Equivalent CTL expression:

$$AG(req\_prio\_A[\neg\text{grant} W ((\text{grant} .\text{dst}=\text{plus\_prio}) + (\text{grant} .AX A[\neg\text{grant} W (\text{grant} .\text{dst}=\text{plus\_prio})]))$$

3. **Sugar Extended Regular Expressions**

The various types of operators we have just seen can be combined to give quite complicated properties. However, to write such properties is sometimes cumbersome and to read them can be difficult. Again, Sugar provides a possible alternative way to circumvent the difficulty, reasoning about sequences: SERE.

These sequences are Sugar constructs, used to describe execution path on which formulas must be checked. They ease the expression of formulas which would be very complex to write in CTL.

**Example:**

$$AG(reqin\_AX(ackout\_AX(\neg abortin\_ackin \& AX ackin)))$$

\[ \uparrow \]
Always(\{reqin; ackout; !abortin\} \implies \{ackin; ackin\})

The sequence has two parts, a list of events \{e_1, e_2, ..., e_n\} and a Sugar formula (f).

- \{e_1; e_2; ... ; e_n\} (f): if on some computation path all the events take place in the given order, then the formula must hold on this path at the same cycle as the last event in the list.
- An event e_i can be: a boolean expression, a Bool. Exp.[*i..j], a Bool. exp. [=i..j], a SERE, ...

Examples:

Suppose that we want to see a scenario in which a cache line is modified (modif\_cache), and later becomes exclusive without being invalidated in between. The following sequence claims that this path is impossible (and its counter example will demonstrate such a path).

\{[*]; modif\_cache; !invalidate \& !exclusive [\*]; exclusive \}(false)

If there is a write request, then it starts at an even address; and during the transfer, data are alternatively written at odd and even addresses. The message begins when the signal ack becomes true followed by the signal abortin becoming false.

Sugar:

Always within !(\{write-req ; ack ; !abortin\}, write_end) 
\(\{{\!data[*] ; data\&!addr[0] ;{\!data[*] ; data\&addr[0]}}[*4]\})

Equivalent CTL expression:

\(AG ( (write\_req.AX ack.AX AX \neg abortin) \_ 
\ AX AX (\neg data U (data. \neg addr[0].
\ AX (\neg data U (data.addr[0]. ... ))) \))

SEREs provide an easy way to express sequences of Boolean expressions over time. There exist many more operators (cf. [14, 15]) such as the possibility to clock an expression (signals of the expression will be evaluated when this clock is true). However, it is not possible to make an exhaustive presentation of all the possibilities offered by Sugar within the framework of this report.
II. SHA-1 validation with FormalCheck and RuleBase

A. Verification tools

1. FormalCheck

FormalCheck is an industrial model checker. FormalCheck supports the synthesizable subsets of the Verilog and VHDL hardware design languages. The user supplies FormalCheck with a set of queries to be verified on the design model, thanks to a friendly, but very binding, user interface.

This tool is able to reduce the model automatically relatively to the property to be checked, to the greatest extent possible. FormalCheck uses two approaches of reduction algorithm (*1-step* and *iterative*). The iterative reduction algorithm attempts to find a small portion of the design that can be used to verify the current query. This technique guarantees that queries proven to be true on the small portion of the design, are also true on the entire design.
This software directly translates a VHDL model into a state machine, and checks the property for all the inputs and in all the possible states. One can express liveness or safety properties, which must be written in three parts:

- The first part is optional, and is used to express when the computation of the property must start:
  
  **After** Boolean expression

- The second part is compulsory, and is used to express the property using a temporal logic:

  **Temporal operator** Boolean expression

  The main temporal operators are:
  - **never:** FormalCheck will explore all possible inputs and reachable states to verify that the condition can never occur.
  - **always:** it will explore all possible inputs and reachable states to verify that the condition can never be false.
  - **eventually:** the property must occur in at least one reachable state.

- The third part, optional, is used to express when the computation of the formula must end:

  **Unless** Boolean expression

For each part, the Boolean expressions are written with signals from the design under test, and from logical operators (including comparisons over arithmetic expressions).

### 2. RuleBase

This is a symbolic model-checking tool, developed by the IBM research labs in Haïfa. It uses an enhanced version of SMV. Large development efforts were made in order to fight the state explosion problem, thus allowing the checking of industrial models.

Modeling the environment is the most important problem when we want to use RuleBase. Environment models are written in the RuleBase Environment Description Language (EDL), a dialect of the SMV language. For example, the constraints on the signal FIN behaviour have been expressed as a set of properties, which have to be assumed by RuleBase (key word `assume`). Then the tool is able to take them into account:

- Assume (AG (fell(FIN) _ !STOP)): if there is a new message (FIN becomes false), we must have finished to process the previous one (STOP is false).

- Assume (AG ((STOP & FIN & !RESET) _ FIN)): if the SHA-1 is computing the message last block (STOP and FIN are true), no new message must be transmitted (FIN remains true).


- Assume \((AG \ (RESET \_ \ !FIN))\): this condition avoids to detect a false error, forcing signal FIN to be false during a reset. Indeed, a property requires to make sure that if FIN is true, signal DIGEST_OK eventually becomes true. Without this condition, such a property will be detected as false, because the SHA-1 cannot raise signal DIGEST_OK to true without having previously recovered a message.

B. Properties to verify

Here are some examples of properties (cf appendix 3 and 4 for all the properties) expressed and checked with FormalCheck and RuleBase:

- When all the contents of the message have been transmitted (FIN becomes true), we must observe the internal signal requiring that the hash process begins its work (TRAITER becomes true):

  FormalCheck:

  Liveness - eventually
  
  \[\text{AFTER} \quad \text{Fin} = \text{rising} \]
  \[\text{EVENTUALLY} \quad \text{Traiter} = 1 \]
  \[\text{UNLESS} \]

  RuleBase:

  \[\text{always}( \neg RESET \& FIN) \rightarrow (AF(\neg RESET \& \text{TRAITER}))\]

- If a message indicating the availability of a digest is observed (DIGEST_OK becomes true), this implies that we are not transmitting message (FIN is false):

  FormalCheck:

  Safety - Always
  
  \[\text{AFTER} \]
  \[\text{ALWAYS} \quad (\text{Digest_OK} = 1) \Rightarrow (\text{Fin} = 1) \]
  \[\text{UNLESS} \]

  RuleBase:

  \[\text{always}(\neg RESET \& \text{DIGEST_OK}) \rightarrow (\neg RESET \& FIN))\]

- If the padding process is in the reset state (RaZ, i.e. Etape = 9), the second process no longer requires the data block (Fin_Phase1 is true):

  FormalCheck:

  Safety - Always
  
  \[\text{AFTER} \]
  \[\text{ALWAYS} \quad (\text{Etape} = 9) \Rightarrow (\text{Fin_Phase1} = 1) \]
  \[\text{UNLESS} \]

  RuleBase:

  \[\text{Always}(\neg RESET \& \text{LETAP}(3) \& \text{LETAP}(2) \& \text{LETAP}(1) \& \text{LETAP}(1))\]
\[ LETAP(0) \rightarrow (!RESET \& \ FIN\_PHASE1) \]

It can be noted that FormalCheck allows to directly use the value of the variable that holds the automaton state. In our case, \textit{Etape} and \textit{Etat} are such variables for the first and second process. The automata states in the checking tools, were represented by natural integer constants (cf Appendix 7).

In a Sugar formula, the vector \text{LETAP}(3..0) stands for this variable \textit{etape}. In RuleBase, to represent this integer value, we had to use a bit vector.

The syntax we used to write these formulas in Sugar differs from the syntax we presented previously. This is due to the RuleBase version we had, which does not support Sugar2.0. An evolution towards a more powerful version will be made available to academia soon.
C. Experimental results

The experiments were made on a SUN BLADE workstation, with a 500 MHz processor, and a 128 megabytes (Mb) memory.

The following tables contain the experimental data for checking the various properties on the model, on the two model checkers.

<table>
<thead>
<tr>
<th>Propriété</th>
<th>Temps de calcul en s.</th>
<th>Taille en Mb.</th>
<th>Résultat</th>
<th>Temps de calcul en s.</th>
<th>Taille en Mb.</th>
<th>Résultat</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIN TRAITER</td>
<td>374</td>
<td>1.78</td>
<td>Verified</td>
<td>4.25</td>
<td>41</td>
<td>Passed</td>
</tr>
<tr>
<td>NoDead</td>
<td>652</td>
<td>51.24</td>
<td>Verified</td>
<td>7.86</td>
<td>52</td>
<td>Passed</td>
</tr>
<tr>
<td>bonmoment</td>
<td>39</td>
<td>53.82</td>
<td>Verified</td>
<td>2.84</td>
<td>43</td>
<td>Passed</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td>Terminated</td>
<td>44.64</td>
<td>53</td>
<td>Passed</td>
</tr>
<tr>
<td>RaZ Phase</td>
<td>42</td>
<td>53.81</td>
<td>Verified</td>
<td>2.26</td>
<td>42</td>
<td>Passed</td>
</tr>
<tr>
<td>etatsFIN</td>
<td></td>
<td></td>
<td>Terminated</td>
<td>7.2</td>
<td>52</td>
<td>Passed</td>
</tr>
<tr>
<td>FIN_NOUV</td>
<td>40</td>
<td>53.81</td>
<td>Verified</td>
<td>6.02</td>
<td>48</td>
<td>Passed</td>
</tr>
<tr>
<td>StopEtat</td>
<td>160</td>
<td>1.79</td>
<td>Verified</td>
<td>2.49</td>
<td>42</td>
<td>Passed</td>
</tr>
<tr>
<td>StopTraite</td>
<td>286</td>
<td>48.52</td>
<td>Verified</td>
<td>7.66</td>
<td>45</td>
<td>Passed</td>
</tr>
<tr>
<td>TraiteFin</td>
<td>205</td>
<td>48.52</td>
<td>Verified</td>
<td>8.52</td>
<td>48</td>
<td>Passed</td>
</tr>
<tr>
<td>TraitBlok</td>
<td>153</td>
<td>48.54</td>
<td>Verified</td>
<td>14.48</td>
<td>52</td>
<td>Passed</td>
</tr>
<tr>
<td>forme</td>
<td>158</td>
<td>51.15</td>
<td>Verified</td>
<td>1.55</td>
<td>41</td>
<td>Passed</td>
</tr>
<tr>
<td>SBlok</td>
<td>278</td>
<td>51.18</td>
<td>Verified</td>
<td>2.82</td>
<td>43</td>
<td>Passed</td>
</tr>
<tr>
<td>bouge</td>
<td>170</td>
<td>48.54</td>
<td>Verified</td>
<td>8.93</td>
<td>52</td>
<td>Passed</td>
</tr>
<tr>
<td>lesFins</td>
<td>174</td>
<td>48.54</td>
<td>Verified</td>
<td>2.03</td>
<td>42</td>
<td>Passed</td>
</tr>
<tr>
<td>derniere</td>
<td>149</td>
<td>48.54</td>
<td>Verified</td>
<td>1.18</td>
<td>40</td>
<td>Passed</td>
</tr>
<tr>
<td>EtapTraite</td>
<td>155</td>
<td>48.54</td>
<td>Verified</td>
<td>4.1</td>
<td>49</td>
<td>Passed</td>
</tr>
<tr>
<td>StopEt6</td>
<td>159</td>
<td>53.67</td>
<td>Verified</td>
<td>5.53</td>
<td>51</td>
<td>Passed</td>
</tr>
</tbody>
</table>

Fig. 8: Results obtained on the reduced version of the model

It appears with this first test set that RuleBase, in its last official version (version 1.4), is much more powerful than the version of FormalCheck we have (version 2.3). Thus, if we look for example at the property “derniere”, IBM’s software is almost 127 times faster, and at the same time, it requires 40 Mb of memory against 48.54 Mb for FormalCheck. These differences are explained mainly by the fact that RuleBase uses several strategies to solve the state explosion problem (BDD, reduction of the model to the relevant subset for the formula considered, checking on-the-fly...).

In addition, two properties remain unsolved with the Cadence software, double and etatsFIN. The checking of these properties was not concluded because they need an extremely long computing time: the first one, for example, was manually interrupted after 14 hours. On the other hand, RuleBase manages to check it, but the computing time is relatively long compared to its performance on the other properties. The memory capacity needed (53 and 52 Mb) for those properties is among the largest for RuleBase too.
III. Verification using monitors

1. The FoCs tool

For the foreseeable future, industrial hardware design will continue to use both simulation and model checking in the design verification process. To date, these techniques are applied in isolation using different tools and methodologies, and different formulations of the problem. This results in cumulative high cost and little (if any) cross-leverage for simulation and formal verification.

With the goal to effectively and advantageously exploit the co-existence of simulation and model checking, the IBM research lab in Haifa developed a tool called FoCs (Formal Checkers). Implemented as an independent component of the RuleBase toolset, it takes RCTL (subset of CTL, plus SERE), and translates them in VHDL programs (monitors), which are added to the simulation environment.

This is not a new concept: manually-written checkers are a traditional part of simulation environments. Checkers facilitate massive random testing, because they automate test results analysis. However, the manual writing and maintenance of checkers is a notoriously high-cost and labor-intensive effort, especially if the properties to be verified are complex temporal ones.

FoCs generate these monitors automatically (see an example of monitor in appendix 5), starting from formal specifications. This is a state machine which enters in an error state during simulation, if the formula is false during one cycle.

![FoCs Environment Diagram](image)

**Fig. 9: FoCs environment**

The verification effort is reduced: the same formal properties written for model checking of small design blocks can be reused for simulation analysis across all higher simulation levels.
During this project, the simulation was performed before FoCs was made available to us. Nevertheless, it was the origin of a thought on hardware verification using monitors.

The idea is to use Sugar expressiveness to automatically produce monitors. The model-checking tool is then used to check each property on the parallel composition of the circuit model and its monitor, only verifying that this monitor never enters an error state. Therefore the verification possibilities offered by model checking on temporal logic properties are eased thanks to Sugar expressiveness.

2. Properties

With the monitors checking the expressed properties, the tool must show that an error state is never entered. Thus, for property "bonmoment", which requires that if signal DIGEST_OK is true, then FIN is true (end of digestion of a message, only if this one is finished), the property we express is (in Sugar syntax):

$$\text{NEVER } (\text{not bonmoment})$$

We check the model formed by the basic model of the circuit, increased with the VHDL code of the considered monitor (cf appendix 5 and 6). The monitor automatically generated by FoCs (in the form of a VHDL process) is manually modified in the following way: the instruction “assert” that it contains is replaced by an assignment with an architecture global signal, which bears the name of the considered property (identifier following the key word “rule” in the Sugar property).
IV. Experimental results analysis

With Sugar, RuleBase has got a formalism allowing to express complex properties in a relatively easy way, as we saw previously.

The results show that RuleBase is much more powerful than the version of FormalCheck we have. However its use is much more difficult because we loose the names of circuit internal signals, such as they were declared in the VHDL source code. A way to avoid this difficulty is to declare the signals we wish to observe as outputs, but this needs a manual modification of the initial description to express the properties. From this point of view, the Cadence software is more user friendly because all circuit internal signals are directly accessible by their VHDL identifier using the user interface.

The use of monitors makes it possible to check properties expressed in Sugar, while taking advantage of this friendly user interface. However, the version of FoCs we used does not support all the richness of Sugar. Expressions such as “AF” or “not” are not implemented, or not for all the possible cases. The formulation of properties using them must be done by diverted means, for example by fixing the interval of time over which the AF have to be checked. The problem comes from the size of the generated monitor, which can double the size of the original model.

The approach model + monitor was tested with RuleBase, and FormalCheck. In both cases, the computing time is increased significantly.
Conclusion

Thanks to the requirements of industry, automatic verification techniques are largely used. However, their use may be complex, and taking full benefit of all their capabilities requires a significant experience. This is why any approach simplifying their use must be taken into account.

This study allowed me to discover the field of coding algorithms for safety problems, studying one of them, the SHA-1, and also to apply several verification techniques to a circuit behavioral specification. The main difficulties that were encountered are due to the modeling of the constraints and the environment.

This study mainly contributed to explore another model checking approach, which aims at accelerating the verification process of a circuit. The monitors, very useful during the circuit simulation phase, that are produced using Sugar and the automatic generator FoCs, can be re-used, possibly with minor modifications, for circuit formal verification. The ease of use of Sugar and FoCs make it possible to express properties in a clear and precise way. Then, the relatively intuitive interface of a tool such as FormalCheck eases the identification of which property should be checked.

There is no doubt that Sugar greatly improves the ease of writing and the readability of properties. FoCs makes it possible to use the model, extended with the monitor, with other model-checkers and thus to enrich the properties they are able to check. However, we could not fully experiment the power of this approach because we are, for the moment, limited by the Sugar subset accepted by the FoCs version we have.
Bibliographical sources

Aout 1999

systems using temporal logic specifications”, ACM Transaction on programming language and systems, 8(2):
244-263, avril 1986

Specification Language by mechanised theorem proving.”, Soumis à CHARME, n°34

simulation checkers from formal specifications”, Proceedings CAV’00, LNCS 1855, Springer Verlag

[5] Laboratoire de recherche IBM, Haifa : “FoCs, Formal Checkers- a productivity tool”
Formal Methods and technologies Group, Vs 1.0, avril 2003,

Version 2.3”, Hardware Verification Group, Concordia University, Montreal,

systems”, Antalya Turkey, 1997

Accelera Version 1.0, 31 janvier 2003

Version 1.4.3 Verification Technologies Group, Janvier 2003

Y. Rodeh, G. Ronin et Y. Wolfshal : “RuleBase: model checking at IBM”, IBM Haifa Research
Laboratory, Israel

Federal Information Processing Standards Publication 180-1, 17 avril 1995

Proceedings Int. Symp. On programming, Turin Italy, avril 1982
Vol. 137 de lectures notes in computer science, pages 337-351, springer 1982

IBM Haifa Research Laboratory, Israel

Sugar”, IBM Haifa Research Laboratory, Israel

[16] David Deharbe :“Vérification formelle de propriétés temporelles: étude et applications au langage VHDL”, THESE de doctorat soutenue le 15 novembre 1996, Laboratoire TIMA (VDS)


Appendix

Appendix 1: Scheme of SHA-1, basic version

Appendix 2: Source code of SHA-1, soft version

Appendix 3: Properties checked with RuleBase

Appendix 4: Properties checked with FormalCheck

Appendix 5: A monitor generated by FoCs

Appendix 6: State encoding
Appendix 1:

Schematic structure of SHA-1, basic version
Appendix 2:

Source code of SHA-1, soft version
The SHA-1 algorithm

Auteur: CHAVET Cyrille
Date: april 11, 2003
Version: 3.5
Descriptif: Light version of the SHA 1

This algorithm is the same that the version 3.0, but the data path has been reduced to enable the verification with our tools. The block is reduced to 6 bits, with the last 2 bits for the message length. IN/Out bus are reduced to 1 bit, All the functions to work with 1 bit datas States of the automatas are replaced by integer constants

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
--use IEEE.numeric_bit.all;
-- ********** State management packs
--use lib_VHDL.pack.all;
--use work.pack.all;

entity sha1 is port(
  DATA : in std_logic;   -- Data bus reduced to 1 bit
  TOK : in bit;          -- Data on the bus
  FIN : in bit;          -- End of data transfer
  VALID : in UNSIGNED(2 downto 0); -- Number of valids -- bits on the bus, not so important here!
  DIGEST: out std_logic; -- Digest, reduced to 1 bit
  DIGEST_OK: out bit;    -- Digest ready
  STOP : out bit;        -- Work in progress, do not send message
  RESET: in bit          -- Reset signal
);
end sha1;

architecture naif of sha1 is

begin

-- SHA-1 encryption algorithm

-- Tools: function and procedure

-- == function 0 a 19 => Reduced to 1 bit

function log_fun0(B,C,D: std_logic) return std_logic is
variable result : std_logic;
begin
    result := ((B and C) or ((not B) and D));
    return result;
end log_fun0;

-- ===============
-- == function 20 a 39

function log_fun1(B,C,D: std_logic) return std_logic is
    variable result : std_logic;
begin
    result := (B xor C xor D);
    return result;
end log_fun1;

-- ===============
-- == function 40 a 59

function log_fun2(B,C,D: std_logic) return std_logic is
    variable result : std_logic;
begin
    result := ((B and C) or (B and D) or (C and D));
    return result;
end log_fun2;

-- ===============
-- == function 60 a 79

function log_fun3(B,C,D: std_logic) return std_logic is
    variable result : std_logic;
begin
    result := (B xor C xor D);
    return result;
end log_fun3;

-- ===============
-- == Rotate function => useful for verification because
-- ==                                    the operand rotate_left was not
-- ==                                    supported by our checking tools

function rotate(Vect: UNSIGNED(1 downto 0); Nbre: natural) return UNSIGNED is
begin
    -- Impossible if Nbre = 0
    return (Vect((1-Nbre) downto 0) & Vect(1 downto (2-Nbre)));
end rotate;

-- ===============
-- Variables and constants declarations
-- ===============

-- == Data saving and padding
signal ecrase: bit := '0'; -- Message crush block[62 and/or 63], reserved the the length of the message
signal traiter: bit := '0'; -- Send a block to the hash process
signal S_blok_null: bit := '0';-- Empty block (block null) to hash
signal S_blok_null_un: bit := '0';-- Block null, with a 1 as first bit, to hash
signal traite_fin: bit := '1'; -- Block hashed
signal stop_sig: bit := '0'; -- Stop sending message to SHA1
signal Fin_phase1: bit := '0'; -- End of copy of block in the table Tab_blok
signal S_OK: bit := '0'; -- End of hashing block null, or null with a 1

-- == Work in progress signals
signal busy1, busy_ended: bit := '0'; -- Work in progress ...

-- =======================
-- == Sampling of the digest on the bus

--type type_donne is (rien, H01, H02, H11, H12, H21, H22,
-- H31, H32, H41, H42, lafin);
--signal Etat_donne: type_donne := rien; -- Current state

signal Fin_Delivre: bit := '0'; -- The digest has been entirely transfered
signal Termine: bit := '0'; -- End of hashing

-- =======================
-- == Total length of the message => reduced to 2 bits

signal longueur: UNSIGNED(1 downto 0) := (others => '0');

-- =======================
-- == Signal blok = data storage (from the bus DATA) => the block size is
reduced to 6

constant max: UNSIGNED(3 downto 0) := "0110";
signal Blok: UNSIGNED(0 to 5);

-- =======================
-- == Digest computation

type Type_Tab_Blok is array (0 to 7) of UNSIGNED(1 downto 0);
signal Tab_Blok: Type_Tab_Blok;

-- == States of hashing automata
--type type_etat is (bronzette, et0, et1, et2, et3,
-- et30, et4, et5, et6, thefin);
--signal etat: type_etat := bronzette; -- current state

-- == States of data saving and block padding automata
--type type_etat2 is (didle, No_Problemo, Ecraser, Depasse, Trans1, Forme1,
-- Trans2, Forme2, Traite, RaZ, Happy_End, Trans12, Trans22);
--signal etape: type_etat2 := didle; -- current state

-- == Variables needed by the hash process => Reduced to 1 bit
signal H0: std_logic := '0';
signal H1: std_logic := '1';
signal H2: std_logic := '1';
signal H3: std_logic := '0';
signal H4: std_logic := '1';
signal A : std_logic := '0';
signal B : std_logic := '0';
signal C : std_logic := '0';
signal D : std_logic := '0';
signal E : std_logic := '0';
signal Temp : std_logic := '0';

-- == Constants => reduced to 1 bit
-- X"5A827999"; Constant 0 <= t <= 19
constant K0_19 : std_logic := '0';
-- X"6ED9EBA1"; Constant 20 <= t <= 39
constant K20_39 : std_logic := '0';
-- X"8F1BBCDC"; Constant 40 <= t <= 59
constant K40_59 : std_logic := '1';
-- X"CA62C1D6"; Constant 60 <= t <= 79
constant K60_79 : std_logic := '1';

-- ======== States encoding for verification
constant cdidle : natural := 0;
constant cNo_Problemo : natural := 1;
constant cEcraser : natural := 2;
constant cDepasse : natural := 3;
constant cTrans1 : natural := 4;
constant cForme1 : natural := 5;
constant cTrans2 : natural := 6;
constant cForme2 : natural := 7;
constant cTraite : natural := 8;
constant cRaZ : natural := 9;
constant cHappy_End : natural := 10;
constant cTrans12 : natural := 11;
constant cTrans22 : natural := 12;
constant cbronzette : natural := 0;
constant cet0 : natural := 1;
constant cet1 : natural := 2;
constant cet2 : natural := 3;
constant cet3 : natural := 4;
constant cet30 : natural := 5;
constant cet4 : natural := 6;
constant cet5 : natural := 7;
constant cet6 : natural := 8;
constant cthefin : natural := 9;
constant cattend : natural := 10;
constant crien : natural := 0;
constant cH01 : natural := 1;
constant cH02 : natural := 3;
constant cH11 : natural := 4;
constant cH12 : natural := 5;
constant cH21 : natural := 6;
constant cH22 : natural := 7;
constant cH31 : natural := 8;
constant cH32 : natural := 9;
constant cH41 : natural := 10;
constant cH42 : natural := 11;
constant clafin : natural := 12;
-- =========
begin
  -- SHA1 algorithm Vs. 3.5

  -- End of transmition waiting for the end of the hash process
  STOP <= (stop_sig or (busy1 and (not busy_ended)));

  -- Data saving and padding process

  RECUP_PADDING: process (TOK, RESET)
  variable NbBlok : UNSIGNED(3 downto 0);  -- Block crossing index
  variable NbStep : UNSIGNED(3 downto 0);  -- Number of blocks in the message
  variable long, NbValid : UNSIGNED(3 downto 0);  -- Message length and number of
        -- bits in the message
  variable etape : natural;                -- Current state of the automata

  begin
    if (RESET = '1') then
      ecrase <= '0';
      traiter <= '0';
      stop_sig <= '0';
      busy1 <= '0';
      NbStep := (others => '0');
      NbBlok := (others => '0');
      NbValid := (others => '0');
      long := (others => '0');
      Blok <= (others => '0');
      etape := cdidle;
    elsif (TOK'event and TOK = '1') then
      case etape is
        when cdidle =>
          stop_sig <= '0';        -- Do not stop data transfert
          traiter <= '0';         -- Do not hash any block
          Sblok_null <= '0';      -- No block null required
          Sblok_null_un <= '0';
          NbValid(3 downto 0) :=
            to_unsigned((to_integer(VALID) +1), 4);
          NbBlok := (others => '0');
          busy1 <= '0';
          if (FIN = '1') then
            -- No message ...
            etape := cHappy_end;
          else
            if (NbValid = 1) then
              etc
-- No incoming message
    etape := cdidle;
else
    -- First bit of the message on the bus
    Blok(0) <= data;
    NbBlok := NbBlok + 1;
    etape := cNo_problemo;
end if;

equipe := ctrans1;
when cNo_Problemo =>
    if (FIN = '0' and NbBlok = 4) then
      -- The current word on the bus will be store on the reserved area
      -- ie, the last 2 bits of the block (reserved for the length of
      -- the message)
        etape := cEcraser;
        else
        if (FIN = '1') then
          -- End of message (message entirely tranfered)
            busy1 <= '1';
            etape := cTrans1;
        else
          -- No problem
            NbValid(3 downto 0) := to_unsigned(to_integer(VALID + 1), 4);
            Blok(to_integer(NbBlok)) <= data;
            NbBlok := NbBlok + 1;
            etape := cNo_problemo;
        end if;
    end if;
when cEcraser =>
    NbValid(3 downto 0) :=
        to_unsigned(to_integer(VALID + 1), 4);
    Blok(to_integer(NbBlok)) <= data;
    NbBlok := NbBlok + 1;
    if (FIN = '0' and NbBlok = max) then
      -- Overflow, the message is too long for one block
      etape := cDepasse;
    else
      if (FIN = '1') then
        -- End of message
          busy1 <= '1';
          etape := cTrans2;
        else
          -- Stay in this state
            etape := cEcraser;
        end if;
    end if;
when cDepasse =>
    Stop_Sig <= '1';
    NbStep := NbStep + 1;
    Traiter <= '1';

etape := cTraite;

when cTrans1 =>
    NbBlok := NbBlok - 1;
    etape := cTrans12;

when cTrans12 =>
    long := NbBlok + NbStep + NbValid;
    longueur(1 downto 0) <= long(1 downto 0);
    etape := cForme1;

when cForme1 =>
    if (NbBlok = 4) then
        -- The end bit will be on the reserved word
        traiter <= '1';
        Sblok_null <= '1';
        Blok(to_integer(NbBlok + NbValid)) <= '1';
    else
        -- No problem
        traiter <= '1';
        Sblok_null_un <= '1';
        Blok(to_integer(NbBlok + NbValid)) <= '1';
        Blok(4 to 5) <= longueur(1 downto 0);
    end if;
    etape := cTraite;

when cTrans2 =>
    NbBlok := NbBlok - 1;
    etape := cTrans22;

when cTrans22 =>
    long := NbBlok + NbStep + NbValid;
    longueur(1 downto 0) <= long(1 downto 0);
    etape := cForme2;

when cForme2 =>
    if (NbBlok = 4) then
        -- Not enough place in this block for the end bit
        traiter <= '1';
        Sblok_null_un <= '1';
    else
        -- Only one bit free, for the end bit
        traiter <= '1';
        Sblok_null <= '1';
        Blok(to_integer(NbBlok + NbValid)) <= '1';
    end if;
    etape := cTraite;

when cTraite =>
    if (Fin_Phase1 = '1') then
        etape := cRaZ;
    else
        etape := cTraite;
    end if;

when cRaZ =>
    if (FIN = '1') then
        busy1 <= '1';
        etape := cHappy_End;
else
  if (traite_fin = '1') then
    etape := cDidle;
  else
    Blok <= (others => '0');
    etape := cRaZ;
  end if;
end if;

when cHappy_End => -- That's all folks
  if (FIN = '0') then
    -- Incoming new message ...
    Blok <= (others => '0');
    NbStep := (others => '0');
    long := (others => '0');
    NbValid := (others => '0');
    traiter <= '0';
    busy1 <= '0';
    etape := cDidle;
  else
    etape := cHappy_end;
  end if;

when others =>
  etape := cDidle;
end case;
end if;
end process;

-- =========================
-- ==  Hash process

DIGESTION: process (TOK, RESET)

variable step, t : UNSIGNED(2 downto 0); -- etapes de calcul
variable etat : natural;     -- etat de l'automate

begin
  if (RESET = '1') then
    t := "000";
    step := "000";
    S_OK <= '0';
    busyended <= '0';
    etat := cbronzette;
    traite_fin <= '1';
    Fin_PhaseI <= '0';
    Termine <= '0';
    une_fois <= '0';
    temp <= '0';
    Tab_Blok(0) <= "00";
    Tab_Blok(1) <= "00";
    Tab_Blok(2) <= "00";
    Tab_Blok(3) <= "00";
    Tab_Blok(4) <= "00";
    Tab_Blok(5) <= "00";
  end if;
elsif (TOK'event and TOK = '1') then
    if (traiter = '1') then
        -- Let's hash the saved block

        case etat is
            when cbronzette =>
                busy Ended <= '0';
                traite fin <= '0';  -- We are hashing a block,
                -- do not send any more message
                etat := cet0;
            when cet0 =>
                Tab_Blok(0) <= Blok(0 to 1);  -- Copy of the block in
                Tab_Blok(1) <= Blok(2 to 3);
                Tab_Blok(2) <= Blok(4 to 5);
                Fin_Phase1 <= '1';  -- Reset of block in the
                -- first process
                etat := cet1;
            when cet1 =>
                if (step < 7) then
                    Tab_Blok(to_integer(step)) <= rotate(
                        ( Tab_Blok(to_integer(step - 3)) xor
                        Tab_Blok(1) xor Tab_Blok(0) xor
                        Tab_Blok(0)), 1);
                    step := step + 1;
                else
                    step := "000";
                    etat := cet2;
                end if;
            when cet2 =>
                A <= H0;
                B <= H1;
                C <= H2;
                D <= H3;
                E <= H4;
                etat := cet3;
            when cet3 =>
                if (t < 1) then
                    Temp <= log fun0(B, C, D);
                    etat := cet30;
                elsif (t < 2) then
                    Temp <= log fun1(B, C, D);
                    etat := cet30;
                elsif (t < 3) then
                    Temp <= Tab_Blok(5)(1);
                    etat := cet30;
                elsif (t < 4) then
                    Temp <= K60_79;
                    etat := cet30;
                else
                    t := "000";
                    etat := cet4;
end if;

when cet30 =>
  E <= D;
  D <= C;
  C <= B;
  B <= A;
  A <= Temp;
  t := t+1;
  etat := cet3;

when cet4 =>
  H0 <= H0 or A;  -- or instead of  +
  H1 <= H1 or B;
  H2 <= H2 or C;
  H3 <= H3 or D;
  H4 <= H4 or E;
  t := "000";
  etat := cet5;

when cet5 =>
  if (Sblok_null = '1' and S_OK = '0') then
    Tab_Blok(0) <= "00";
    Tab_Blok(1) <= "00";
    Tab_Blok(2) <= longueur(1 downto 0);
    Tab_Blok(3) <= "00";
    Tab_Blok(4) <= "00";
    Tab_Blok(5) <= "00";
    Tab_Blok(6) <= "00";
    Tab_Blok(7) <= "00";
    S_OK <= '1';
    etat := cet1;
  elsif (Sblok_null_un = '1' and S_OK = '0') then
    Tab_Blok(0) <= "10";
    Tab_Blok(1) <= "00";
    Tab_Blok(2) <= longueur(1 downto 0);
    Tab_Blok(3) <= "00";
    Tab_Blok(4) <= "00";
    Tab_Blok(5) <= "00";
    Tab_Blok(6) <= "00";
    Tab_Blok(7) <= "00";
    S_OK <= '1';
    etat := cet1;
  else
    -- Here there are two ways:
    -- The message is finished, so we can sample the digest
    -- The message is not finished, the current result are saved

    if (FIN = '1') then
      -- End of message

      Termine <= '1';
      etat := cattend;
    else
      -- Deal with the remainder of the message

      Traite_fin <= '1';
      etat := cet6;
    end if;
  end if;
when cet6 =>
etat := cbronnette;

when cattend =>
  if (Fin_Delivre = '1') then
    Termine <= '0';
    busy_ended <= '1';
    etat <= cthefin;
  else
    etat <= cattend;
  end if;
when cthefin =>
  if (FIN = '1') then
    etat := cthefin;
  else
    -- New message to hash
    t := "000";
    step := "000";
    S_OK <= '0';
    DIGEST_OK <= '0';
    busy_ended <= '0';
    Traite_fin <= '1';
    Fin_Phase1 <= '0';
    etat := cbronnette;
    temp <= '0';
    Tab_Block(0) <= "00";
    Tab_Block(1) <= "00";
    Tab_Block(2) <= "00";
    Tab_Block(3) <= "00";
    Tab_Block(4) <= "00";
    Tab_Block(5) <= "00";
    Tab_Block(6) <= "00";
    Tab_Block(7) <= "00";
    H0 <= '0';
    H1 <= '1';
    H2 <= '1';
    H3 <= '0';
    H4 <= '1';
  end if;
when others => etat := cbronnette;
end case;
else
  t := "000";
  step := "000";
  S_OK <= '0';
  DIGEST_OK <= '0';
  busy_ended <= '0';
  Traite_fin <= '1';
  Fin_Phase1 <= '0';
  etat := cbronnette;
  temp <= '0';
end if;
end if;
end process;
-- =============
-- == Result sampling

DELIVRER: process (TOK, RESET)
    variable Etat_donne : natural;  -- current state
begin
    if (RESET = '1') then
        Etat_donne := crien;
        Fin_Delivre <= '0';
        DIGEST_OK <= '0';
    elsif (TOK'event and TOK = '1') then
        if (Termine = '1') then
            -- The digest is sampled as 1 bit words on the bus DIGEST
            case Etat_donne is
                when crien =>
                    Fin_Delivre <= '0';
                    DIGEST_OK <= '1';
                    Etat_donne := cH01;
                when cH01 =>
                    DIGEST <= H0;
                    Etat_donne := cH02;
                when cH02 =>
                    DIGEST <= H0;
                    Etat_donne := cH11;
                when cH11 =>
                    DIGEST <= H1;
                    Etat_donne := cH12;
                when cH12 =>
                    DIGEST <= H1;
                    Etat_donne := cH21;
                when cH21 =>
                    DIGEST <= H2;
                    Etat_donne := cH22;
                when cH22 =>
                    DIGEST <= H2;
                    Etat_donne := cH31;
                when cH31 =>
                    DIGEST <= H3;
                    Etat_donne := cH32;
                when cH32 =>
                    DIGEST <= H3;
                    Etat_donne := cH41;
                when cH41 =>
DIGEST <= H4;
   Etat_donne := cH42;

when cH42 =>
  DIGEST <= H4;
  Etat_donne := clafin;

when clafin =>
  Fin_Delivre <= '1';
  DIGEST_OK <= '0';

when others =>
  Etat_donne := crien;

end case;
else
  Fin_Delivre <= '0';
  Etat_donne := crien;
  DIGEST_OK <= '0';
  DIGEST <= '0';
end if;
end if;
end process;
end naif;
Appendix 3:

Properties checked with RuleBase
Properties for RuleBase

Here are the SHA-1 properties as they were checked with RuleBase.

rule FIN_TRAITER{
"After the end of the message, the process must ask its hashing."

formula
"After Fin = 1, Traiter is 1"
{ AG(!(RESET & FIN) | (AF(RESET & TRAITER)))
}
}

rule NoDead{
"After the end of the message, one can observe the end of hashing signal."

formula
"After Fin = 1, Digest_OK is 1"
{ AG( (!RESET & rose(FIN)) -> (AF(!RESET & rose(DIGEST_OK)))
}
}

rule bonmoment{
"If signal DIGEST_OK is true, it means that the message is ended."

formula
"If digest_OK is true, so Fin must be true."
{ AG(! (!RESET & DIGEST_OK) | (!RESET & FIN))
}
}

rule double{
"If a block null, or a block null with a 1 as first bit, is being requiered, the hash procees is going to be in the state et1."

formula
"Block null, or block null and 1, implie that the hash process will be for a moment in state et1."
{ AG(! (!RESET & (SBLOK_NULL | SBLOK_NULL_UN)) | (ABF[1..200](!RESET & !LETA(3) & !LETA(2) & LETA(1) & !LETA(0)))
}
}

rule RaZ_Phase{
"If the saving and padding process is in the state RaZ, that means that FIN_PHASE1 is true."

formula
"RaZ => Fin_Phase1"
{ AG(! (!RESET & LETAP(3) & !LETAP(2) & !LETAP(1) & LETAP(0))
 | (!RESET & FIN_PHASE1)
}
}
rule etatsFIN{
"If the signal FIN is True, each process will enter in their final state."

formula
"FIN => AF (etat = thefin and etape = HappyEnd)"
{ AG(! (!RESET & FIN)
    | (AF (!RESET & LETAP(3) & !LETAP(2) & LETAP(1) & !LETAP(0) &
        LETA(3) & !LETA(2) & !LETA(1) & LETA(0)) )
    )
    )
}
}

rule FIN_NOUV{
"When FIN becomes true, the process will be in the state thefin."

formula
"FIN => AF (etat = thefin)"
{ AG(! (!RESET & FIN)
    | (AF (!RESET & LETA(3) & !LETA(2) & !LETA(1) & LETA(0)) )
    )
    )
}
}

rule StopEtat{
"While the signal stop_sig is true, the first two processes can enter in their final state."

formula
"Stop_sig => (etat != thefin) and (etape != Happy_End)"
{ AG(! (!RESET & STOP_SIG & ( (LETAP(3) & !LETAP(2) & LETAP(1) & !LETAP(0)) | (LETA(3) & !LETA(2) & !LETA(1) & LETA(0)) )
    )
    )
}
}

rule StopTraite{
"While the signal STOP_SIG is true, the signal TRAITER will be true."

formula
"Stop_sig => Traiter"
{ AG(! (!RESET & STOP_SIG)
    | (AF (!RESET & TRAITER) )
    )
    )
}
}

rule TraiteFin{
"If the first process is stopped (the block is full, so STOP_SIG becomes true), so TRAITER_FIN must become true."

formula
"Stop_sig => Traite_Fin"
{ AG(! (!RESET & STOP_SIG)
    | (AF (!RESET & TRAITE_FIN) )
    )
    )
}
rule TraitBlok{
"The hashing of a block is finished (TRAITE_FIN becomes true), either the
message is not finished (FIN is false) and the first process is found in the
initial state, or the message is finished (FIN is true) and in this case, the
first process is found in its final state"

formula
"TRAITE_FIN => (Fin and etape = Happy_End) or (not Fin and etape = didle)"
{ AG(! (rose(TRAITE_FIN))
  | AF( (FIN & !RESET & LETAP(3) & !LETAP(2) & LETAP(1) & !LETAP(0)) | 
    (!FIN & !RESET & !LETAP(3) & !LETAP(2) & !LETAP(1) & !LETAP(0)) )
  )
}
}

rule Forme{
"If the first process is in state FORME1 or FORME2, that means that the
message is finished."

formula
"((etape = forme1) or (etape = 2)) => FIN"
{ AG(! (!RESET & ( (!LETAP(3) & LETAP(2) & !LETAP(1) & LETAP(0))
   | (!LETAP(3) & LETAP(2) & LETAP(1) & LETAP(0)) ))
   | (FIN)
  )
}
}

rule SBlok{
"It is impossible for the signal FIN to be true when one the signals
SBLOK_NULL or SBLOCK_NULL_UN are true."

formula
"Never: not FIN and (SBLOK_NULL or SBLOCK_NULL_UN)"
{ AG(! (!RESET & !FIN & (SBLOK_NULL | SBLOK_NULL_UN))
  )
}
}

rule bouge{
"When a message is coming, the first process will be in state No_Problemo"

formula
"not FIN => AF (etape = No_Problemo) "
{ AG( (!RESET & (fell(FIN))) ->
  (ABF[1..10] (!LETAP(3) & !LETAP(2) & !LETAP(1) & LETAP(0)) )
  )
}
}

rule lesFins{
"There is mutual exclusion between the signals FIN and STOP_SIG."

formula
"never (FIN and Stop_sig)"

rule derniere{
"After the request for hashing a block, one hope for an end of hash signal."

formula
"Traiter => AF Traite_Fin"
{ AG( (!RESET & TRAITER) -> (AF (!RESET & TRAITE_FIN)) ) }
}

rule EtapTraite{
"If there is a message, the first process will pass by state Traite."

formula
"If there is a message, it will be treated"
{ AG( (!RESET & rose(FIN)) | (AF ((!RESET & LETAP(3) & !LETAP(2) & !LETAP(1) & !LETAP(0))) ) ) }
}

rule StopEt6{
"If the hashing process will be in state et6, that means that the data transfer will be stopped."

formula
"(Etat = et6) => Stop"
{ AG(! (RESET & LETA(3) & !LETA(2) & !LETA(1) & !LETA(0)) | (!RESET & STOP) ) }
}
Appendix 4:

Properties checked with FormalCheck
Properties for FormalCheck

Here are the properties checked with FormalCheck. They are the same as those checked with RuleBase:

**FIN_TRAITER**
"After the end of the message, the process must ask its hashing."

Liveness - EVENTUALLY
AFTER Fin = rising
EVENTUALLY Traiter = 1
UNLESS

NoDead
"After the end of the message, one can observe the end of hashing signal"

Liveness - EVENTUALLY
AFTER Fin = rising
EVENTUALLY Digest_OK = 1
UNLESS

bonmoment
"If signal DIGEST_OK is true, it means that the message is ended"

Safety - ALWAYS
AFTER ALWAYS (Digest_OK = 1) => (Fin = 1)
UNLESS

double
"If a block null, or a block null with a 1 as first bit, is being required, the hash process is going to be in the state et1"

Liveness - EVENTUALLY
AFTER (Sblok_Null = 1) or (Sblok_Null_Un = 1)
EVENTUALLY Etat = 2
UNLESS

RaZ_Phase
"If the saving and padding process is in the state RaZ, that means that FIN_PHASE1 is true."

Safety - ALWAYS
AFTER ALWAYS (Etape = 9) => (Fin_Phase1 = 1)
UNLESS

etatsFIN
"If the signal FIN is True, each process will in their final state."

Liveness - EVENTUALLY
AFTER Fin = 1
EVENTUALLY (Etat = 9) et (Etape = 10)
UNLESS
FIN_NOUV
"When FIN becomes true, the process will be in the state thefin"

Safety - ALWAYS
AFTER
ALWAYS (Fin = 1)
UNLESS

StopEtat
"While the signal stop_sig is true, the first two processes can enter in their final state."

Safety - ALWAYS
AFTER
ALWAYS (Stop_Sig = 1) => (Etape /= 10 et Etat /= 9)
UNLESS

StopTraite
"While the signal STOP_SIG is true, the signal TRAITER will be true."

Safety - ALWAYS
AFTER
ALWAYS (Stop_Sig = rising) => (Traiter = rising)
UNLESS

TraiteFin
"If the first process is stopped (the block is full, so STOP_SIG becomes true), so TRAITER_FIN must become true"

Liveness - EVENTUALLY
AFTER
EVENTUALLY Traite_Fin = 1
UNLESS

TraitBlok
"The hashing of a block is finished (TRAITE_FIN becomes true), either the message is not finished (FIN is false) and the first process is found in the initial state, or the message is finished (FIN is true) and in this case, the first process is found in its final state"

Liveness - EVENTUALLY
AFTER
EVENTUALLY (Fin = 0 et Etape = 0) ou (Fin = 1 et Etape = 10)
UNLESS

forme
"If the first process is in state FORME1 or FORME2, that means that the message is finished"

Safety - ALWAYS
AFTER
ALWAYS (Etape = 7 ou Etape = 5) => (Fin = 1)
UNLESS
SBlok

"It is impossible for the signal FIN to be true when one the signals
SBLOCK_NULL or SBLOCK_NULL_UN are true."

Safety - Never
AFTER
NEVER Fin = 0 et (Sblok_Null = 1 ou Sblok_Null_Un = 1)
UNLESS

bouge

"When a message is coming, the first process will be in state No_Problemo"

Liveness - EVENTUALLY
AFTER Fin = 0
EVENTUALLY Etape = 1
UNLESS

lesFins

"There is mutual exclusion between the signals FIN and STOP_SIG."

Safety - Never
AFTER
NEVER Stop_Sig = 1 et Fin = 1
UNLESS

derniere

"After the request for hashing a block, one hope for a end of hash signal"

Liveness - EVENTUALLY
AFTER Traiter = 1
EVENTUALLY Traite_Fin = 1
UNLESS

EtapTraite

"If there is a message, the first process will pass by state Traite"

Liveness - EVENTUALLY
AFTER Fin = 0
EVENTUALLY Etape = 8
UNLESS

StopEt6

"If the hashing process will be in state et6, that means that the data transfer
will be stopped."

Safety - ALWAYS
AFTER
ALWAYS (Etat = 8) => (Stop_Sig = 1)
UNLESS
Appendix 5 :

A monitor generated by FoCs
Here is a monitor generated by FoCs for the property dernièrè (the automatically generated instructions are in black):

RuleBase:

rule derniere{
"After the request for hashing a block, one hope for an end of hash signal."

formula
"Traiter => AF Traite_Fin"
{ AG( (!RESET & TRAITER) -> (AF (!RESET & TRAITE_FIN))
    }

}

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sha1 is port(
DATA : in std_logic;           -- Data bus
TOK : in std_logic;            -- Clock
FIN : in std_logic;            -- End data transfer
VALID : in UNSIGNED(2 downto 0);   -- Number of valid bits
DIGEST: out std_logic_vector(4 downto 0);   -- Digest
DIGEST_OK: out std_logic;          -- Digest available
STOP : out std_logic;           -- Work in progress,
                                  -- do not transfer data
RESET: in std_logic            -- Signal reset
);
end sha1;

architecture naif of sha1 is

-- ===================================================================
-- ========           Algorithme de cryptage SHA-1            ========
-- ===================================================================

-- MonitORS

function l2b (src: std_logic) return boolean is
variable R: boolean;
begnin
if (src = '1') then
    R := true;
else
    R := false;
end if;
return R;
end l2b;

    SIGNAL   focs_enable_derniere_1 : std_logic;
    SIGNAL   focs_v_derniere_1 : std_logic_vector(0 TO 4);
SIGNAL  Letap       : UNSIGNED (3 downto 0);  
SIGNAL  Leta        : UNSIGNED (3 downto 0);  
SIGNAL DIGEST_OK_SIG : std_logic;  
SIGNAL derniere     : boolean ;  

-- ==============================================================  
--                  END    MONITEURS  
-- ==============================================================

-- ==============================================================  
-- Fonctions et procedure outils  
-- ==============================================================

-- ===============
-- == Fonction 0 a 19
function log_fun0(B,C,D: std_logic) return std_logic is

  variable result : std_logic;
  begin
  result := ((B and C) or ((not B) and D));
  return result;
  end log_fun0;

-- ===============
-- == Fonction 20 a 39
function log_fun1(B,C,D: std_logic) return std_logic is

  variable result : std_logic;
  begin
  result := (B xor C xor D);
  return result;
  end log_fun1;

-- ===============
-- == Fonction 40 a 59
function log_fun2(B,C,D: std_logic) return std_logic is

  variable result : std_logic;
  begin
  result := ((B and C) or ( B and D) or (C and D));
  return result;
  end log_fun2;

-- ===============
-- == Fonction 60 a 79
function log_fun3(B,C,D: std_logic) return std_logic is

  variable result : std_logic;
  begin
  result := (B xor C xor D);
  return result;
  end log_fun3;

-- ===============
-- == Fonction rotation    POUR LA VERIF UNIQUEMENT
function rotate(Vect: UNSIGNED(1 downto 0); Nbre: natural) return UNSIGNED is
begin
  -- Impossible si Nbre = 0
  return (Vect((1-Nbre) downto 0) & Vect(1 downto (2-Nbre)));
end rotate;

-- ================================================================================
-- Declarations
-- ================================================================================

-- == Recup et mise en forme des donnees
signal ecrase : std_logic:= '0';       -- On ecrit dans l'espace longueur
signal traiter: std_logic := '0';       -- Il faut passer un bloc au SHA-1
signal Sblok_null: std_logic := '0';       -- Un bloc vide a traiter
signal Sblok_null_un: std_logic := '0'; -- Bloc vide + premier bit 1 a traiter
signal traite_fin: std_logic := '1';       -- Fin de traitement d'un blok
signal stop_sig: std_logic := '0';       -- Suspension envois message
signal Fin_phase1: std_logic := '0';       -- Fin recup du blok dans Tab_blok
signal S_OK : std_logic := '0';         -- Traitement Blok null ou null_un
signal busy1, busy_ended : std_logic := '0';       -- Work in progress ...

-- == Longueur totale du message
signal longueur : UNSIGNED(1 downto 0) := (others => '0');

-- == Type Type_Block <=> block de recup des donnees
constant max : UNSIGNED(3 downto 0) := "0110";
signal Blok : UNSIGNED(0 to 5);

-- == Mot pour calcul du digest
type Type_Tab_Blok is array (0 to 7) of UNSIGNED(1 downto 0);
signal Tab_Blok : Type_Tab_Blok;

-- == Variables de calcul du digest
-- X"67452301"
signal H0 : std_logic := '0';
-- X"EFCDAB89";
signal H1 : std_logic := '1';
-- X"98BADCFE"
signal H2 : std_logic := '1';
-- X"10325476"
signal H3 : std_logic := '0';
-- X"C3D2E1F0"
signal H4 : std_logic := '1';

signal A : std_logic := '0';
signal B : std_logic := '0';
signal C : std_logic := '0';
signal D : std_logic := '0';
signal E : std_logic := '0';
signal Temp : std_logic := '0';

-- == Constantes
-- X"5A827999";

MSc Thesis
TIMA-CMP groupe VDS, 2002/2003
-- Constante 0 <= t <= 19
constant K0_19 : std_logic := '0';
-- X"6ED9EBA1"
-- Constante 20 <= t <= 39
constant K20_39 : std_logic := '0';
-- X"8F1BBCDC"
-- Constante 40 <= t <= 59
constant K40_59 : std_logic := '1';
-- X"CA62C1D6"
-- Constante 60 <= t <= 79
constant K60_79 : std_logic := '1';

-- ========= POUR LA VERIFICATION
constant cdidle : natural := 0;
constant cNo_Problemo : natural := 1;
constant cEcraser : natural := 2;
constant cDepasse : natural := 3;
constant cTrans1 : natural := 4;
constant cTrans2 : natural := 5;
constant cForme1 : natural := 6;
constant cForme2 : natural := 7;
constant cTraite : natural := 8;
constant cRaZ : natural := 9;
constant cHappy_End : natural := 10;
constant cTrans12 : natural := 11;
constant cTrans22 : natural := 12;
constant cbronzette : natural := 0;
constant cet0 : natural := 1;
constant cet1 : natural := 2;
constant cet2 : natural := 3;
constant cet3 : natural := 4;
constant cet30 : natural := 5;
constant cet4 : natural := 6;
constant cet5 : natural := 7;
constant cet6 : natural := 8;
constant cthefin : natural := 9;

signal bip : std_logic := '0'; -- On est dans etat = cthefin
-- =========

begin

-- Connexion Hi <-> Digest
DIGEST(4) <= H0;
DIGEST(3) <= H1;
DIGEST(2) <= H2;
DIGEST(1) <= H3;
DIGEST(0) <= H4;

-- N.B.: C'est par le signal DIGEST_OK
-- que l'on signale que le digest est disponible
-- Suspension transmission ou attente fin de traitement
STOP <= (stop_sig or (busy1 and (not busy_ended)));

-- Process de recuperation et de mise en forme des donnees
RECUP_PADDING: process (TOK, RESET)

  variable NbBlok : UNSIGNED(3 downto 0);  -- Indice de parcours d'un block
  variable NbStep : UNSIGNED(3 downto 0);  -- Nbre de block (Blok)du message
  variable long, NbValid : UNSIGNED(3 downto 0);  -- Longueur du message
  -- et nbre de bits valides du message
  variable etape : natural;  -- Etat de l'automate de padding

begin
  [...]
end process;

-- Process de calcul de la signature
DIGESTION: process (TOK, RESET)

  variable step : UNSIGNED(2 downto 0);  -- etapes de calcul
  variable t : UNSIGNED(6 downto 0);
  variable etat : natural;  -- etat de l'automate

begin
  [...]
end process;

-- ==============================================================
-- MONITEURS
-- ==============================================================

PROCESS
  VARIABLE focs_ok_derniere_1 : std_logic;
  VARIABLE focs_num_of_fails_derniere_1 : INTEGER;
  VARIABLE focs_vout_derniere_1 : std_logic_vector(0 TO 4);
BEGIN
  WAIT UNTIL TOK'EVENT AND TOK = '1';
  IF ( l2b( RESET ) ) THEN
    derniere <= true;
    focs_enable_derniere_1 <= '1';
    focs_ok_derniere_1 := '1';
    focs_num_of_fails_derniere_1 := 1;
    focs_v_derniere_1(0 TO 4) <= "11111";
  ELSIF ( l2b( focs_enable_derniere_1 ) ) THEN
    focs_ok_derniere_1 := NOT(( focs_v_derniere_1(4) AND (NOT( reset )
                     AND NOT( traiter ) ) ));
    IF ( l2b( NOT( focs_ok_derniere_1 ) ) ) THEN
      focs_num_of_fails_derniere_1 := (focs_num_of_fails_derniere_1 - 1);
      IF ( ( focs_num_of_fails_derniere_1 = 0 ) ) THEN
        focs_enable_derniere_1 <= '0';
      END IF;
    END IF;
  END IF;
END IF;
Modelisation and validation of a chip embeded architecture for secure applications

MSc Thesis  
TIMA-CMP groupe VDS, 2002/2003

```vhdl
focs_vout_derniere_1(0 TO 4) := (( ( '0' AND '1' ) AND NOT( '0' AND NOT( '1' ) AND NOT( traiter ) ) ) AND ( NOT( reset ) AND NOT( traiter ) ) ) AND ( NOT( reset ) AND NOT( traite_fin ) ) ) AND ( NOT( reset ) AND NOT( traiter ) );
focs_vout_derniere_1(0 TO 4) <= ( ( focs_vout_derniere_1(0) AND ( focs_vout_derniere_1(0) OR focs_vout_derniere_1(1) ) ) AND ( ( focs_vout_derniere_1(0) OR focs_vout_derniere_1(1) ) OR focs_vout_derniere_1(3) ) ) AND ( ( focs_vout_derniere_1(0) OR focs_vout_derniere_1(1) ) OR focs_vout_derniere_1(3) ) );
derniere <= ( NOT( ( ( NOT( focs_enable_derniere_1 ) OR focs_ok_derniere_1 ) OR RESET ) ) )
END IF;
END PROCESS;
end naif;
```
Appendix 6:

State encoding
Integer values have been used for the state encoding.

**First process:**

- didle _0
- No_Problemo _1
- Ecraser _2
- Depasse _3
- Trans1 _4
- Forme1 _5
- Trans2 _6
- Forme2 _7
- Traite _8
- RaZ _9
- Happy_End _10
- Trans12 _11
- Trans22 _12

**Second process**

- Init _0
- et0 _1
- et1 _2
- et2 _3
- et3 _4
- et30 _5
- et4 _6
- et5 _7
- et6 _8
- thefin _9

*Fig. 10 : State encoding for the automata of the two main processes*