Automatic Generation of Validation Stimuli for Application-Specific Processors

O. Goloubeva, M. Sonza Reorda, M. Violante

Politecnico di Torino, Torino, Italy,
www.cad.polito.it

Abstract[*]

Microprocessor soft cores offer today an effective solution to the problem of rapidly developing new system-on-a-chips. However, all the features they offer are rarely used in embedded applications, and thus designers are often involved in the challenging task of soft-core customization to obtain application-specific processors. This paper proposes a novel approach to help designers in the simulation-based validation of application-specific processors. Suitable input stimuli are automatically generated while reasoning only on the software application the processor is intended to execute, while all the details concerning the processor hardware are neglected. Experimental results on a 8051 soft core show the effectiveness of the proposed approach.

1. Introduction

Thanks to the availability of deep sub-micron technologies, designers have now plenty of silicon area for their designs, up to the point that it is now common to find embedded systems integrated on a single chip that feature memory modules, processor cores and even embedded programmable logic modules.

To effectively support the design of such a kind of systems, which are known as System-on-Chips (SOCs), vendors started offering Intellectual Property- (IP-) cores ready for implementing complex tasks: for example microprocessor cores are now available ranging from simple controllers (like those implementing the Intel 8051 instruction set) to more complex pipelined processors (like those based on the SPARC v8 architecture). Being such a kind of IP-cores available, most of the work in SOC design consists in integrating different IP-cores: designers can indeed implement a SOC by properly customizing and connecting the needed IP-cores coming from potentially very different sources. For fostering this design approach, several attempts have been made to provide IP-cores with standard interfaces. The idea behind them is to simplify the communications among heterogeneous IPs during both normal operations (like for example the WISHBONE interface [1]) and test ones (like the IEEE P1500 standard).

Although silicon area is available in quantity, designers still face the need to minimize the size of their designs. Large area occupation still has several drawbacks: high manufacturing costs, low yield, high power consumption, just to mention a few of them. In an IP-core-based design flow, this implies the possibility of customizing the adopted IPs to make them implement just the needed features. IP customization is particularly efficient when the SOC is intended for being deployed in an embedded application. In this case all the IP-cores the SOC employs perform a very specific task that does not change during the SOC lifetime, and which is well defined since the beginning of the SOC design. Moreover, the customization is effective only when very complex IP-cores are considered, as in the case of processors. For example, if the designed SOC is using a processor IP whose architecture embeds a 32x32 parallel multiplier, but the embedded application it is aimed at does not require multiplication operations, then designers can save a huge amount of area by removing the multiplier from the IP.

Processor-core customization must be demanded to vendors when IPs come under the form of hard cores. In

[*] This work has been partially supported by the Italian Ministry for University through the project Reconfigurable platforms for wideband wireless communications (PRIMO).

this case, designers cannot modify the IP architecture and thus they have to fully rely on their suppliers. This may have dramatic impact on the IP cost, but it also greatly simplifies the task for the IP end users, which are freed from the very complex task of guaranteeing IP correctness. It is indeed up to IP suppliers to identify possible bugs introduced in the core during the customization process.

Conversely, when processor cores are available as soft IPs, i.e., designers have fully access to the IP source code, the process of core customization may be performed by the end user, which becomes responsible also for guaranteeing the correct operation of the IP, i.e., of the *validation* of the resulting IP, after its un-used components have been removed.

The problem of validating processor cores may be tackled either by means of formal methods or by means of simulation-based techniques. Formal methods have been successfully applied even to very complex architectures [2]-[4], but their limitations often make them suitable only for validating single components. As a result, most of the validation effort is demanded to simulation-based techniques: the processor is extensively stimulated with a wide range of workloads, whose aim is to cover all the possible corner cases, possibly enlightening design errors. For successfully achieving such a goal, the process of generation of the workload is crucial, since from its goodness it depends the possibility of effectively discovering design bugs. Several approaches have been proposed in literature to solve the complex problem of generating suitable workloads for general-purpose processors. They rely either on high-level behavioral HDL descriptions of the processors, or on more detailed register-transfer models to generate and evaluate the goodness of the computed workloads versus pre-defined metrics [5][6]. No matter which approach is adopted, the produced workloads consist of *test programs* the processor core should execute and suitable *input stimuli* for the test programs.

The workload computation problem can be greatly simplified when the processor cores to be validated are intended for being deployed in embedded applications. In this case the program a core should run is *known in advance*, while for general-purpose processors the executed program may change from time to time. The processor is indeed intended for running just one application, and therefore the test program coincides with the embedded application. Designers should thus focus their efforts in the development of the input stimuli for the test program, only.

Based on this observation, this paper proposes a novel approach to generate validation input stimuli for processor cores that have been customized for embedded applications. An automatic flow is proposed that, starting from the source-level code of the embedded application and a description of the processor core devoted to its execution, performs the following operations:

- It automatically customizes the selected processor by removing from its description all the un-used instructions, so that the resulting core embeds only those hardware components that are actually needed by the embedded application.
- It automatically generates a set of input stimuli suitable for being used during the validation of the obtained processor core.

The main novelty of this paper lies in the approach we adopted for generating the input stimuli to be used during processor validation. Conversely from the already available approach, input stimuli are generated while analyzing *only* the source-level code of the application the processor executes, while all the details about the underlying hardware, i.e., the processor, are neglected. Input stimuli generation is performed according to the approach presented in [7], where high-level metrics applied to behavioral descriptions of hardware/software systems are used to drive the generation of input stimuli, which is aimed at covering design errors and manufacturing defects.

Experimental results are reported on a soft core implementing the Intel 8051 instruction set, showing the effectiveness of the approach we propose.

The paper is organized as follows. Section 2 describes the proposed processor customization and validation flow. Section 3 details the adopted input stimuli generation process, whereas Section 4 reports some experimental results. Finally, Section 5 draws some conclusions.

2. Design flow

In this Section we describe the approach we developed for customizing a given processor core and generating suitable validation inputs.

In developing our approach, we assumed that the considered embedded application is composed of three phases:

- An *acquisition* phase, during which the data the application is intended to process are read from input devices.
- A *processing* phase, during which the acquired data are elaborate by the algorithms the embedded application implements.
- A *presentation* phase, during which the obtained results are send to output devices.

These phases can be intermingled; indeed, our approach does not mandate that one phase is completed before another one is started.

We also assumed that the targeted processor core is available as synthesizable register-transfer model. The processor source code is not encrypted and all its details are accessible to designers. Finally, we assumed that a simulation-based approach is exploited for performing the validation of the considered processor.

The flow we developed under the aforementioned assumptions is shown in Figure 1.

Three main phases compose our flow. The *Instruction Set Extraction* flow is shown in the leftmost part of Figure 1. According to it, the embedded application source code is first compiled and then linked with the needed libraries, thus obtaining the binary code the processor core should execute. Then, a binary code analyzer tool identifies the sub-set of the processor instruction set that is needed for executing the given embedded application. The result of the Instruction Set Extraction flow is thus the list of assembly instruction the processor should implement in order to correctly execute the embedded application it is devoted to.

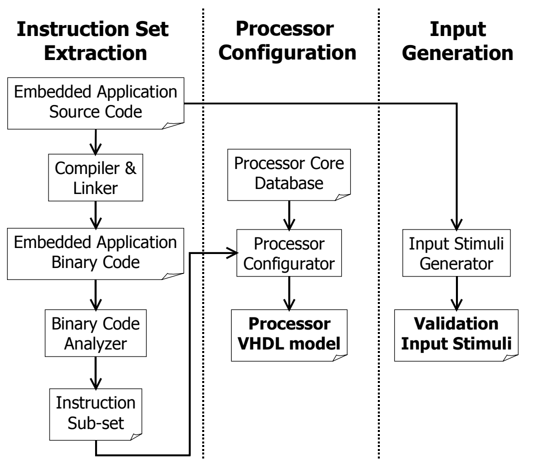

Figure 1: The proposed processor customization and validation flow

The obtained information is then forwarded to the *Processor Configuration* flow (shown in the center of Figure 1), which takes care of generating the proper processor model implementing only the required sub-set of the processor instruction set. A previously defined processor core database is exploited during this step, which contains for each instruction in the processor instruction set, the list of VHDL statements needed for its decoding, sequencing and execution. The database is exploited by the processor configurator tool for generating the VHDL source code of all the modules in the processor that are devoted to instruction management, namely: decoding unit, control unit, arithmetic/logic unit. At the end of the Processor Configuration flow an instance of the adopted processor core is available, which has been customized for executing the given embedded application.

The last phase is the *Input Generation* flow, which is depicted in the rightmost part of Figure 1. According to this flow, the embedded application source code is processed by the input stimuli generator tool, which is in charge of computing a set of input stimuli, *test vectors*, able to maximize given metrics. While performing the simulations needed for validating the processor, the generated test vectors are provided to the embedded application as inputs for its acquisition phase. Further details about the input stimuli generator are reported in Section 3. At the end of this flow, a set of validation input stimuli is available that designers may use to prove the correctness of the obtained customized processor core while running the given embedded application.

3. Test vector generation

The goal of the test vector generation process is to identify a set of stimuli that, when used as inputs for the considered application, is able to identify any error that modifies the expected application behavior. In our case, possible error locations are the application source code (due to bugs introduced during the coding process) and the processor model (due to bugs already present in the IP-core or bugs introduced during the processor customization process).

We performed the test vector generation process by addressing a purely behavioral description of the considered application, i.e., the embedded application source code of Figure 1. We indeed tackle only the application that the processor run, while all the low-level details about the processor hardware are neglected.

The test generation process is guided by high-level fault models that abstract the effects of errors in both the application and the processor model. For this purpose we considered the high-level fault models described in [8], which provide an accurate estimation of the test capabilities of input vectors while working on behavioral descriptions. The considered fault models are:

- *Bit coverage*: each bit in every variable in the application can be stuck-at zero or one. The bit coverage measures the percentage of stuck-at bits that are propagated on the application outputs by a given test sequence.
- *Condition coverage*: each condition in the application can be stuck-at true or stuck-at false. Then, the condition coverage is defined as the percentage of stuck-at conditions that is propagated to the application outputs by a given test sequence.

In order to fruitfully exploit the aforementioned high-level fault models within a test generation tool, we developed a high-level fault simulation environment, which implements the *Saboter* [9] approach through a two-step process:

1. The application source code is first instrumented by adding suitable statements that fulfill two purposes:

a. They alter the behavior of the application according to the supported fault models.

b. They allow observing the behavior of the application to gather meaningful statistics (in particular, they provide access to the contents of all the variables in the application).

During this phase, the list of faults to be considered during fault simulation is computed and stored.

2. A given set of input vectors is applied to the inputs of the application resorting to the adopted simulation environment. During the execution of the application, a preliminary run is performed without injecting faults, and the output trace of the application is recorded. Then, each fault in the previously computed fault list is injected and faulty output trace is recorded. By comparing the faulty trace with the fault-free one, we then compute the high-level coverage figure the vectors attain.

Simulation is performed by compiling the C code of the instrumented embedded application and then by running it on a workstation.

The test generation algorithm we developed is intended for refining an already existing set of test vectors, which can be either randomly generated or hand-produced by designers.

For this purpose, our high-level test generator (HLTG) implements a Random Mutation Hill Climber (RMHC) algorithm, whose pseudo-code is reported in the figure 2.

A RMHC is a Hill Climber that, given a current solution, evaluates neighbor solutions in a completely random order until an improvement is found. When an improvement is found, the process is iterated over the new solution. The process is repeated until a given termination condition is met.

In our algorithm a solution is a sequence S of test vectors; each test vector is applied over the model inputs according to the assumptions stated in section 2.

Starting from an initial solution S, a new solution S' is computed by applying a random mutation operator. This operator supports three types of mutations: it complements one randomly selected bit within a randomly selected vector of S, it increases the number of vectors in S by adding a randomly generated vector in a randomly selected position in the test sequence, or it decrease the number of vectors in S by removing a randomly selected vector in the sequence. The new solution S' is accepted if and only if it increases the goodness of the previous solution S.

In the HLTG algorithm, the goodness of a solution is defined as follows:

$$\texttt{Fitness}(S) = K_1 \cdot \texttt{Coverage}(S) + K_2 \cdot \texttt{NS}(S)$$

where:

- `Coverage(`S`)` is the average of the bit coverage and condition coverage as measured by the high-level fault simulator previously described.
- `NS(`S`)` is the number of different states the application traverses during the evaluation of a set of vectors. The state is defined as the content of every variable in the application at the end of the evaluation of one input vector. This figure is computed by exploiting the information provided by the high-level fault simulator.

These two figures are linearly combined through two constants K_1 and K_2, whose values are selected to let the first term to prevail over the second one. This assumption guarantees that a new solution is accepted only if it does not reduce the number of faults the previous solution detects.

```
HLTG(S)
{
  while( termination condition not met )
  {
      S' = apply_random_mutation(S)
      if(Fitness(S') > Fitness(S))
      {
        S = S'
        if( new faults are detected )
          save_solution(S)
      }
  }
}
```

Figure 2: HLTG algorithm

4. Experimental results

In this Section we report the results coming from several experiments we performed to analyze the capabilities of the design flow described in Section 2. Sub-section 4.1 reports results concerning our processor customization approach, which show that an instruction set may be significantly pruned when a given embedded application is concerned. Sub-section 4.2 reports results about the test vectors generation approach we developed. These results confirm the soundness of the proposed approach, and they show that the adopted high-level metrics, which are applied only to the application running on the processor and that we used to drive the test generation algorithm, are in very good agreement with lower level metrics measured on the processor hardware description.

4.1. Results of the processor customization

The processor we selected for developing some benchmark applications is a soft core that implements the Intel 8051 instruction set. The soft core is coded in about

7,200 lines of synthesizable VHDL language, and its instruction set implements 106 instructions.

As a preliminary step for the application of our design flow, we manually inspected the VHDL code of the processor, and we identified all the information needed by the binary code analyzer tool and for building the processor core database depicted in Figure 1. This step lasted for about 3 days and was performed by a skilled VHDL designer. It is worthwhile to underline that, although expensive, this step needs to be performed only once, whenever a new processor core is introduced in the design flow. The obtained information can then be re-used for any new embedded application exploiting the already analyzed core.

Table 1. The considered applications

Application Name	Lines of C code [#]	Instruction Set [#]	Validation Input Stimuli [#]	CPU time [s]
BARCODE	198	27	4,731	955
ELLIPF	113	19	130	439
LRU	107	36	6,810	1,213

After this preliminary step, we considered the three applications summarized in Table 1, taken from the high-level synthesis'92 suite. For each of them, we applied the design flow in Figure 1, thus generating a customized processor core, and the corresponding validation input stimuli. The binary code of each application was obtained through the KEIL C compiler [10].

Table 1 reports for each application the number of C lines in its source-level code. Moreover, it reports the number of instructions within the Intel 8051 instruction sets that are needed for running it and that are implemented by the obtained customized version of the processor. Finally, it reports the number of test vectors in the validation input stimuli set HLTG computed, as well as the CPU time for processor customization and HLTG execution. The figures in Table 1 shows the relevancy of the processor customization approach we adopted: for the considered applications just a relatively low number of instructions is needed among the whole Intel 8051 instruction set. By pruning the initial instruction set of the un-used instructions we can thus significantly reduce the area occupation of the synthesized processor core.

4.2. Results of the test vector generation

We adopted as measure of the goodness of a given set of validation input stimuli S the number of processor instructions that have been fully tested by S, obtaining the figure we called *instruction coverage*. An instruction is considered *tested* when simulating S all the statements and the branches belonging to the VHDL implementation of the instruction are executed.

Table 2. Coverage results for HLTG and randomly generated validation input stimuli

Application Name	Instruction Set [#]	Instructions Tested by HLTG vectors [%]	Instructions Tested by random vectors [%]
BARCODE	27	100.0	96.2
ELLIPF	19	100.0	100.0
LRU	36	100.0	97.2

To measure the instruction coverage we applied a given set of input stimuli to the application, by simulating[1] the execution of its binary code on the VHDL model of the customized processor. In order to implement the input/output communications needed by the acquisition and presentation phases described in Section 2, we resorted to the four input/output ports the Intel 8051 offers. They are 8 bit-wide bi-directional ports, which are memory mapped in the Intel 8051 addressing space. From the application developer point of view, the input/output ports correspond to C variables (`P0`, `P1`, `P2` and `P3`) that can be either read or written. Read operations on one of these variables correspond to data transfers from input devices to the processor, while write operations correspond to data transfers from the processor to output devices.

Two sets of input stimuli have been used. The first set is composed of the vectors computed by HLTG, while the second set is composed of randomly generated vectors. In both the experiments the same number of test vectors have been simulated.

From the results reported in Table 2, we can observe that HLTG vectors are able to successfully test all the instructions in the considered processor cores, while random vectors fail short in achieving such a goal for two of the three considered applications. In these cases, random vectors were not able to cover part of the VHDL statements implementing the `JNZ` instruction (for the LRU application) and `JNE` instruction (for the BARCODE application).

These results suggest the importance of cleverly selecting the test vectors to be used during simulation-based validation. Although simple, the adopted test vector generation approach is able to rapidly provide useful test vectors that overcome the limitations of purely randomly generated vectors.

To better investigate the capabilities of the high-level metrics described in Section 3 to accurately model low-level errors, we compared in Table 3 the coverage figures measured on the source-level code of the considered

[1] Simulations have been performed through the ModelSim VHDL simulator on a Sun Enterprise/250 machine running at 400 MHz and equipped with 2 Gbytes or RAM.

applications with those reported in Table 2, which have been measured by simulating the VHDL model of the processor.

Table 3. Comparing high-level with low-level metrics

Application Name	High-level Coverage of HLTG vectors [%]	High-level Coverage of random vectors [%]	Instructions Coverage of HLTG vectors [%]	Instructions Coverage of random vectors [%]
BARCODE	97.2	75.6	100.0	96.2
ELLIPF	99.9	99.9	100.0	100.0
LRU	98.8	98.2	100.0	97.2

As the reader can observe from Table 3, the high-level coverage has the same trend of the instruction coverage. For both BARCODE and LRU, HLTG vectors perform better than random ones when coverage figures are measured both at the high level (on the application source code) and at the low level (on the processor VHDL model). Similarly, random vectors and HLTG ones provide the same figures for ELLIPF when evaluated either on the application source code or on the processor model. These results indicate that accurate testability estimation can be performed starting from purely behavioral descriptions (such in the case of the source-level code of an application) while all the details of the underlying hardware (the processor devoted to executing the application) are neglected.

5. Conclusions

This paper proposed a novel approach to help designers in the simulation-based validation of application-specific processors. Suitable input stimuli are automatically generated while reasoning only on the software application the processor is intended to execute, while all the details concerning the processor hardware are neglected. Experimental results on a Intel 8051 soft core showed the effectiveness of the proposed approach, both in terms of area saving due to processor customization and validation capabilities due to clever selection of the test vectors used during simulation-based validation.

6. References

[1] OPENCORES, “WISHBONE System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores”, Revision B.3, September 2002

[2] N. A. Harman, “Verifying a Simple Pipelined Microprocessor Using Maude”, Lecture Notes in Computer Science, 2001, Vol. 2267, pp. 128-142

[3] D. Van Campenhout, T. N. Mudge, J. P. Hayes, “High-Level Test Generation for Design Verification of Pipelined Microprocessors”, Design Automation Conference, 1999, pp. 185-188

[4] M. N. Velev, R. E. Bryant, “Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exception, And Branch Prediction”, Design Automation Conference, 2000, 112-117

[5] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, “On the Test of Microprocessor IP Cores”, IEEE Design, Automation & Test in Europe, 2001, pp. 209-213

[6] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, “Fully Automatic Test Program Generation for Microprocessor Cores”, IEEE Design, Automation & Test in Europe, 2003, pp. 1006-1011

[7] O. Goloubeva, M. Sonza Reorda, M. Violante, “High-level test generation for hardware testing and software validation”, Proc. 8th IEEE International Workshop on High Level Design Validation and Test, 2003, pp. 143 - 148

[8] F. Ferrandi, F. Fummi, D. Sciuto, “Test Generation and Testability Alternatives Exploration of Critical Algorithms for Embedded Applications”, IEEE Trans. on Computers, Vol. 51, No. 2, February 2002, pp. 200-215

[9] J. Boué, P. Pétillon, Y. Crouzet, “MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance”, Proc. Int. Symp. on Fault Tolerant Computing, FTCS-28, 1998, pp. 168-173

[10] www.keil.com