Holistic Automated Code Generation: No Headache with Last-Minute Changes

Klaus Strohmayer
Dialog Semiconductor
Kärntnerstrasse 518, A-8054 Graz-Seiersberg
Email: Klaus.Strohmayer@diasemi.com

Norbert Pramstaller
Dialog Semiconductor
Kärntnerstrasse 518, A-8054 Graz-Seiersberg
Email: Norbert.Pramstaller@diasemi.com

***Abstract*—In this article, we introduce holistic automated code generation (HACG). It extends the state-of-the-art approaches for automated code generation by targeting not only the register file but also other regular structures appearing in a modern System-on-Chip (SoC). HACG is enabled by combining handwritten and automated code as well as fully automated dependency handling resulting in a self-maintained environment. This allows to quickly react on design changes throughout the entire development phase. HACG significantly improves productivity, quality, and consistency of the entire design process of a mixed-signal SoC.**

I. INTRODUCTION

Mixed-signal System-on-Chip (SoC) designs are getting more and more demanding with respect to system complexity, first time right, and time to market. State-of-the-art design techniques and verification methodologies help dealing with complex mixed-signal designs. Additionally, automated code generation is applied in order to ease the maintenance of regular structures such as the register file, analog/digital interfaces, level shifter instances, etc. (see [1], [2], [3], [4], [6]).

Functional changes are part of a designers daily business. At any phase of the development flow, changes are either requested directly by the customer or are resulting out of the design and verification progress. Examples of such changes are new or altered features, the need for additional trimming bits or trimming range extensions, the adaption of analog configuration parameters, or the extension of test modes and scan isolation values. Default register values and register map re-orderings are also common changes.

In an SoC, a small change (e.g. redefinition of a single register or an additional required test mode) in general requires modifications on multiple steps in the design flow. This results in significant effort for design, verification, and implementation. Even worse, these changes and in particular last-minute changes are very error-prone and can impact already passing regression runs, synthesis and place & route possibly delaying the project schedule. Existing automated code generation approaches mainly target the handling of register files. These approaches help dealing with changes in this area and are definitely a valuable first step. Nevertheless, on top of the register file, several modifications still need to be done manually. More precisely, in addition to regenerating the register file the related connections through the hierarchies, pins, etc. need to be adapted manually to get the environment up and running again. In a mixed-signal SoC, regular structures appear throughout the entire design flow: specification, source code generation at register transfer level (RTL), verification planning, verification, physical implementation (synthesis, and place & route), lab evaluation, production testing, and application-software development. Extending automated code generation to covering also these steps in the design flow will allow an even faster reaction on (last-minute) changes and hence reduce the risk for injecting errors. We will refer to this extension to the entire design flow as *holistic automated code generation (HACG)*. Compared to existing solutions, HACG mainly differentiates by the fact that it supports combing handwritten and automated generated code. Further, it is able to automatically derive dependencies for the generation process.

The remainder of this article is structured as follows. In Section II, we present an overview of a general mixed-signal SoC. We focus on a common case of a configuration register located in the digital domain controlling an analog component. Furthermore, we list several language constructs that appear in our design flow. Based on this, we derive the requirements for HACG in Section III. We show in detail how HACG works and give an example for the application of the presented tool. Finally, we summarize in Section IV.

II. THE JOURNEY OF A SINGLE BIT IN AN SOC

In this section, we present an overview of a typical SoC design. We particularly focus on an example of a configuration register used to control an analog component such as an LDO. After this, we list several language constructs found in our digital design flow. This serves to demonstrate in which areas we can apply the principle of HACG.

Figure 1 shows the general structure of a typical mixed-signal SoC. The main components are the central control unit (either a dedicated control state machine or a microcontroller including memory), interface units (e.g. SPI, I2C), and the digital control to analog components (e.g. ADC, LDO, switching converter, oscillator, etc). All these components communicate via a centralized bus system.

A. Detailed description of a configuration register

At the bottom of Figure 1, a configuration register controlling an analog component is shown. This configuration register (Figure 1.A) is controlled via the centralized bus. The

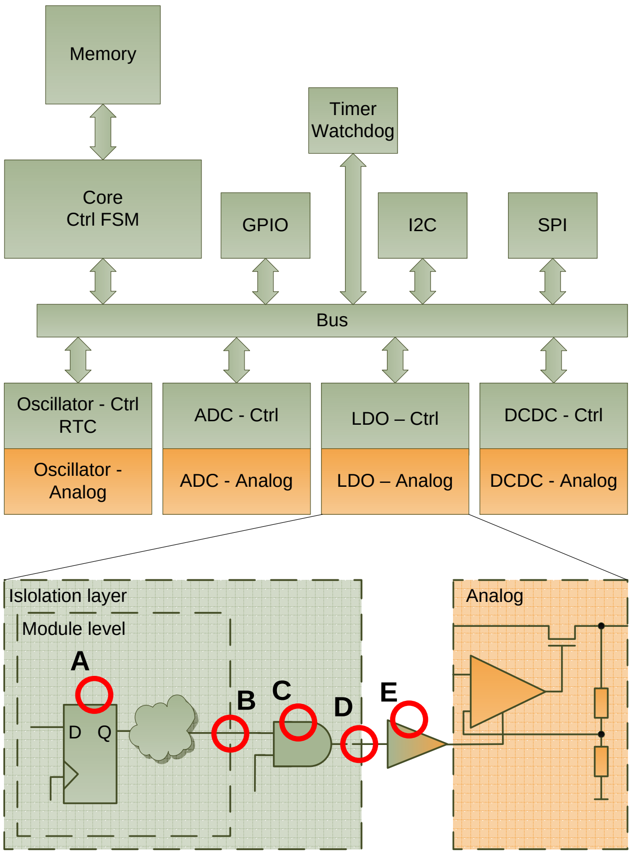
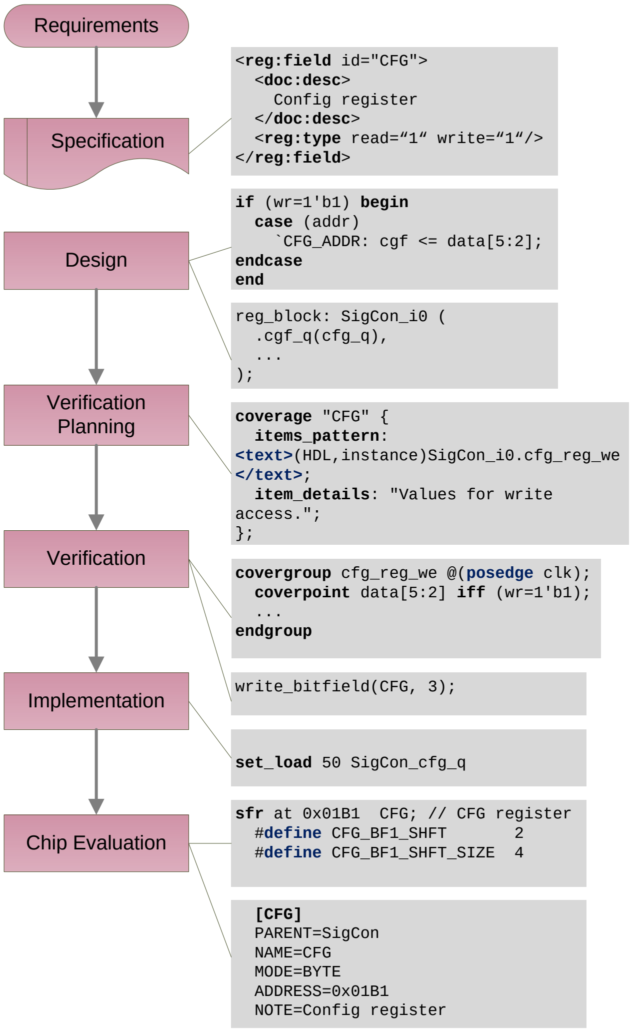


Fig. 1. General SoC Structure.

output of this register is then (optionally) processed by the combinational logic according to the specified functionality of the analog component. After this the signal crosses the module hierarchy (Figure 1.B). It is a common design approach to include dedicated logic at the digital top level for test and scan isolation (Figure 1.C). After the insertion of the test isolation logic the signal crosses the digital/analog boundary (Figure 1.D). Modern SoCs are usually composed of several voltage domains. Therefore, it is necessary to provide level shifters between the different voltage domains (Figure 1.E).

The description represents a typical case for a configuration register located in the digital domain controlling an analog component. It is important to note that in a modern SoC a significant amount of module hierarchies, thousands of configuration registers, and several hundreds of level shifters are not an exception. Even more importantly, the definition of analog trim and configuration registers often go through multiple iterations throughout the design process. Such changes require manual interaction in various steps of the design flow.

This simple description already makes clear that an efficient method for handling the generation of these registers and the connection through different hierarchies significantly reduces the overall design effort. Additionally, the risk of injecting errors can be minimized.



```
<reg:field id="CFG">
  <doc:desc>
    Config register
  </doc:desc>
  <reg:type read=“1“ write=“1“/>
</reg:field>
```

```
if (wr=1'b1) begin
  case (addr)
    `CFG_ADDR: cgf <= data[5:2];
endcase
end
```

```
reg_block: SigCon_i0 (
  .cgf_q(cfg_q),
  ...
);
```

```
coverage "CFG" {
  items_pattern:
<text>(HDL,instance)SigCon_i0.cfg_reg_we
</text>;
  item_details: "Values for write
access.";
};
```

```
covergroup cfg_reg_we @(posedge clk);
  coverpoint data[5:2] iff (wr=1'b1);
  ...
endgroup
```

```
write_bitfield(CFG, 3);
```

```
set_load 50 SigCon_cfg_q
```

```
sfr at 0x01B1  CFG; // CFG register
  #define CFG_BF1_SHFT       2
  #define CFG_BF1_SHFT_SIZE  4
```

```
  [CFG]
  PARENT=SigCon
  NAME=CFG
  MODE=BYTE
  ADDRESS=0x01B1
  NOTE=Config register
```

Fig. 2. Language Constructs in the Design Flow.

B. Language constructs in an SoC

In Section II-A, we have shown the importance of an efficient method for handling configuration registers. Looking at the design flow, we see that the information on these registers appears in several steps. The same information is described by different language constructs. In the following, we will describe these language constructs appearing in the design flow. The flow and the language constructs are shown in Figure 2. Note that for the sake of visibility we have not drawn the iterative behavior of the design flow.

The specification of an SoC includes register descriptions, memory maps, and for instance pin lists. These descriptions can be automatically derived from a database by using XML, Visual Basic macros, or other languages.

The implementation of the digital design functionality at RTL is done by using hardware description languages (HDLs) such as SystemVerilog, Verilog, or VHDL. Amongst others, it includes language constructs for register definitions, register implementation, connectivity through multiple hierarchies, test mode isolation (e.g. scan isolation), and level shifters. An example is given in Section II-A.

For verification planning it is important to define verification metrics of the registers in the verification plan. Furthermore, the according implementation as part of the source code describing the hardware functionality has to be defined. The metrics include coverage information and test cases for verifying the register accesses. Also reset values or test isolation values can be part of the verification plan. The verification plan including the metrics and the mapping to the source code of the design is stored in a tool dependent format. For instance the format used by Incisive Enterprise Manager from Cadence [7].

For verification it is important to increase the level of abstraction. This requires high-level functions to access and verify the registers. These functions are written in the applied verification language (e.g. SystemVerilog or e). Additionally, the implementation of coverage and assertion constructs to support metric driven verification is also a part of the verification process. This is done by using SystemVerilog Assertions (SVA) or by the Property Specification Language (PSL).

For the (physical) implementation of the digital design it is required to define design constraints. Such constraints are for instance applied to the interfaces between the analog and digital domain or to the connection to I/Os. Defining constraints can for example be done with *tcl*.

The last step in our design flow is the chip evaluation. For chip evaluation high-level functions are often written in C or LabView. Many of these functions are based on definitions placed in C-header files. For the debugging of the controller software the register information (configuration) needs to be provided to the debugger (e.g. Keil μVision [8]). The debugger derives this information from a configuration file.

All the language constructs listed in this section are ideal candidates for automated code generation. In combination with the register handling this leads us to the principle of HACG described in the next section.

III. HOLISTIC AUTOMATED CODE GENERATION (HACG)

Holistic automated code generation is based on the following four prerequisites.

1) Usage of a unique database (XML, Excel, etc.) for each regular structure
2) Automated generation of regular structures derived from the unique database (databases)
3) Combination of handwritten and automated generated code within source files throughout all steps of the design flow
4) Automated handling of dependencies between unique database and automated code generation.

Approaches for dealing with 1. and 2. are available on the market (see [1], [2], [3], [4]), or do exist as proprietary in-house solution [6]. These solutions for automated code generation mainly target the register generation and are based on a unique database approach. Even if documentation, implementation, and verification of the registers is employed by these tools the following features are missing: 3. the combination of handwritten and generated code and 4. the automated handling of dependencies of the generated data from the unique data source. Items 3. and 4. add the term "holistic" to automated code generation as they enable dealing with regular structures in the entire design flow. The idea for combining handwritten and automated generated code follows the basic principles of template based design. This combination supports a very deep integration in the design process. Furthermore, all native files are being parsed in order to determine the dependencies from the corresponding unique database (or databases). As a result it is possible to setup a self-maintained environment, i.e. a change in the database is fully covered by HACG consistently updating the generated data in the according source files. The approach is fully language independent and can easily be introduced step by step to already existing designs.

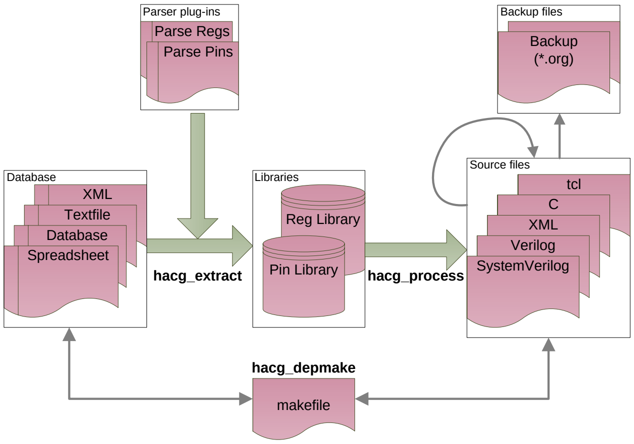


Fig. 3. HACG flow.

A. HACG Flow

In this section, we describe the basic flow of HACG. The flow is graphically depicted in Figure 3. It consists of two main steps, namely the data base extraction and the processing of the database. Additionally, in order to maintain the dependencies between source files and the libraries, an automatically derived makefile is provided. In the following the according functions of HACG are described in detail.

- ***hacg_extract*****:** During the first step of HACG the unique databases are being parsed by applying dedicated plug-ins. These can be plug-ins for parsing Excel spreadsheets, for parsing XML files, etc. The output of hacg_extract is a proprietary binary library containing the data of the parsed database. For instance extracting the data from a register-list spreadsheet results in a register library and a pin database results in a pin library.
- ***hacg_process*****:** In the second step of HACG the source files are being processed, where the parsing tool is looking for templates (pragmas) defined in HACG. The templates are based on the template toolkit TT2 [5]. Each of these templates includes data fields that correspond to information that is stored in the libraries generated by applying hacg_extract. The templates in combination

with the data stored in the proprietary libraries result then in the final source code (see also Section III-B). An additional feature of hacg_process is that a backup copy of each processed source file is created. This allows to undo changes in the case of an erroneous template which is of particular help in the initial phase of template definition.

- ***hacg_depmake*:** This process checks the dependency of the source files in relation to the data in the library and generates a make file out of it.

If there is a change in the database (e.g. pin list or register list) hacg_extract and hacg_process need to be repeated. If there is a change in the source file (e.g. adapting the template or adding new data fields to the template) only hacg_process needs to be redone. This step will be discussed in more detail in Section III-B.

B. HACG Applied

In this section, we describe the processing of source files via hacg_process. Figure 4 depicts the flow diagram for processing a single source file and Figure 5 shows a source-code snippet with an embedded code generation template. This snippet combines hand written and automatically generate code.

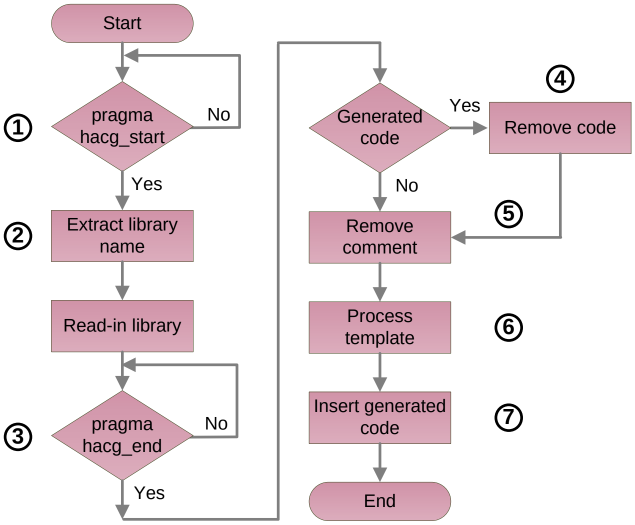


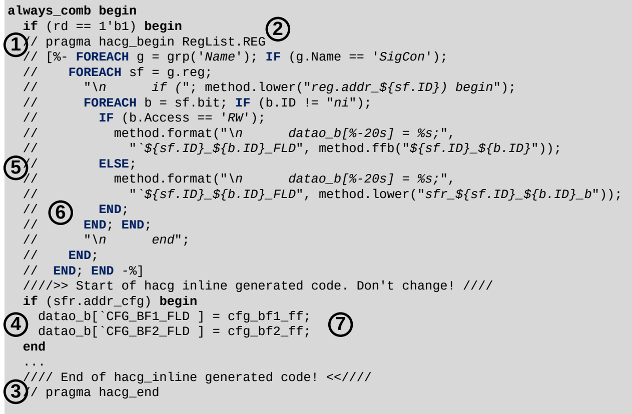
Fig. 4. HACG processing of source files

Step 1 starts with the search for the keywords “pragma hacg_begin” in all lines beginning with a comment delimiter. This determines the start of a section for HACG. Note that all lines of the template need to start with the comment delimiter of the used language.

In step 2, the library name is extracted. The name of the library defines the relationship between the unique database and the data required for processing the template.

Once the library data is read in, the keyword “pragma hacg_end” is searched within the source file (step 3). This is done in order to determine the end of the HACG template.

In step 4 it is verified if a generated source code already exists. Any generated source code is placed between “////>> Start of hacg inline generated code. Don’t change! ////” and the unique termination “//// End of hacg inline generated code! <<////”. If auto-generated code already exists it is simply removed.



```
always_comb begin
  if (rd == 1'b1) begin
  // pragma hacg_begin RegList.REG
  // [%- FOREACH g = grp('Name'); IF (g.Name == 'SigCon');
  //    FOREACH sf = g.reg;
  //      "\n      if ("; method.lower("reg.addr_${sf.ID}) begin");
  //      FOREACH b = sf.bit; IF (b.ID != "ni");
  //        IF (b.Access == 'RW');
  //          method.format("\n      datao_b[%-20s] = %s;",
  //            "`${sf.ID}_${b.ID}_FLD", method.ffb("${sf.ID}_${b.ID}"));
  //        ELSE;
  //          method.format("\n      datao_b[%-20s] = %s;",
  //            "`${sf.ID}_${b.ID}_FLD", method.lower("sfr_${sf.ID}_${b.ID}_b"));
  //        END;
  //      END; END;
  //      "\n      end";
  //    END;
  //  END; END -%]
  ////>> Start of hacg inline generated code. Don't change! ////
  if (sfr.addr_cfg) begin
    datao_b[`CFG_BF1_FLD ] = cfg_bf1_ff;
    datao_b[`CFG_BF2_FLD ] = cfg_bf2_ff;
  end
  ...
  //// End of hacg_inline generated code! <<////
  // pragma hacg_end
```

Fig. 5. Example template (pragma) of HACG.

Afterwards, the comment delimiter in front of the template code is removed (step 5). This allows to execute the template toolkit. Note that the source code file is not directly affected; the template is directly processed in memory.

In step 6, new source code is generated by processing the template. The data used for processing the template is begin derived from the library determined in step 2.

Finally, in step 7 the newly generated source code is inserted at the location of previously removed code. The template is now fully processed.

Steps 1 to 7 are repeated as often as necessary within one source file. It is possible to have several templates using the data from different libraries within one file.

IV. SUMMARY

In this paper, we have emphasized that modern SoCs include several regular structures that can be exploited for automated code generation. We have identified four prerequisites that allow to exploit all regular structures in the design flow. Our approach, referred to as holistic automated code generation (HACG) is enabled by combining handwritten and automated code as well as fully automated dependency handling resulting in a self-maintained environment. This approach significantly improves productivity, quality and consistency of the entire design process of a mixed-signal SoC. It enables designers to react on unavoidable (last-minute) design changes with small effort and low risk: it helps reducing designer’s headache.

REFERENCES

[1] PDTi^{TM} *SpectaReg* http://www.productive-eda.com
[2] Denali Blueprint https://www.denali.com/en/products/blueprint.jsp
[3] Agnisys *IDesignSpec*^{TM} http://www.agnisys.com
[4] Semifore *CSRCompiler*^{TM} http://www.semifore.com
[5] The Template Toolkit http://www.tt2.org
[6] B. Banerjee and S. Rajan and S. Naidu, *Automated approach to Register Design and Verification of complex SOC.* Presented at Design and Verification Conference & Exhibition (DevCon) 2011, San Jose, CA http://events.dvcon.org/2011/proceedings/papers/11_2.pdf

[7] Cadence *Incisive Enterprise Manager* http://www.cadence.com
[8] *KEIL*^{TM} μVision http://www.keil.com/uvision/uv4.asp