System-On-Chip Data Processing and Data Handling Spaceflight Electronics

I. Kleyner[1], R. Katz[2] and H. Tiggeler[3]
[1]Orbital Sciences Corp.
[2]NASA/Goddard Space Flight Center
[3]University of Surrey

Abstract

This paper presents a methodology and a tool set which implements automated generation of moderate-size blocks of customized intellectual property (IP), thus effectively reusing prior work and minimizing the labor intensive, error-prone parts of the design process. Customization of components allows for optimization for smaller area and lower power consumption, which is an important factor given the limitations of resources available in radiation-hardened devices. The effects of variations in HDL coding style on the efficiency of synthesized code for various commercial synthesis tools are also discussed.

I. INTRODUCTION

Performance requirements for spaceflight electronics have been increasing as detectors produce greater amounts of data at higher resolutions. Concurrently, there is an increasing need to produce spacecraft electronics in shorter periods of time using less spacecraft resources.

It is a common occurrence that similar or identical spaceflight data handling and processing hardware functions tend to be redesigned repeatedly by independent teams of engineers or even within the same team. This obviously is inefficient and costly in both development and verification time. Alternatively, commercial IP cores are available for many functions required for spaceflight hardware design; however, these cores tend to be ill suited for the specifics of flight hardware design due to the limitation of resources available in radiation-hardened devices and reliability issues, particularly for the radiation environment.

The objective of our research is to develop, demonstrate, and refine architectural techniques and tools for FPGA designers to permit rapid, reliable development of high-speed processing and data handling functions, integrated onto a single chip.

II. IP GENERATION ENVIRONMENT

Development of an integrated IP generation environment was the first phase of the project. As shown in Figure 1, Kompiler combines customized VHDL code-writing capabilities with built-in test vector generation, synthesis execution, and simulation for verification.

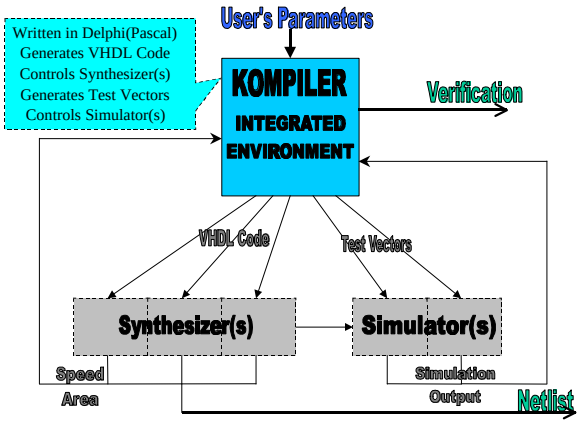

Figure 1. Kompiler IP Generation Flow.

Generation of VHDL code for a component is performed by utilizing a

previously coded generic “template," the contents of which are embedded in the source code of Kompiler. The front end user interface provides selection of various options to satisfy the requirements of the particular application for which the component is intended to be used, while allowing a trade-off between resources and features. For a data-heavy component (such as a ROM) input data is provided as a specially formatted ASCII text file. Additionally, a synthesis tool selection option is provided to optimize the created code for known specifics of a particular tool to be used for synthesizing the component. Furthermore, more than one coding style for a block may be available. As a result, a customized version of a generic component’s VHDL code is produced.

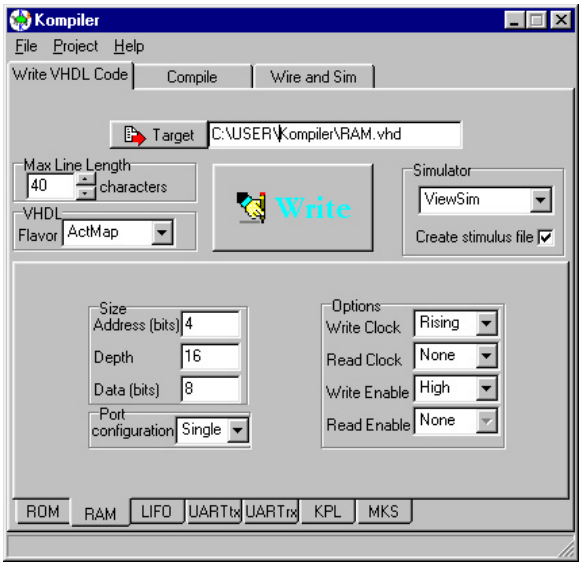

Figure 2. Kompiler User Interface.

Creation of customized VHDL code for a component is a useful feature by itself, but it can be even more powerful if used in conjunction with other Kompiler features. During the next step, the VHDL code generated by Kompiler is synthesized from within the Kompiler environment, using the command line interface of the synthesis tool. If more than one coding style for the component can be utilized or different synthesis tools are available, the code creation/synthesis loop can be repeated for all available combinations. In this fashion, the optimal solution may be obtained with minimal effort from the design engineer. In case of FPGA space applications, this is often the one consuming the least amount of resources and power.

After completing the code creation/synthesis phase, the component’s implementation is automatically verified. The Kompiler calls the gate-level simulator via its command line interface, with the previously generated stimulus/checking file as a parameter. Design verification is a labor-intensive, error-prone phase of the design process, often being larger than the detailed circuit design. Utilizing test vectors algorithmically generated by the Kompiler saves considerable time and effort, as does Kompiler’s automated processing of the simulation results.

III. COMBINATIONAL IP BLOCKS AND CODING STYLES

As was mentioned above, System-on-Chip (SoC) building blocks to be implemented in the Kompiler environment include a variety of universal and commonly used components such as ROM, RAM, FIFO, LIFO, and Register File. More complex and specific structures such as UART Receiver, UART Transmitter, Timer, and Data Encoder/Decoder can also be developed into Kompiler blocks. In addition to these data handling functions, arithmetic cores such as an array multiplier, CORDIC functions, and DSP cores (FIR filter, IIR filter, Correlator) are prime candidates.

The generation of combinational logic blocks (ROM component) was chosen as the first Kompiler application for a variety of reasons. Such a component is described

functionally by defining the value of output for every combination of inputs. This is rather laborious and error-prone to code manually, but the task can easily be automated. In addition, the ROM component represents a convenient opportunity to study the advantages and disadvantages of different coding styles, since automated code creation allows for fast and efficient implementation of different coding techniques.

The most straightforward style for the ROM component, named “Word-Case" is the most natural for “human” coding. One long VHDL Case statement is used to implement the structure with the address used as the evaluated expression. The list of choices represents all possible combinations of inputs and corresponding values for outputs based on the contents of the input data file, which contains the functional description of the block.

```
When "000001010" => ZData <= "100101";
When "000001011" => ZData <= "111100";
When "000001100" => ZData <= "101101";

        ...
```

Alternatively, each bit of the output can be coded using separate *Case* statements (“Bit-Case” style)

```
When "0000000" |
     "0000100" |
     "1111110"     => ZData(0) <= '0';

          ...
```

or input data can be represented by an array of constants (“Hans-Array” style)

```
constant ROM : rom_array := (
                   "0000000000000001",
                   "1110001001100001",
                   "1101000100110001",
                   "1110001101001001",
                        ...
```

Additionally, we have developed two more styles of coding for combinational logic blocks. By viewing the ROM as a set of functions, the minterms can be selected and written as logic equations in a sum-of-product format. Either the "ones" may be grouped and the minterms written directly or the "zeros" may be written and the resulting function complemented.

When selecting "ones" we have generated code segments that look like this:

```
ZData(2) <= '0' or ( ZA(0)
             and not ZA(1)
             and not ZA(2)
             and not ZA(3)
             and not ZA(4)
             and not ZA(5)
             and not ZA(6) )
  or ( ZA(0) and     ZA(1)
             and not ZA(2)
             and not ZA(3)
             and not ZA(4)
             and not ZA(5)
             and not ZA(6) )
  or ...
```

When selecting "zeros" the generated code is structured in this manner:

```
ZData(2) <= not ( '0' or ( not ZA(0)
                       and not ZA(1)
                       and not ZA(2)
                       and not ZA(3)
                       and not ZA(4)
                       and not ZA(5)
                       and not ZA(6) )
  or                 (     not ZA(0)
                       and ZA(1)
                       and not ZA(2)
                       and not ZA(3)
                       and not ZA(4)
                       and not ZA(5)
                       and not ZA(6)
  or ...
```

Obviously, writing the “logic equations” code manually is not a task that can be accomplished reasonably for a logic block of any significant size. However, the algorithm for automating such task was rather straightforward to code into the Kompiler. The resulting VHDL code is often somewhat

bulky and quite unreadable for a human eye, but, as was discovered later, sometimes preferred by certain synthesis tools.

Besides the usefulness of the ROM component for the Kompiler as a useful SoC building block, it enabled us to study the efficiency of different synthesis tools. This included the effects of processing VHDL descriptions of blocks of logic written using different coding styles. Additionally, for a fixed description, the targeting of the generated netlist into different technologies was studied. Among the synthesis tools available for us during this study are Actmap from Actel, Synplify from Synplicity, Design Compiler from Synopsys and Leonardo from Exemplar. A few real-life input data sets were randomly selected and all five aforementioned coding styles were utilized to generate VHDL blocks, which were then synthesized for various Actel FPGA devices. Some of the results are illustrated in Figure 3.

The first test case (Figure 3a) represents results (in terms of module count) of implementing a 128x16 bit sine wave LUT targeting Actel Act 3 technology. As can easily be seen from the chart, the size of the synthesized block for the same data content varies significantly depending on the coding style as well as the synthesis tool used. For example, Actmap synthesized “Word-Case” and “Hans-Array” style code most efficiently, producing a module count of 108 and 103 modules respectively. However, the Synplify synthesizer produced its best results of 199 and 201 modules for code written in “Logic 0” and “Logic 1” styles, while doing somewhat worse with case/array styled code. Other interesting trends may be observed. For example, the fact that the “Bit-Case” style produced the most efficient resource consumption in conjunction with Synopsys Design Compiler, but was the least efficient with the Exemplar and Synplify tools.

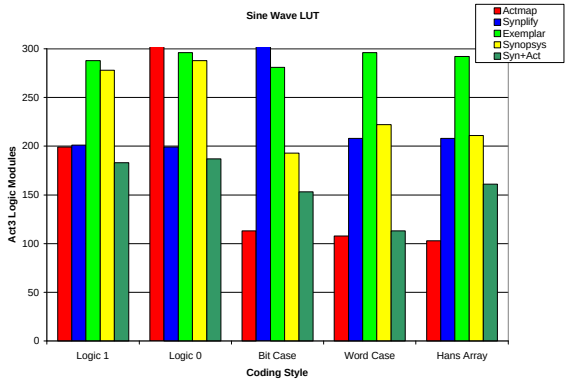

Figure 3a. Sine Wave Synthesis Results for Actel Act 3 Target Technology

The next selected test case was identical to the one above with the only exception of a different target technology; Act 3 was replaced with Actel SX. In some aspects the results (represented by the chart in Figure 3b) were similar to the ones in the previous example. For instance, the Synplify tool was most efficient for synthesizing “Logic1/Logic0”-style code, Synopsys “preferred” code written in “Bit-Case” style, and Actmap was most compatible with “Word-Case” and “Hans-Array” styles. Some results, however, were quite surprising. For example, Actmap synthesis for SX technology required a factor of two or more increase in the number of modules (from 108 to 216 for “Word-Case," from 103 to 270 for “Hans-Array”) as compared to Act 3 technology. It is noted that the SX C-Cell is a superset of the Act 3 C-Module and should have been more efficient [1]. In fact, out of all synthesis tools utilized in the study, only Synplify was able to take advantage of the improvement in the SX logic module relative to the Act 3 module, resulting in lowering of the total module count. The relative lack of performance in Actmap for SX in comparison to Act 3 technology can partially be explained by the fact that the version of the tool available at the time of study did not have all of the optimization algorithms implemented. The next revision of the Actmap software did

show improvement by bringing the module count for SX ("Word-Case" style) down to 136. However, the module count for Act 3 technology in the newer revision also was 136, up from 108 for the previous release!

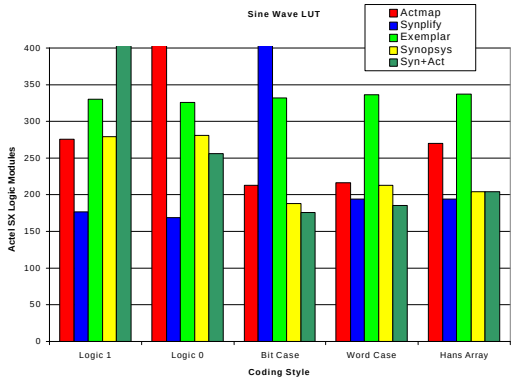

Figure 3b. Sine Wave Synthesis Results for Actel SX Target Technology

The chart in Figure 3c also represents synthesis results targeted for SX technology, but for a very different data pattern or ROM content – a look-up table for a correlator. This particular data set contains a certain number of "don't care" values in the address field. This can be efficiently utilized in "Logic1/0" style code which selects minterms to write logic equations but not in the "Case" style, since VHDL syntax does not allow "don't care" values in the choice field of *Case* statement. In this test case both Synplify and Actmap were most effective with the "Logic 1" coding style, apparently being able to take advantage of "don't care" values in the address field. This data set had a prevalence of "0"s for the output values with no more than 2 out of 8 output bits containing "1"s for any address value. This was not the case with the Synopsys Design Compiler tool, as the "best" results (lowest module count) were achieved synthesizing with the "Word-Case"-style code.

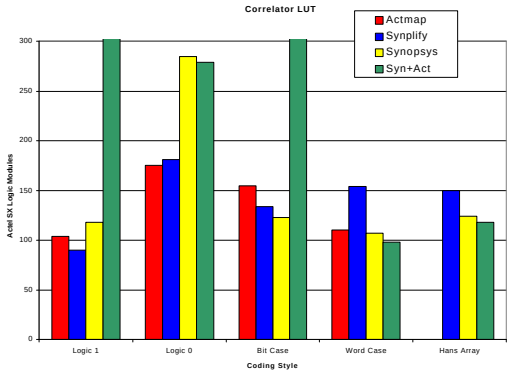

Figure 3c. Correlator LUT Synthesis Results for Actel SX Target Technology

A few more input data sets of various contents were tested in a similar fashion. Some quite consistent trends were detected; i.e. certain synthesis tools were more efficient with certain coding styles. The most obvious result of the study was the fact that for any synthesis tool used the resource-effectiveness of implementing a combinational block was highly dependant on the content of data, coding style, target technology and sometimes even the tool software revision level.

IV. SEQUENTIAL IP BLOCKS

Given the relative successfulness of the experiment with the ROM component for the Kompiler, additional relatively simple blocks, which include sequential elements, were developed into Kompiler features. These included RAM, LIFO, and UART receivers and transmitters. The development effort for these blocks concentrated on making them highly customizable. This allows a user, for example, to select the active level of logic, synchronous/asynchronous, single/dual port configuration, active edge for clock, and exact depth and width of memory. UART IP cores are readily available on the commercial market, but usually they are very comprehensive and carry many features that may not be needed for a particular

application. For example, the simplest version of the UART IP core available from Actel requires 180 Actel SX modules. On the other hand, the Kompiler offers separate receiver and transmitter blocks that can be stripped-down to only needed level of functionality. This provides the benefit of significant resource savings and frequently lower power dissipation, a goal of most space-borne designs. Additionally, some sub-blocks of both the receiver and transmitter can be optionally utilized by other components of a SoC design or can be shared between the receiver and transmitter modules. As a result, simple but fully functional double-buffered versions of both transmitter and receiver were synthesized utilizing less than 40 modules for each block, which represents significant resource savings. The Kompiler also offers single buffer version for minimal resource utilization.

Kompiler generated blocks like ROM, RAM, and LIFO, even though very often useful by themselves and able to save a designer a significant amount of time and/or resources are relatively primitive in their nature. Building "manually" a whole system-on-chip with any meaningful functionality out of such basic components can be a very labor-intensive and error-prone task. For the next phase of the Kompiler development, the emphasis was on creating a more complex component with functionality sufficient to become a foundation for a potential complete system capable of performing a useful task. Additionally, we set the goal of complete automation of this task.

As a prototype for such a component, the AM29CPL154, an obsolete single-chip Field Programmable Controller by AMD, was selected. In the past this part was used extensively by many engineers for implementing complex state machines and controllers (i.e. focal plane array controllers). For space flight applications, the commercial part was not acceptable because of radiation and power consumption concerns. For example, testing by JPL showed that the device was susceptible to Single Event Latchup (SEL) [2]. The Kompiler-integrated version of the part, called the 29KPL154, inherited all features of the original controller including compatibility with the native assembler and simulator. Following our development philosophy, we required that it be fully customizable to fit *exactly* the requirements of a particular application and be fully integrated into Kompiler environment. An additional goal was to analyze the processor and its instruction set architecture to explore possible optimizations for implementing small processors.

AM29CPL154 original features (all preserved in 29KPL154 component) include a 512x36-bit external program ROM, 8 test inputs, 16 user outputs, 28 instructions, and a 17x9-bit stack [3]. Developing the 29KPL154 as a Kompiler block required not just copying and implementing the architecture of the original device, but allowing for customization and optimization of the component as well. Among the user-configurable options for the 29KPL154 are program memory internal, external or a combination, custom sizing of stack depth (0 to 17) and additional test inputs (up to 8) and user outputs (up to 16). If internal program memory is desired, the contents of microprogram are synthesized as a ROM block with the implementation utilizing the optimal style of coding. The user program is analyzed by the Kompiler and only utilized instructions and test conditions are implemented in hardware; the decoder, execution hardware, registers, etc. for instructions not used for an application are not included in the final synthesized design. Additionally, instruction encoding may be optionally optimized by inserting "don't

care's into unused fields of certain instruction. An assembler typically assigns either a '1' or a '0' to an unused field in the instruction. The Kompiler, when post-processing the assembler's output, replaces the assigned values with a "don't care," giving the optimizer more freedom to produce a more compact design.

The AM29CPL154 assembler is fully integrated into the Kompiler platform allowing for "one-click" generation process. The user only needs to select desired options and configurations and supply a file with assembly code (Figure 4). Two of the building blocks for 29KPL154 were implemented using previously developed components - program memory is based on the ROM component and the stack utilizes the LIFO component.

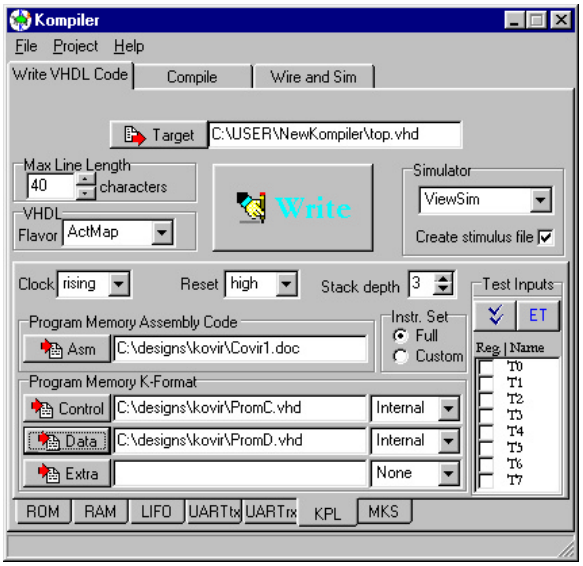

Figure 4. 29KPL154 Generation.

To evaluate the potential benefits of customization and optimization for the 29KPL154 a program used in a previous 29CPL154 application was utilized to generate a customized processor. The sample program occupies approximately one quarter of the maximum allowable 512 word program memory and was originally used for prototyping a Focal Plane Array controller for a spaceflight mission. The 29KPL154 component was generated for three different levels of customization - unoptimized 29CPL154-like, optimized program ROM only, and a fully customized version with custom-tailored stack, instruction decoder, and optimized instruction encoding. The results of synthesis using Actmap and Synplify tools for Act 3 as a target technology are shown on the chart (Figure 5).

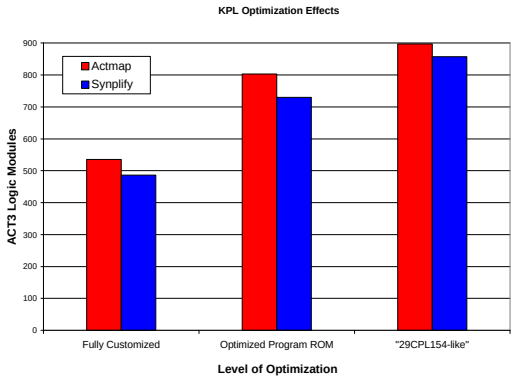

Figure 5. Controller Implementation with 29KPL154 utilizing various levels of customization.

The chart clearly shows that the implementation of a customized processor has significant gain for the user. In this example, we see that the optimized 29KPL154 brings about approximately 40% savings in resource consumption for this particular design, while still maintaining full compatibility with the original device.

V. CONCLUSIONS

The Kompiler development is still very much a work in progress, as the verification effort for 29KPL154 is under way and additional components are incorporated into Kompiler environment. Indeed, the extensible design of the Kompiler is geared towards continuously adding new components and features, while presenting the user with a consistent, easy-to-use interface.

Nevertheless, the concept of generating small and moderately sized custom-configured blocks of IP using an integrated environment such as Kompiler seems to be proven as a realistic task. It is also quite obvious that at the very least using automated generation of IP blocks shortens the development time and helps avoid simple coding errors. For certain tasks, like generating a large block of combinational logic the manual coding approach is extremely labor-intensive and error-prone, and utilizing Kompiler code-generating capabilities allows for tremendous savings in engineering time and optimization sometimes unattainable with traditional methods. The rapid generation of HDL using different coding styles makes optimization of large logic blocks feasible, practical, and effective.

It was also shown that along with small and simple blocks, more complex and comprehensive cores capable of substantial functionality can be integrated into fully automated IP-generating environment such as Kompiler. The 29KPL154 has the capability to be a cornerstone of SoC implementation using a small processor for a spaceflight application, yet it can be customized to fit in a small, radiation-hardened FPGA.

We have also determined that combining HLL-based software equipped with a simple user interface with VHDL permits a convenient and flexible approach to hardware design. The resulting environment is suitable for implementing components ranging from very simple and universal to more complex and specialized; however, the task of embedding and customizing a component as well as designing a suitable user interface escalates greatly with block's increasing complexity.

Additionally, it was shown conclusively that various synthesis tools perform identical tasks with various degree of efficiency depending on the style of coding, the specific data content, and the software revision level of the tool.

REFERENCES

[1] Actel Corporation. Actel 54SX Family FPGA, 1999.

[2] JPL/NASA Radiation Effects Database. http://radnet.jpl.nasa.gov/SEE/1188.txt

[3] Advanced Micro Devices. Am29CPL154 Field-Programmable Controller, 1990.