Comparative study of Model-based hardware design tools

Van Beeck, Kristof
Heylen, Filip
Meel, Jan
Goedemé, Toon

February 5, 2010

Lessius | Campus De Nayer,
Association K. U. Leuven,
Jan De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium
{kristof.vanbeeck, filip.heylen, jan.meel, toon.goedeme}@denayer.wenk.be

Abstract

Nowadays hardware design has become a very complex task due to the fact that programmable hardware, such as FPGA's, becomes increasingly complex with regard to the number of transistors. In this paper, we will give an overview of some important tools for model-based hardware design. Instead of starting from a complex, low-level VHDL or Verilog description, these tools enable to describe an algorithm in a model-based description. They even promise a totally automatic translation of this model-based description to HDL. These high level tools bring hardware design (based on FPGA's) to a wider audience, while increasing designer productivity. We tested a selection of these model-based design tools on an image processing example, and examined their ease-of-use and effectiveness.
Keywords: *model-based design, vhdl, verilog, automatic code generation, hardware abstraction, image processing*

1 Introduction

The complexity of integrated circuits increases every year. As stated by Moore's law, the number of transistors on integrated circuits doubles every two year, as shown in figure 1. This offers the designers the possibility to create faster, larger and more complex designs using less area. Due to the fact that the productivity of design tools increases with a lower rate, a

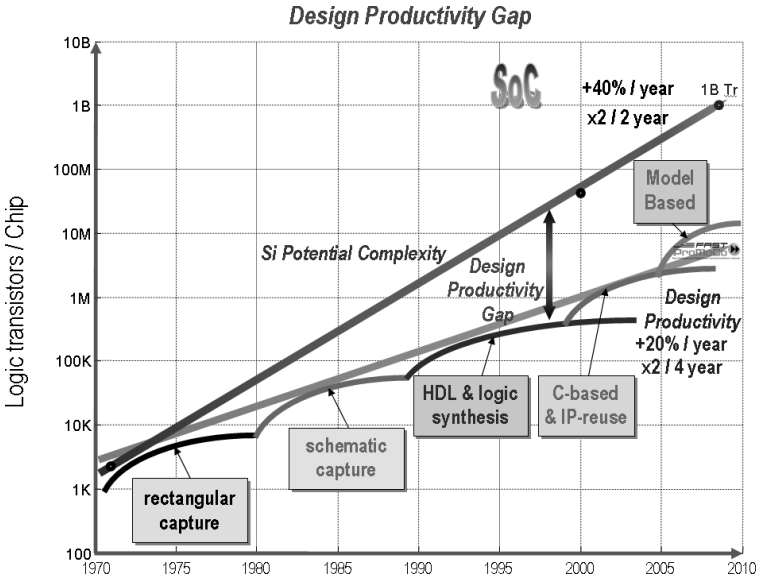

Figure 1: Moore's law and resulting Design Productivity Gap

design productivity gap arises. Implementing complex algorithms using low level hardware languages like VHDL or Verilog, results in a lot of low level code. Simulation times can become large. To reduce the complexity of the description and the simulation time, a higher level of abstraction is needed. Model-based hardware design is a technique in which abstraction is used to control the complexity of new designs.

Several model-based design tools enter the market today and the way each of them implement this new level of abstraction is tool-specific. Some tools use a graphical interconnection of blocks while other tools limit the length of handwritten code by using a different programming syntax with a higher level of abstraction. Once the model is described and simulated, automatic code generation can be applied. On one button click, these tools generate HDL code based on the hardware model description. The length, quality and readability of this code are all aspects that will have to be taken in account when choosing the proper code-generation tool for a specific task.

In section 2 we will discuss the related work. In section 3 we will give an overview of the most important tools for model-based hardware design of image processing algorithms. We will discuss a selection of these tools in more detail, namely Xilinx System Generator in sec. 4 and Bluespec in sec. 5. We will describe their capabilities and the image processing experiments that were completed. A conclusion and indication of future work is given in section 6.

2 Related work

According to a lot of reported use-cases, model-based design techniques are increasingly used. A number of recent publications clearly prove the validity of model-based approach for rapid prototyping, especially in the case of complex and computational intensive signal processing algorithms.

The advantages of using a higher abstraction level description are e.g. emphasized in the work of Haldar [3] *et al.*, using Matlab as abstract modelling language, and the work of Papandreou [7] *et al.*, starting from Simulink.

Specifically, in image and video processing applications, where complex algorithms meet stringent execution time demands, the use of model-based design is used frequently in order to facilitate the hardware design process [9, 2]. Toledo [10] *et al.* developed a general framework with Xilinx System Generator components for image processing applications. Also, Stephen Se presents in [8] a nice example of a model based realisation of a complex image processing application, namely David Lowe's well known SIFT feature extractor [5].

Model-based design is also reported for applications in telecommunication [1]. Mülbauer [6] *et al.* and Huet [4] *et al.* prove that this design methodology is very suited to solve the problem of hardware/software partitioning.

3 Tools overview

Several important model-based design tools are already on the market. Table 1 gives an overview. This table also mentions the generated code for each tool. Although most of the tools produce HDL code, the actual generated code output can differ significantly. Multiple factors should be considered when discussing model-based design tools, such as *code-readability*, *length* and *quality*, *ease-of-use* and *learning curve*, *compiler power*, *existing libraries*, *controllability* and *traceability*.

- Readability, length and quality of the generated code are important factors in our evaluation of these tools.
- The ease-of-use and learning curve can also be deciding factors when selecting a tool. For many companies the time-to-market of a product is very important. Thanks to the higher abstraction level used in the model-based hardware design methodology, it is possible to create, test and implement complex algorithms much faster. Note that this speed-up is also determined by the available hardware libraries in the tool.

- Another important factor is the power of the compiler to generate parallelism out of the model-based design code.
- Traceability is also an issue. One would like a straight-forward translation from the model-language to the HDL description.
- Designers should maintain controllability over the generated hardware and software. The abstracter model-based level should leave room for fine-tuning of the generated code, and for specific hardware accelerating methods. Not all tools generate HDL code the same way.

In this paper, we review two model-based hardware design tools in detail: Xilinx System Generator and Bluespec. In future work, we plan to broaden our study to other model-based design tools.

Company	**Product**	**Generated code**
Mathworks	Simulink HDL Coder	HDL Code
	Real-Time Workshop	ANSI C Code, C++
Xilinx	System Generator	HDL Code
	Accel DSP	HDL Code
Silicon Software	VisualApplets	HDL Code
Synplicity	Synplify DSP	HDL Code
Altera	DSP Builder	HDL Code
	SOPC Builder	HDL Code
Univ. Leiden	Compaan/Laura	HDL Code
Bluespec	Bluespec System Verilog	Verilog
Synfora	Pico Express	RTL/SystemC
Agility	Agility MCS	C Code
	DK Design Suite	HDL Code
Scilab	Scicos-HDL	HDL Code

Table 1: Overview of the different model-based hardware design tools

4 Xilinx System Generator

Unlike the standard HDL languages Xilinx System Generator provides a model-based design interface using an extended library of building blocks to create hardware. Rather than at textual code level, Xilinx System Generator takes the abstraction level one step higher, and uses the Mathworks Simulink environment to provide a graphical approach. Connected graphical building blocks and their parameterization form the description of the model.

4.1 Specifications

Xilinx System Generator is integrated as a blockset in the model-based design environment Simulink from the Mathworks. The graphical user interface is powerful, yet easy-to-learn and easy-to-use. System Generator provides the Simulink environment with a list of specific Xilinx building blocks which can be used to create designs optimized for Xilinx FPGA's. The Matlab workspace can be used during simulation, while the hardware itself is modelled using the System Generator blocks.

The Xilinx System Generator blockset is extensive. These blocks range from very simple ones like adders, multipliers and registers to complex blocks such as FFT's, filters and memory interfaces. System Generator is most suited for DSP engineers. Signal processing engineers traditionally explore their algorithms in Matlab/Simulink. With System Generator they now have the capability to start from this environment to implement their system in an automatic way. Additionally to the standard blockset, it is possible to implement user-defined blocks based on self-written Matlab or HDL code. These blocks enable reusing formerly created HDL IP cores.

The system model can be simulated in the Simulink environment. This higher abstraction level reduces the analysis and debugging time. For real hardware testing, Xilinx System Generator supports the possibility to perform hardware in-the-loop co-simulation. Computationally intensive parts of the design can be downloaded into the FPGA and run in hardware, while another part runs on the workstation (PC) itself. This further accelerates the simulation. Interfacing between the Simulink environment (on PC/workstation) and FPGA platform is done using a slower standard JTAG connection or a gigabit Ethernet connection which allows a fast data transfer.

When the design is modelled, automatic VHDL or Verilog code generation can take place. Each library building block corresponds with a generic HDL file. Xilinx System Generator instantiates these HDL files with the correct parameters and applies the needed connections. The generated HDL code is used as entry for the automatically performed implementation steps: logic synthesis and place-and-route.

4.2 Experiments

We tested the possibilities of Xilinx System Generator, by implementing a simple 3×3 convolution filter for streaming video. The block diagram of the considered convolution filter is shown in figure 2. The convolution filter consists of three major functions: line buffers, a multiplier array and adder

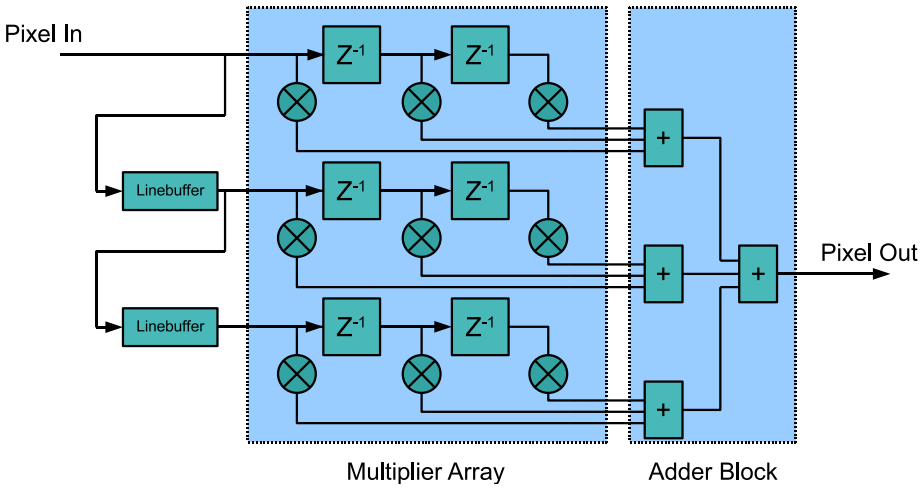

Figure 2: Blockdiagram of the convolution filter

logic.

Note that this is a simplified view, without any pipelining or hardware optimisations. Although this is not a very complex design, it can easily be seen that the use of Xilinx System Generator yields a speedup of the design entry when compared with standard hand-written HDL design. The designer just needs to drag the needed blocks from a library and has to make the correct connections. Changes in the design can be implemented fast. For example, different kernel coefficients can be tested without the need to rewrite any line of code. An example of a 3×3 Gaussian kernel is given below.

$$G_k(x, y) = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

We simulated our designs, using the Matlab workspace to read and write images, so that we could check the functionality of the model more rapidly than with the more conventional HDL testbench. As a means of abstraction, Xilinx System Generator allows to group multiple models into a single subsystem. Using this hierarchical division of the model in subsystems makes the design more comprehensive and easier to overlook. Conversion between Matlab floating point numbers and the hardware fixed-point implementation is done using *gateway_in* and *gateway_out* blocks.

Figure 3 shows our implementation of the convolution filter in Xilinx System Generator.

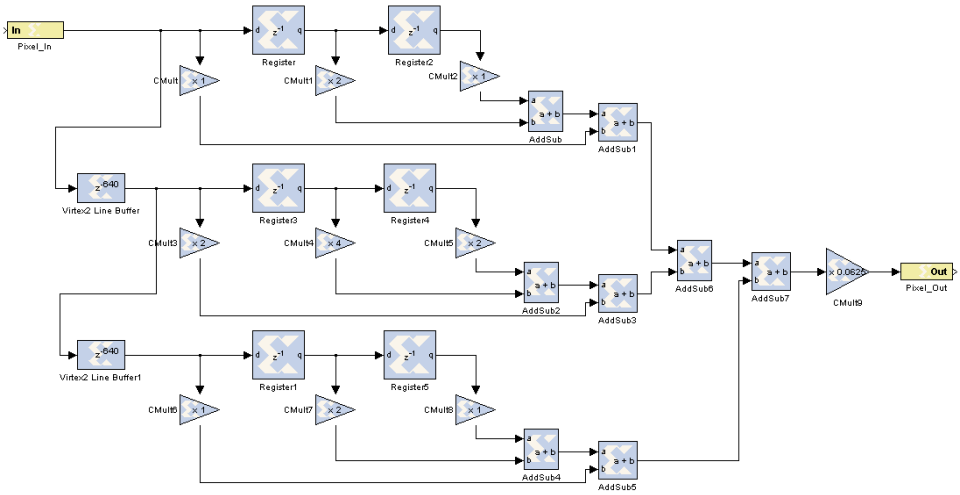

Figure 3: Xilinx System Generator design of the convolution filter

Two line buffers are needed to separate three sequential lines of the input image that needs to be convolved with the 3×3 kernel. The size of the line buffer is chosen according to the resolution of the input image. Xilinx System Generator results in an efficient hardware implementation due to use of optimised hardware blocks during synthesis. As an example, the line buffers are implemented with a Single Port RAM with the Read Before Write Option selected, rather than with a long serial flip-flop chain, thereby fully using the FPGA's capabilities.

4.3 Hardware-in-the-loop

Xilinx System Generator extends Simulink through a direct interface to hardware platforms, hardware-in-the-loop (HIL) co-simulation. Xilinx System Generator automatically generates an FPGA bit stream for a selected part of the model, mostly a computational intensive function. Communication between the hardware platform and the Matlab/Simulink environment is done through a JTAG or a gigabit Ethernet connection. This hardware co-simulation allows the user to exploit the processing power of the FPGA hardware to significantly accelerate simulations. In the meantime the model is validated on the hardware platform.

4.4 Implementation

When simulation results are correct the implementation steps are done automatically by the tool. In this experiment Verilog code was generated for a

Xilinx Spartan-3A DSP 3400A FPGA. On an Intel Core 2 Quad running at 3GHz code is generated in less than 2 minutes. The output results in 6508 lines of Verilog code. Although this number of code lines is rather extensive, a lot of code is allocated to instantiation and port mapping of the different components in the design. When implementing the same design using hand-written code, the length of the code obviously would be shorter but the time spend in writing and debugging it would have been longer. Traceability and readability of this automatically generated code is rather poor. Code generation is realised using the instantiation of generic library blocks which are parameterized based on the settings of the designer. The synthesized design takes an area of 120 slices in the FPGA, and has a maximum clock frequency of 260 MHz.

4.5 System Generator results

Xilinx System generator is an easy-to-learn HDL code generation tool that is very useful when implementing complex signal processing algorithms. The library is extensive, HDL code can be incorporated in the model description to re-use existing IP. The higher level of abstraction of the model description results in reduced simulation times. HIL can be used to further accelerate simulation. Matlab can be used as an input for simulations. The generated HDL code is rather extensive, but when compared to maximum frequency and area, it can compete with hand-written HDL. Traceability and readability of the generated code are rather poor.

5 Bluespec

The Bluespec tool uses an abstract programming language, BSV or Bluespec System Verilog, to realize higher level of abstraction. Standard HDL languages describe hardware at gate or register level. BSV allows the description of a design using so-called *Guarded Atomic Actions*. The hardware design is coded at behaviour level, rather than at clock level.

5.1 Specifications

Unlike standard HDL languages, BSV uses a syntax that describes the behaviour of a design. The designer can code a complex system using simple rules. These rules are coded as if they happen independently of each other. The Bluespec compiler figures out which rules can *fire* simultaneously, while maintaining correct design behaviour. Clock and reset signals are implicit in the design, and thus are not visible to the designer. Input and output

ports aren't defined at signal level as is the case in standard HDL languages. Instead these ports are included in *methods*. These methods are used to define the interface of the hardware to the outside world. Methods do not map into a single input or output port, but consist of a number of input and/or output ports, needed for that specific action. All the functionality is described with modules, registers, wires and own written hardware. Modules have to be instantiated, like objects in an object-oriented programming language. This allows for a more general parameterization than the standard HDL languages.

We will illustrate this with an example. Suppose that a FIFO interface is needed to create a pixel line buffer. In standard HDL one would need to declare input and output ports like clocks, reset, read_enable, write_enable, data_in, data_out and so on. In BSV a FIFO has an interface as follows:

```
interface FIFO#(type a);
     method Action enq (a x);
     method Action deq;
     method a first;
     method Action clear;
endinterface: FIFO
```

As can be seen in this code example, a FIFO has several actions, like enqueuing(enq) and dequeuing(deq). These actions are called using an object-oriented approach, e.g. an element is placed in the FIFO using $fifo.enq(value)$. Bluespec generates the timing logic and enable signals behind this action. No knowledge is needed of the low-level signal negotiation. After the design is completed, one can automatically generate HDL code. The Bluespec compiles and checks every line of the code, and depending on the different rules and their underlying implicit or explicit conditions, the compiler generates HDL code for each of these rules. Depending on the type of module, the compiler knows whether a specific action can take place. For example a register can only be loaded at the rising edge of the clock. In this way a design can be coded at behavioural level.

5.2 Experiments

With Bluespec we have implemented the same 3×3 convolution filter, with a structure shown in figure 2. The line buffers are efficiently implemented in Block RAM in Xilinx FPGA's, using the Bluespec library component. A circular buffer is used in which the read pointer is always one address higher than the write pointer. These line buffers delay the incoming pixels according to the resolution of the input images. Further specifications are similar to the Xilinx System Generator design. The multiplier array,

which implements the filter's kernel, consists of nine constant multipliers. The coefficients are hard-coded in BSV. The adder block consists of three pipelined adders. Input pixels and output pixels of the image are 8 bit. The coefficients, which are smaller than 1 because of normalisation, are implemented as 8 bit fixed point numbers. The multiplier values, which are added during the internal calculation, are truncation to 8 bit at the end stage of the convolution filter.

Compilation of the Bluespec code for the Xilinx Spartan-3A DSP 3400A FPGA target results in 1872 lines of generated HDL code including generated comments. When one understands how the Bluespec compiler translates the Bluespec rules in hardware, the readability of the code is relatively high. Every hand-written line of code is checked for parallelism and is converted into HDL. In this way a name-correlation exists between the different methods, rules and signals in BSV and the corresponding generated HDL code. This results in a good traceability. The synthesis result uses about 127 slices and has a maximum clock frequency of 200 MHz .

5.3 Bluespec results

Bluespec System Verilog has a lot of potential as a model-based design tool. It is rather hard to learn the mechanism behind the compiler, but when one gets the whole concept, Bluespec offers a very efficient model-based design approach. Rather then thinking at logic gate-level, Bluespec abstracts the hardware to its behavioural level. Communication between models is done using methods, and the behaviour is described in different rules. Bluespec figures out which of these rules and methods can be called. The generated code has a high correlation with the BSV code, which implicates that readability and traceability are good.

6 Conclusion

Model-based design is an efficient way to design complex algorithms. Although not yet widely used, it has many advantages. Designs can be prototyped extremely fast, without the need for a low level description and waveform simulations. Standard HDL languages are automatically generated from an abstract model. A designer can concentrate on the behaviour, not on the detailed logic implementation. Model-based design offers a possibility to drastically reduce the design time. The full capability of very complex System-On-Chip hardware can be explored. When using standard HDL languages, debugging and simulation usually take more time than the creation of a description of the actual hardware that has to be build. There

are disadvantages as well. The generated code can be rather large and complex compared to hand-written HDL designs. Readability, traceability and fine-tuning of the generated HDL code can sometimes be a challenging task. When, for example, very specific hardware actions are needed, such as multiple clock domains, the only solution could be to dive back into the generated HDL code.

Different tools are on the market allowing model-based design. Each of them implement the higher level of abstraction in a different way. Some tools use a graphical approach, other use a textual programming language at a higher abstraction level. In this paper we have briefly highlighted two of them: Xilinx System Generator and Bluespec. Table 2 gives a summary of the results of our implementation of a 3×3 convolution filter with Xilinx System Generator and Bluespec.

	System Generator	**Bluespec**
Number of lines	6508	1827
Number of slices	120	127
Max clock freq(MHz)	260	200

Table 2: Comparison of the 3×3 convolution filter implementation

We conclude that Xilinx System Generator is an efficient tool for model-based hardware design of signal processing algorithms, starting from the environment where they are explored and validated. The generated HDL code is rather long and complex. It is a perfect tool when standard applications are designed without the need for hand-written changes. Bluespec has a longer learning curve. It works in a completely different way. Hardware systems are described using several rules in which the Bluespec compiler seeks for parallelism. The generated HDL code has a large correlation with the designers BSV-code.

In the future we plan to investigate more potentially interesting tools for their ability to create model-based hardware designs.

Acknowledgments

The authors greatly acknowledge the financial support of the Flemish Institute for the Advancement of Science in Industry (IWT), Flanders, Belgium.

References

[1] D. Agarwal, C.R. Anderson, P.M. Athanas, *An 8 GHz ultra wideband transceiver prototyping testbed*, The 16th IEEE International Workshop

on Rapid System Prototyping (RSP 2005), pp. 121-127, 8-10 June, 2005.

[2] Daniel Denning , Neil Harold , Malachy Devlin , James Irvine, *Using System Generator to Design a Reconfigurable Video Encryption System*, Lecture Notes in Computer Science, Springer Berlin/Heidelberg, pp. 980-983, ISBN 3-540-40822-3, September 30, 2003.

[3] Malay Haldar, Anshuman Nayak, Alok Choudhary, Prith Banerjee, *A System for Synthesizing Optimized FPGA Hardware from MATLAB*, 2001 International Conference on Computer-Aided Design (ICCAD '01), p. 314, 2001.

[4] S. Huet, E. Casseau, O. Pasquier, *Design exploration and HW/SW rapid prototyping for real-time system design*, in proceedings of the 16th IEEE International Workshop on Rapid System Prototyping, pp. 240-242, 8-10 June, 2005.

[5] David G. Lowe, *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, 60, 2 (2004), pp. 91-110, 2004.

[6] Felix Mühlbauer, Christophe Bobda, *Design and Implementation of an Object Tracker on a Reconfigurable System on Chip,* Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06), p. 230-236, 2006.

[7] Nikolaos Papandreou, Maria Varsamou, Theodore Antonakopoulos, *Transmission Systems Prototyping Based on Stateflow/Simulink Models*, 15th IEEE International Workshop on Rapid System Prototyping (RSP'04), pp. 174-179 2004.

[8] Stephen Se, Tim Barfoot, Piotr Jasiobedzki, *Visual Motion Estimation and Terrain Modeling for Planetary Rovers*, in Proceedings of the International Symposium on Artificial Intelligence for Robotics and Automation in Space (iSAIRAS), Munich, Germany, September 2005.

[9] B. Thornberg, L. Olsson, M. O'Nils, *Optimization of memory allocation for real-time video processing on FPGA*, The 16th IEEE International Workshop on Rapid System Prototyping (RSP 2005), pp. 141-147, 8-10 June, 2005.

[10] C. Vicente-Chicote, A. Toledo Moreo and P. Sánchez-Palma, *Image Processing Application Development: From Rapid Prototyping to SW/HW Co-simulation and Automated Code Generation*, in proceedings of Second Iberian Conference on Pattern Recognition and Image Analysis, IbPRIA 2005, pp. 659-666, Estoril, Portugal, June 7-9, 2005.