

Creating Parameterized and Energy-Efficient System Generator Designs *

Jingzhao Ou, Seonil Choi, Gokul Govindu, and Viktor K. Prasanna
EE - Systems, University of Southern California
{ouj, seonilch, govindu, prasanna}@usc.edu

Abstract

MATLAB/Simulink based system-level tools are becoming popular for FPGA designs these days. One major advantage offered by these tools is that the designer can make use of the ability of MATLAB/Simulink to perform arithmetic-level simulation, which is much faster than the behavioral and architectural simulations in traditional FPGA design flows. As energy efficiency of a design is becoming more and more important, it is desired that the designer can derive energy efficient designs using these design tools. In the paper, we propose a new tool called PyGen, which can be used to develop parameterized and energy efficient designs using MATLAB/Simulink based design tools. The implementation of an adaptive beamforming application is used as an example to illustrate the design flow using this new tool.

1. Introduction

The densities of recent FPGAs have made the development of complex digital signal processing applications possible. The traditional way of describing FPGA designs using hardware description languages (HDL) is too time consuming and prevents the communication between the hardware designer and the algorithm developer. There are many system-level tools these days to address this issue. One example is the system-level design tools based on MATLAB/Simulink, such as *DSP Builder* [1] from Altera and *System Generator* [10] from Xilinx. There are several advantages offered by these tools. First of

all, there is no need to know HDLs. After a design is described in MATLAB/Simulink, the tools can automatically translate it into the corresponding HDL implementation. This allows people from the signal processing community, who are usually familiar with MATLAB/Simulink and unfamiliar with HDL, to get better involved in the whole design process. Second, the designer can make use of the ability of MATLAB/Simulink to perform arithmetic-level simulation, which is much faster than the behavioral and architectural simulations in traditional FPGA design flows. Third, these tools support hardware-in-the-loop simulation and automatic testbench generation, which are useful for further verification on the actual hardware.

The Xilinx *System Generator* for DSP is a high-level design environment built on top of MATLAB/Simulink. Xilinx IP cores and HDL designs are made available through the block set within *System Generator*. A user assembles a design by using the blocks from the block set and connecting them via a GUI. The block set contains (1) blocks that represent the basic FPGA resource such as registers, multiplexers, etc.; (2) blocks that represent control logic, mathematical functions, and memory; (3) blocks that represent proprietary IP cores such as FFT, DCT, etc. Besides, there is a *Resource Estimator* block that allow the users to quickly estimate the resource utilization of their designs.

As FPGAs are more and more widely applied to embedded system designs, energy efficiency of FPGA designs is becoming increasingly important. This is especially critical in the designs of battery operated embedded systems used in many aerospace and military applications. One example is software defined radio (SDR). The processing of many communication algorithms requires high computational ability and might

*This work is supported by the DARPA Power Aware Computing and Communication Program under contract No. F33615-C-00-1633 monitored by Wright Patterson Air Force Base.

consume much power while the wireless base stations and mobile terminals of SDR are working in a power constrained environment. FPGAs stand out as an attractive choice for implementing various functions of SDR due to their high performance, low energy dissipation per unit computation, and reconfigurability.

There are, however, difficulties when applying *System Generator* in FPGA designs. First of all, it is not easy using *System Generator* to describe complex signal processing algorithms, which may contain thousands of blocks. Assembling these blocks by hand can be overwhelming. Second, development of *truly* parameterized designs, which is crucial for design space exploration and design optimization, is not possible using *System Generator*. Even though some parameterization such as precision can be realized using the subsystem masks provided MATLAB/Simulink, it is not possible via the GUI to describe designs in which the collection of blocks and the way that they are connected change depending on the design parameters. Third, even though FPGA resource usage of designs can be estimated using the *Resource Estimator* block in *System Generator*, there is currently no support for rapid energy estimation. The reason for this is because, unlike RISC processors and DSPs, FPGAs are too fine grain to be modeled at high level. Lacking of a high-level architecture hinders rapid energy estimation. There are techniques proposed to overcome this problem. For example, [2] proposes a fast energy estimation technique for reconfigurable architectures based on domain-specific modeling. [6] presents a performance model of reconfigurable System-on-Chip (RSoC) architectures and a dynamic programming based system-level optimization technique for a class of linear pipeline applications. Designs with good time and energy performance can be produced using these techniques. However, since it is not possible to directly extend the block set in *System Generator*, it is not easy for the end users to integrate their own performance models and optimization algorithms into the block set of *System Generator* to derive energy-efficient designs. Finally, the current version of *System Generator* only supports development of point designs. There are no explicit methods to traverse the possible design space and perform system-level optimization.

To address the above problems, we present a new

design tool called *PyGen*, which uses the Python scripting language [7] to develop FPGA designs. Using this new tool, users can describe their designs using Python and translate them to corresponding *System Generator* designs. Parameterized *System Generator* designs can be created using *PyGen*. We also integrate performance models and a system-level optimization technique into this tool. Then, users are able to create energy-efficient designs on *System Generator*. As an example, we implement an adaptive beamforming application using our tool. This adaptive beamforming application is widely used in the base stations of software defined radio to better exploit the limited radio spectrum [8]. These base stations are placed in inaccessible and distributed locations and need to perform a large amount of computation, which dissipates a lot of energy. Thus, energy efficiency is crucial in the implementation of this beamforming application.

The organization of this paper is as follows. Section 2 discusses related work. Section 3 presents the design methodology of *PyGen*. Illustrative design examples using *PyGen* are given out in Section illustrative-examples. Finally, we conclude in Section 5.

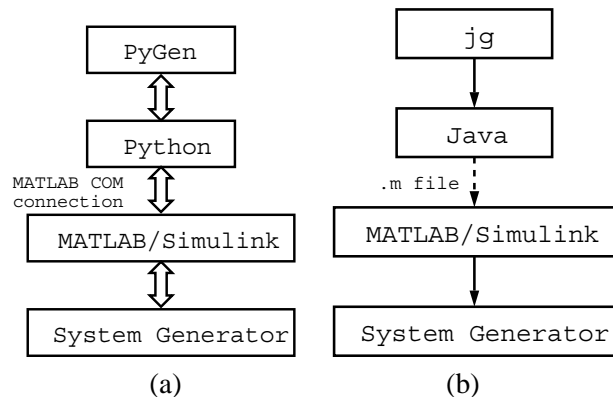


Figure 1. Architecture of (a) *PyGen* and (b) *jg*

2 Related Work

There are system-level design tools such as DK2 [3] from Celoxica and Forge [4] from Xilinx which use high-level languages similar to C and Java for describing FPGA designs. When using these tools, the users describe their applications in C or Java and rely on the compiler to infer the appropriate architectures to im-

plement these applications and perform optimization such as loop unrolling, pipelining. This is different from system-level tools based on MATLAB/Simulink, which provide a high-level abstraction of the underlying hardware resource and allow the end users to describe the data flow and its hardware realization through the high-level abstraction. *PyGen* manipulates this high-level abstraction using Python scripting language.

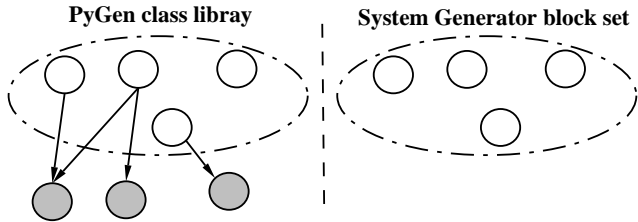


Figure 2. *PyGen* class library and the corresponding *System Generator* block set

jpg, a tool developed by Xilinx, uses Java to describe a *System Generator* design [9]. For designs using *jpg*, users compile their Java code, generate MATLAB programs and execute them in *System Generator*. This design process prevents effective debugging, which is crucial for complex designs. Also, rapid energy estimation and system-level optimization is not supported by the current version of *jpg*. As is illustrated in Figure 1(b), due to lack of mechanisms to obtain information from *System Generator*, *jpg* is unable to support the rapid energy estimation techniques proposed in [2].

3. Methodology

The architecture of *PyGen* is illustrated in Figure 1(a). The lines with arrows on both ends mean that *PyGen* and *System Generator* can obtain information from each other (*two-way communication*). The design flow is shown in Figure 3. The shaded blocks in the figure specify the add-ons of *PyGen* to the original design flow. A *System Generator* design begins by either manipulating the GUI in *System Generator* or writing Python code in *PyGen*. Since *PyGen* is a Python package, users need to import it using the Python script `import PyGen` before describing their designs using it. *PyGen* contains a module that connects it to MATLAB and a class library whose classes are used to construct user designs. The mod-

ule provides the mechanism for Python to interact with MATLAB and *System Generator*. All the blocks in the *System Generator* block set have their corresponding classes in the *PyGen* class library (see Figure 2). The class library can also be extended to derive parameterized designs. Users describe their designs by instantiating classes in the class library, which is equivalent to dragging and dropping blocks from the *System Generator* block set to the user designs. After describing a design in *PyGen*, a corresponding diagram in *System Generator* is created. The performance model and the system-level optimization technique in [6] are integrated into *PyGen*. Thus, users can estimate the performance of their designs, such as area, energy consumption, etc. Users can also optimize and modify their designs based on their design requirements. Once the final design is identified, users can follow the remaining design flow of *System Generator* to synthesize their designs into target devices.

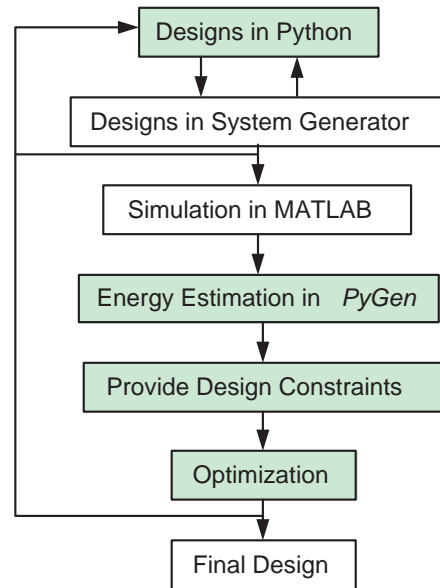


Figure 3. *PyGen* design flow

3.1 The *PyGen* Module

To allow for MATLAB and Python to obtain information from each other, we use the MATLAB COM (Component Object Model) server for communication. We have developed a module within *PyGen* that connects it to the MATLAB COM server. This enables

access to the methods in Simulink and *System Generator* and allows us to manipulate the *System Generator* block set. After a design is described using Python, the module calls the appropriate functions in MATLAB and automatically translates the design into a corresponding design in *System Generator*. Any changes in *System Generator* will also be reflected in *PyGen* through this module. Since Python is an interactive language, the design flow of *PyGen* is also interactive and is different from that of *tg* illustrated in Figure 1(b), which is non-interactive. Using *PyGen*, users can more easily develop their designs step-by-step and make necessary changes as the design evolves in *System Generator* during the development period. Performance tuning of the designs using the techniques discussed in Section 3.3 and Section 3.4 is also easier in *PyGen* due to its interactivity.

3.2 Parameterized Designs

Using the flexible classes, very high level dynamic data types, and the dynamic typing provided by Python, the class library in *PyGen* can be easily extended to represent new parameterized designs. By leveraging the object-oriented class inheritance, users are able to derive classes that represent designs in which the connections of the blocks change depending on the design parameters. Besides, users can apply class encapsulation to their own classes and expose only those design parameters that interest them. In the example considered in this paper, a new class *CompMAC* is derived with degree of parallelism as its parameter. This parameter is not available in any of the *System Generator* blocks.

3.3 Performance Modeling

We integrate the domain specific modeling technique in [2] as well as the performance model in [6] into *PyGen*. This is achieved by using the object-oriented programming concepts such as inheritance offered by Python to extend *PyGen* class library. Thus, additional information on the utilization of FPGA resources and the energy consumption obtained through techniques such as domain-specific modeling, can be associated with the new classes. Methods, such as `GetPerformance()`, etc., are provided in *PyGen* to obtain performance values of the object.

3.4 System-Level Optimization

The extended *PyGen* classes and objects contain performance values of the designs they represent. This enables system-level optimization in *PyGen*. A class `SysOpt` is provided in *PyGen*. If users use a class that is inherited from `SysOpt`, they can input the design requirements using the methods provided by it. Currently, `SysOpt` uses the dynamic programming algorithm in [6]. By invoking the `optimize()` method of the instantiated object, users can find out the optimized design based on the design requirements. The parameters of the object will be automatically set according to the results from the optimization algorithm. By inheriting the `SysOpt` class and overriding the related methods, users can also implement their own optimization algorithms.

4. Illustrative Examples

In this section, we will demonstrate the design flow using *PyGen* by showing the design of an LMS (Least Mean Squared) MVDR (Minimum Variance Distortionless Response) adaptive beamforming application.

4.1 Design of a Parametrized MAC Component

One important part of the LMS-based MVDR adaptive beamforming application [5] is the complex-number MAC (Multiply-and-Accumulate) component. The parameterization of the beamforming application requires that degree of parallelism and precision of this MAC component can be parameterized first. Appendix I presents the Python code that can be used in *PyGen* to generate this basic component with different parameters. Instantiating the *CompMAC* class with `par=4` and `par=8` automatically generates *System Generator* designs representing the complex number MAC with 4 and 8 inputs. These are shown in Figure 4. By calling the `GetPerformance()` method associated with the *CompMAC* object, performance estimates of these complex number MACs on Virtex-II XC2V3000, can be obtained through domain-specific modeling, as shown in Table 1.

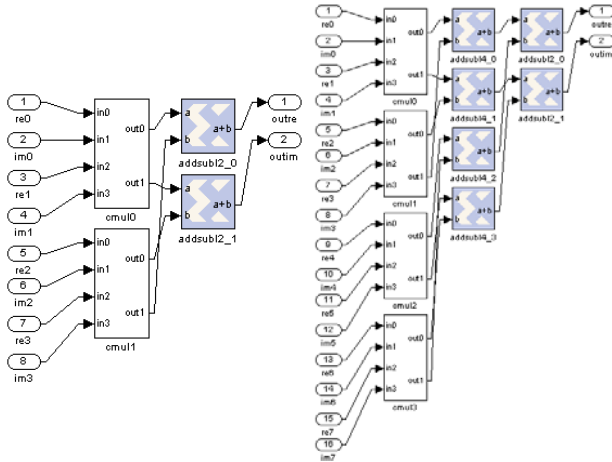


Figure 4. Complex number MAC System Generator designs with different inputs generated by PyGen

Table 1. Estimates of the complex MAC component for various input sizes (110 MHz)

Input size	4	8	16
Energy (nJ)	1.43	4.16	11.13
Area (slice/mult)	128/8	304/16	688/32
Time (cycle)	3	4	5

4.2 Design of an MVDR Adaptive Beamforming Application

The task graph of the adaptive beamforming application is shown in Figure 5, which consists of three tasks. The filtering task is executed for each of the sample data while the LMS adaptive beamforming task and the weight coefficient task are executed every 8 data samples. The implementation of the application should be able to sustain an input data rate of 105 mega samples per second.

To implement the application, we create a Python class named MDVR. Using different `par` and `pre` (precision) values to instantiate this class, System Generator designs of the beamforming application using different complex number MAC architectures and with different input/output precision are created (see Figure 6). The performance of these different designs on Virtex-II XC2V3000 is obtained through low-level simulation using Xilinx ISE 5.2i and XPower. It is shown in Table 2.

We use the `SysOpt` class to optimize the design and obtain the parameter values of the MDVR object

that lead to the most energy-efficient design. The design trade-offs are analyzed as below. Since filtering is the most time critical task and thus should have the full degree of parallelism in the design in order to satisfy the throughput requirement. Due to the resource constraints of the target device, the designs for other tasks have a smaller degree of parallelism in order to just use the dedicated multipliers for computation.

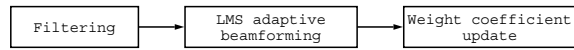


Figure 5. Task graph of the MVDR beamforming application

5. Conclusion

A novel design tool called *PyGen* is proposed in this paper. Results are provided to illustrate how parameterized and energy-efficient designs can be derived using this new tool. The development of *PyGen* is still in progress. Our future work will include extending to tackle energy efficient designs using platform FPGAs, which incorporate FPGA fabric, embedded processors and memory systems on a single chip.

References

- [1] DSP Builder, Altera, Inc., <http://www.altera.com/products/software/system/products/dsp/dsp-builder.html>.
- [2] S. Choi, J.-W. Jang, S. Mohanty, V. K. Prasanna, "Domain-Specific Modeling for Rapid Energy Estimation of Reconfigurable Architectures," *ERSA*, 2002.
- [3] DK2, Celoxica, Inc., <http://www.celoxica.com/products/tools/dk.asp>.
- [4] Forge, Xilinx, Inc., <http://www.xilinx.com/ise/advanced/forge.htm>.
- [5] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, 2002.
- [6] J. Ou, S. Choi, V. K. Prasanna, "Performance Modeling of Reconfigurable SoC Architectures

Table 2. Simulation results of LMS MVDR beamforming with various precisions (110 MHz)

Precision (bit)	8	16
Power (mW)	437	698
Area (slice/mult/BRAM)	870/48/3	1608/53/3
Time (cycle)	18	18

* Slice stands for the number of slices used. Mult stands for the number of embedded multipliers used. BRAM stands for the number of Block RAMs used.

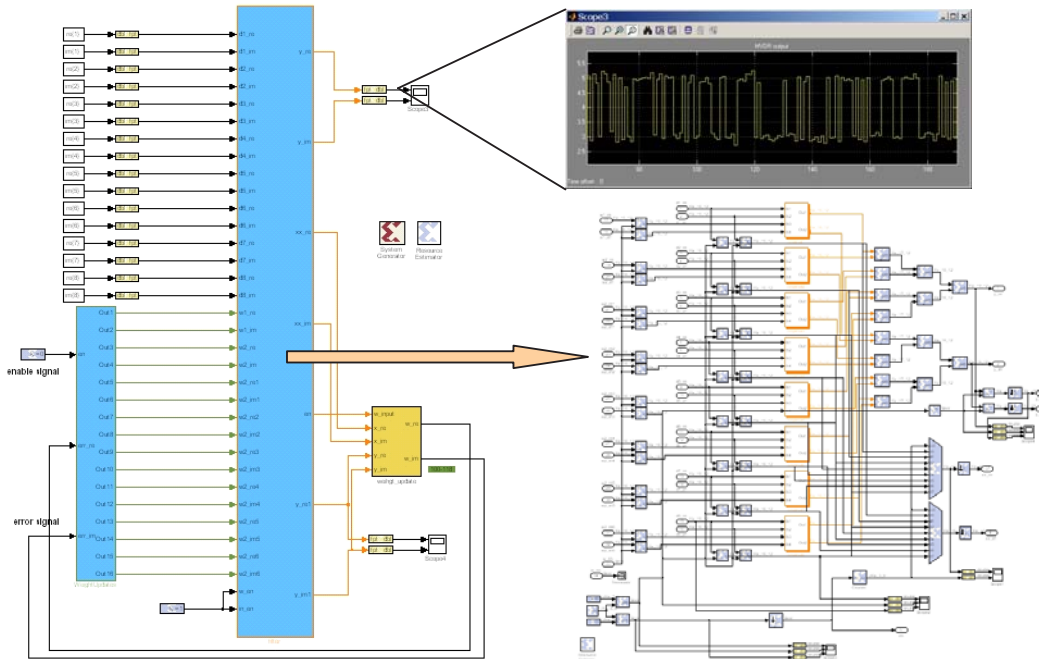


Figure 6. MVDR beamforming application designed using *PyGen*

and Energy-Efficient Mapping of A Class of Applications,” *FCCM*, 2003.

[xil_prodcart_product.jsp?title=system_generator](http://www.xilinx.com/xil_prodcart_product.jsp?title=system_generator)

[7] Python, <http://www.python.org>.

[8] J. Razavilar, F. Rashid-Farrokhi, and K. J. R. Liu, “Software Radio Architecture with Smart Antennas: A Tutorial on Algorithms and Complexity,” *IEEE JSAC*, Vol. 17, No.4, 1999,

[9] J. Stroomer, J. Ballagh, H. Ma, B. Milne, J. Hwang, N. Shirazi, “Creating System Generator Design Using *kg*,” *FCCM*, 2003.

[10] Xilinx, *System Generator for DSP*, <http://www.xilinx.com/xlnx/>

Appendix I: Python code for the parameterized MAC design

```

class CompMAC(PyGenBlock):
    def __init__(self, name, par):
        PyGenBlock.__init__(self, name)
        self.Blocks(par)
        self.Links(par)

    def Blocks(self, par):
        # add the input ports
        self.inp = []
        for i in range(par):
            self.inp.extend([self.add(InPort, 're' + str(i)),
                             self.add(InPort, 'im' + str(i))])
        # add the complex number multiplication sub systems
        self.cmul = []; self.nextcol()
        for i in range(par>>1):
            self.cmul.append(self.addsubsys(CompMult, 'cmul' + str(i)))
        # add the addsub blocks
        self.addsub = []; count = 0; col = par>>1
        while col > 1:
            self.nextcol()
            for i in range(col):
                self.addsub.append(self.add(AddSub, 'addsub1' + str(col) + '_' + str(i)))
            col >>= 1;
        self.nextcol()
        # add the output ports
        self.outp = [self.add(OutPort, 'outre'), self.add(OutPort, 'outim')]

    def Links(self, par):
        # links between the input ports and the complex MAC
        for i in range(0, par<<1, 2):
            self.addlink(self.inp[i], 1, self.cmul[i/4], i%4+1)
            self.addlink(self.inp[i+1], 1, self.cmul[(i+1)/4], (i+1)%4+1)
        # links between the complex MAC and the adders
        for i in range(0, par>>1, 2):
            self.addlink(self.cmul[i], 1, self.addsub[i], 1)
            self.addlink(self.cmul[i], 2, self.addsub[i+1], 1)
            self.addlink(self.cmul[i+1], 1, self.addsub[i+1], 2)
            self.addlink(self.cmul[i+1], 2, self.addsub[i], 2)
        # links between the adders
        col, o1, o2 = par>>1, 0, par>>1;
        while col > 2:
            for i, j in zip(range(0, col, 4), range(0, col>>1, 2)):
                try:
                    self.addlink(self.addsub[i+o1], 1, self.addsub[j+o2], 1)
                    self.addlink(self.addsub[i+o1+1], 1, self.addsub[j+o2+1], 1)
                    self.addlink(self.addsub[i+o1+2], 1, self.addsub[j+o2], 2)
                    self.addlink(self.addsub[i+o1+3], 1, self.addsub[j+o2+1], 2)
                except: pass
            o1 += col; col >>= 1; o2 += col;
        # links between the adders and the output ports
        self.addlink(self.addsub[-2], 1, self.outp[0], 1)
        self.addlink(self.addsub[-1], 1, self.outp[1], 1)

```