

# 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

## 1. Introduction

Xilinx *System Generator* for DSP [7] is a high-level design environment built on top of Mathwork's MATLAB/Simulink. Xilinx IP cores, designs in VHDL or Verilog, and MATLAB functions 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. There are, however, difficulties with *System Generator*. For example, when a design contains many blocks, assembling them by hand can be overwhelming. Also, 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.

For designs using FPGAs, energy-efficiency becomes increasingly important. This is especially critical in the designs of battery operated embedded systems used in many aerospace and military applications. [1] addresses this issue and proposes a fast energy estimation technique for reconfigurable architectures based on domain-specific modeling. [2] 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*, these techniques cannot be integrated into it to derive energy-efficient designs. New mechanisms need to be added to *System Generator* to support such integration. *tg*, a tool developed by Xilinx, uses Java to describe a *System Generator* design [6]. For designs using *tg*, 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.

In this paper, we present a new design tool using the Python scripting language [3], called *PyGen*. Using *PyGen*, 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. Thus, users can use *PyGen* to create energy-efficient designs. As an example, we implement an adaptive beamforming application using *PyGen*. This adaptive beamforming application is widely used in the base stations of software defined radio to better exploit the limited radio spectrum [4]. 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 the beamforming application.

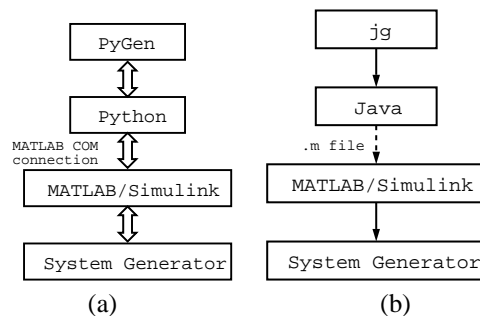


Figure 1. Design flow using (a) *PyGen* and (b) *tg*

## 2. Methodology

The design flow using *PyGen* is illustrated in Figure 1(a). 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 module 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

\*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.

*System Generator* is created. The performance model and the system-level optimization technique in [2] 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 in *PyGen* 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.

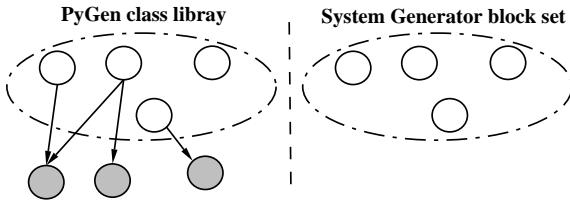


Figure 2. *PyGen* class library and the corresponding *System Generator* block set

## 2.1 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 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 2.3 and Section 2.4 is also easier in *PyGen* due to its interactivity.

## 2.2 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.

## 2.3 Performance Modeling

We integrate the domain specific modeling technique in [1] as well as the performance model in [2] 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.

## 2.4 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 [2]. 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.

## 3. Illustrative Examples

To illustrate the effectiveness of our *PyGen* tool, we implement an LMS-based MVDR adaptive beamforming application [5]. The basic element in the architecture of the application is a complex number multiply-and-accumulate (MAC). Appendix I presents the Python code that can be used in *PyGen* to generate this basic architecture. 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 3. 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.

We implement 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

Table 1. Estimates of complex MAC architecture for various input sizes (50 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

Table 2. Simulation results of LMS MVDR beamforming with various precisions (33 MHz)

Precision (bit)	8	16
Energy (nJ)	47.24	79.52
Area (slice/mult/BRAM)	800/44/10	1437/44/10
Time (cycle)	8	8

\* 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.

and with different input/output precision are created (see Figure 4). 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. In the full version of the paper, we will present the results using the `SystemOp` class to obtain the parameter values of the MDVR object that lead to the most energy-efficient design.

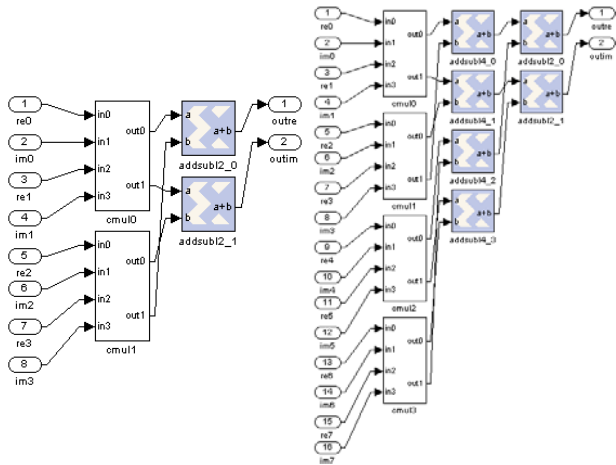


Figure 3. Complex number MAC *System Generator* designs with different inputs generated by *PyGen*

## 4. Conclusion

A novel design tool called *PyGen* is proposed in this paper. Results are provided to show that parameterized and energy-efficient designs can be easily derived using *PyGen*.

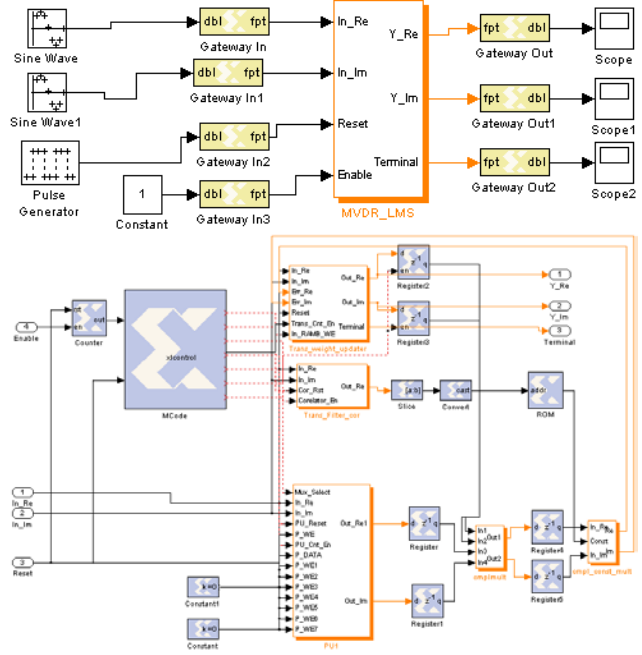


Figure 4. MVDR beamforming application designed using *PyGen*

## References

- [1] S. Choi, J.-W. Jang, S. Mohanty, V. K. Prasanna, "Domain-Specific Modeling for Rapid Energy Estimation of Reconfigurable Architectures," *ERSA*, 2002.
- [2] J. Ou, S. Choi, V. K. Prasanna, "Performance Modeling of Reconfigurable SoC Architectures and Energy-Efficient Mapping of A Class of Applications," *FCCM*, 2003.
- [3] Python, <http://www.python.org>.
- [4] 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.
- [5] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, 2002.
- [6] J. Stroomer, J. Ballagh, H. Ma, B. Milne, J. Hwang, N. Shirazi, "Creating System Generator Design Using *jpg*," *FCCM*, 2003.
- [7] Xilinx, *System Generator for DSP*, [http://www.xilinx.com/xlnx/xil\\_prodcat\\_product.jsp?title=system\\_generator](http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_generator)

## Appendix I

```
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, 'addsubl' + 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)
```