CASPER Memo 11: Xilinx System Generator for DSP in the CASPER Group

Aaron Parsons, Daniel Chapman, Henry Chen

Jan. 23, 2007

1 Introduction

Xilinx System Generator (XSG) for the Matlab/Simulink environment is a powerful tool for generating digital signal processing (DSP) algorithms which map into FPGA hardware. Currently in the Center for Astronomy Signal Processing and Electronics Research (CASPER) group, we use XSG to generate most of the DSP hardware used in our spectrometers, correlators, beamformers, etc.

2 Getting Started

You will need a BWRC account (talk to Tom Boot) to access any of the intel2650-[1-9].eecs.berkeley.edu servers which contain the custom Simulink environment we have developed. Once you log onto one of these machines, you will want to map a network drive (from My Computer) to set up z: as "\\hitz\designs\BEE". Directories of interest on this drive are mlib (a repository for stable libraries), mlib_devel (for the lastest, sometimes buggy libraries), and mlib_devel_8_2 (a work-in-progress for updating our libraries to the latest version of XSG). Inside any of these directories, run the appropriate *.bat file to start Matlab with the appropriate evironment. Also available on the z drive is a "projects" directory, which is a repository for chip designs. Make yourself a directory here for any applications you might be working on. Also under the projects directory is a "astro_library_testbenches", which is where we keep the testbenches for any library blocks we develop. Whenever editing an existing library block, refer to the appropriate testbench in this directory for verifying its functionality. Similarly, for any new library blocks which you may develop, place a testbench file here.

Once you have loaded Matlab from one of the mlib directories, type "simulink" at the command prompt to load the default XSG libraries. Under the list of libraries, there is one called "Simulink" which is useful for constructing testbenches, but does not map into FPGA hardware. Blocks here (scopes, constants, sine waves, etc.), are vital for taking advantage for the rich test environment that Simulink makes available. Below this library are a series of "BEE2" libraries which are custom libraries we have developed. For full descriptions of the functionality in many of these

libraries, please refer to any of the several overview papers which we have published on our group (available at http://seti.berkeley.edu/casper). In summary, the "BEE2 System Blockset" provides interfaces to the on-chip CPU, and to off-chip hardware specific for boards which have been ported to the CASPER toolflow (which will be discussed later). The rest of the BEE2 libraries provide parameterized DSP blocks. The last library of interest is the "Xilinx Blockset" library, which is provides the fundamental set of blocks which map to FPGA hardware. Under this library are things like counters, adders, multipliers, registers, and tons of other things. Besides the major DSP blocks we have constructed like Polyphase Filter Banks, FFTs, X Engines, etc., most of the tools you'll need to design for FPGAs are in this library.

3 Toolflow

XSG has limited support for state-machines, clock boundaries, and asynchronous interaction, and so we have developed a library of VHDL interfaces with XSG representations for interfacing between DSP components of the design and other hardware components of the FPGA chip and surrounding devices. Blocks in this library are generally colored yellow. To compile a design for a CASPER-compatible board, drag an "XSG core config" block from the System Blockset library. This block will allow you choose the board you are compiling for, as well as the clock rate you're shooting for and the synthesis tool to use for compiling a netlist. Once you've made an XSG design using yellow blocks and DSP, you can compile it by running "bee_xps" from the Matlab command line. This window should have your design name on top (make sure it is the base name of the design you want to compile), and a bunch of compile steps with checkboxes. By clicking "Run XPS", your design will be synthesized through XSG, interfaced to all the VHDL drivers called for by the yellow blocks, compiled to a netlist with XST/Synplify, mapped into FPGA hardware, routed to make timing, and a bit file will be generated. Futhermore, a basic set of software will be compiled (to run on one of the CPUs associated with this FPGA) for talking to your design.

More on BORPH, Bit Files, Programming FPGAs...

4 Generic FPGA Design Concepts

4.1 Binary Point

In decimal, a number is formulated by multiplying each character by a power of 10. We know which number is multiplied by 10^0 because it lies immediately to the left of the decimal point, for example:

$$\begin{aligned} 462.15 &= 4 \times 10^2 + 6 \times 10^1 + 2 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} \\ &= 400 + 60 + 2 + \frac{1}{10} + \frac{5}{100} \\ &= 462.15 \end{aligned}$$

In binary, the number multiplied by 2^0 lies immediately to the left of the **binary point**.

$$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

$$
\begin{aligned}
&= 4 + 0 + 1 + \frac{0}{2} + \frac{1}{4} \\
&= 5.25 \ (\textit{in decimal})
\end{aligned}
$$

When programming in simulink, we often need to know the specifics of a number, namely how many bits it is made up of and where the binary point lies. Simulink displays this information using the format n_m, where n is the bit width of the number and m is how many bits lie to the right of the binary point. For example, simulink describes 101.01 as a 5_2 number. A data type is called fixed-point when the binary point is fixed in one location.

4.2 Negative Numbers

In mathematics, we represent a negative number by adding a '-' character to the front of it. A computer deals only with 0's and 1's, so it must use another method. The most obvious solution is to simply use the most significant bit to represent the sign, a method known as sign-magnitude, but it has problems. One is that the number zero is represented twice (+0 and -0), and another is that accounting for these negative numbers in addition and subtraction requires extra circuitry.

A better solution is a method known as **two's complement**, which is what computers generally use. To negate a number in this system, you invert the number bit-wise and then add 1 to it. Additionally, a number whose most significant bit is '1' is defined to be negative. The result of these rules is that the "higher half" of the unsigned numbers wrap around and become the negative numbers. With 3 bits, we have:

	000	001	010	011	100	101	110	111
unsigned	0	1	2	3	4	5	6	7
sign-magnitude	0	1	2	3	-0	-1	-2	-3
two's complement	0	1	2	3	-4	-3	-2	-1

What is negative zero?? It's a lie, and it's also a bug waiting to happen. Note that the two's complement system only represented zero once. What's more, adding and subtracting two's complement numbers the normal way (as if they were unsigned) produces the correct result. In Simulink, only unsigned and two's complement numbers are supported.

5 Simulink Basics

Simulink consists of 2 realms: pure floating-point simulation, and fixed-point cycle-accurate simulation. Only the latter compiles into an FPGA. If you turn on "Format→Port/Signal Displays→Port Data Types" (which you should always do for your designs), you'll see that some lines in your design are labeled "double", which means they are for pure simulation, and others are "UFix_8_0", or "Fix_8_7", or whatever, and they map into FPGAs. Simulink is smart about data type propagation and knows, for example, that adding a Fix_8_7 to a UFix_3_0 results in a Fix_12_7 number. Most of the time, this is what you want. When it's not, most blocks have configuration parameters (avail-

able in their mask, which you can access by double-clicking the block) to set the output precision and the rounding/truncation method.

Masks are excellent ways of providing simple interfaces to complex blocks. You can create your own mask by selecting a group of blocks, right-clicking, and selecting "Create Subsystem". This will make a sub-block which hides your blocks underneath. Once you have created a subsystem, you can then right-click and select "Mask Subsystem" to create a masked block. Masks allow you to create an interface for passing variables (under "Parameters") which will then exist in the namespace for any subblock underneath. Note that to resolve a variable's value, Simulink will look to higher and higher containing blocks, and eventually will look into the namespace at the Matlab prompt. Once you have created variables under the Parameters tab, you may use those variable names in the place of numbers for any blocks underneath. You may also use these variables in the "Initializaiton" tab, which executes Matlab code before trying to evaluate anything underneath this block. You can use this to make new variable with new values. Later we will show you how to use this field to create blocks which dynamically draw subblocks.

5.1 Documentation

The documentation pane of the Simulink Mask Editor gives three fields: Mask type, Mask description, and Mask help. The first two are displayed when you double-click on a masked block. What you put in Mask help is available when you right-click on a mask and select "Help" from the popup menu.

The help files for each block should be written as a separate HTML file; feel free to make full use of HTML's formatting capabilities and include images if useful. The HTML file and all supporting files should be put in the "doc" folder in casper_library. To allow Simulink to open the help file in the built-in help browser, only the following line is needed in the "Mask help" field:

```
eval(‘xlWeb([getenv("MLIB_ROOT"),"\casper_library\doc\filename.html"])’)
```

For examples of how to document blocks see the BEE2 ADCScope and CIC blocks.

6 Simulink Mask Scripting

Masks are good for beginning the process of parameterizing designs. They provide simple interfaces and propagate variables. However, sometimes designs need to be parameterized in ways which cannot be implemented with simple variable propagation. For example, many designs (FFTs, X Engines, FIRs) require that a stage be tiled a number of times which is determined from one of the input parameters. To address this parameterization need, we make use of some "under the hood" Simulink/Matlab commands for implementing blocks, connecting wires, and setting parameters dynamically. Basically, this is a hack of the Simulink environment, and as a result, is very tricky. Through a large amount of trial and error, we have come upon a methodology which side-steps various bugs, instabilities, and inefficiencies in the Simulink language.

6.1 Writing an _init Function

Each block which is dynamically drawn needs set of commands, called from the "Initialization" tab in its mask, which make the appropriate changes based on input parameters. The testing and debugging process is aided if these parameters exist as a separate function in an *.m file which can be called either from the command line or from the mask. Our convention is to name these files "blockname_init.m" with the function being "blockname_init". This function should accept 2 arguments: "blk", and "varargin". "blk" is a reference to the block which is being edited, and "varargin" is a cell array of variable name/value pairs. Thus, a typical block init function definition might be:

```
function typ_blk_init(blk, varargin)
```

And this init function would be called as:

```
typ_blk_init(gcb, 'varname1', 'value1', 'varname2', 'value2')
```

"gcb" is a built-in function for returning a reference to the current active block.

In the definition of your init function, immediately following the declaration, should be a bunch of comments which define the usage of your function. It should describe what kind of block this init function operates on, and what varnames and values are required. Documentation is important in order for you parameterized libraries to be useful to others. Please do it.

If you would like to enter default values for any of these variables, you may use the following line:

```
defaults = {'varname1', value1, 'varname2', value2}
```

The very next thing that your init function should contain after usage documentation and default values is the following line:

```
if same_state(blk, 'defaults', defaults, varargin{:}), return, end
```

"same_state" is a hash function written by our group which hashes varargin to detect if this block is already configured for the supplied parameters. If you have not defined defaults, omit the defaults keyword and variable. This command is always paired with the following command, which should always be the last line in your init function:

```
save_state(blk, varargin{:});
```

This function stores a hash of varargin in the "UserData' parameter, which is accessed by "same_state" for determining whether the block is already appropriately configured. This solves an inefficiency problem with Simulink. Whenever Simulink prepares a design for simulation or compilation, each

mask initialization is evaluated several (usually 4) times. Each time a mask initialization is evaluated, all masked subblocks have their mask initializations evaluated several times. The result is that for heirarchical designs (the kind we like to build) with many levels of nested blocks evaluate masks a number of times which grows exponentially with the number of nested blocks. The answer is to bypass repetitive mask evaluation by detecting if a block is already configured appropriately, and returning immediately if so.

The next line after "same_state" should be:

```
check_mask_type(blk, 'blk_type');
```

The string you enter here should match what you enter in "Mask Type" under the "Documentation" tab of your mask. This line ensures that if an incorrect block is accidentally selected when you call an init function, it will not destroy the contents of that block.

After checking your mask type, call:

```
munge_block(blk, varargin{:});
```

This function disables a block's link with its library representation. Simulink seems to have a problem with having active library links with dynamically draw blocks, and this prevents the ensuing headache. However, the tradeoff is that if a library block is updated, the changes are not propagated to an instance of the block unless you right-click the block and select "Link Options→Restore Link→Use library block". C'est la vie. This function also has the ability to statically configure all subblocks if varargin has the variable "munge" with value "dumbdown", or to unmask all subblocks if "munge" is set to "unmask". Use these with caution.

Next in your init function you need to retrieve all the variables you need from varargin using "get_var", which works like this:

```
var = get_var('varname', 'defaults', defaults, varargin{:});
```

Again, if you do not have default values defined, you may omit the keyword "defaults" and the defaults variable. In general, you should check the values returned by "get_var" for appropriate values, and call "error" with some descriptive text if otherwise.

At some point, before you start drawing blocks, you need to call:

```
delete_lines(blk);
```

You can guess what it does. Whenever you want to instantiate a block, call:

```
reuse_block(blk, 'blkname', 'library/location', 'varname', 'value', ...);
```

This function checks to see if a block by that name already exists, and if it does, it configures it with the "varname", "value" pairs which follow. If it does not exist, this function then creates an instance from the library and configures it appropriately. It turns out that instantiating a block from the library can be an expensive operation, so this function avoids doing it when possible. As an FYI, in XSG 7.X, all of the standard Xilinx blocks are in the "xbsIndex_r3" library (you can always right-click a Xilinx block and say "Link Options→Go To Library Block"). Under XSG 8.X, the library incremented to "xbsIndex_r4". Migration of many of our blocks to XSG 8.X may be as simple as replacing "r3" with "r4", but the default parameters of some of the blocks in the library may have changed, so this will require extensive testing to make sure that nothing breaks. My recommendation is that all new block should be written using "r4".

One of the variables you should set is "Position", which should be set to a 4 element array [x_left, y_top, x_right, y_bottom]. If you've already placed a block and want to know where it is, you can always run "get_param(gcb, 'Position')" from the Matlab command line. Finally, although you can set any variable to a string which is the name of another variable in your namespace (equivalent to typing the variable into the mask of that block), it is preferred to set the variable to the "num2str(var)" of another variable. This makes it easier to see how a block is configured when clicking around and may (?) speed design updates.

Lines may be added to your design as follows:

```
add_line(blk, 'blk1/1', 'blk2/1');
```

where the number after the slash represents the port number of the block in question (counting from 1). By default, this draws straight lines. You can turn "Autorouting" "on" if you want, but I like straight lines. It lets everyone know you did this with a script, not by hand.

Finally, after you've reused all your blocks and added all your lines, you need to call:

```
clean_blocks(blk);
```

This function finds any blocks which do not have all their inputs connected and deletes them. This way, you do not have to worry about what subblocks may have been in your block when you started drawing. You simply place and connect the blocks you want, and this function automatically clears out the rest.

If you desire, you may add the following to your script:

```
set_param(blk, 'AttributesFormatString', 'some note about config');
```

This puts a little blurb of text underneath then name of your block so that anyone can easily see its current configuration. Useful.

And don't forget to put that "save_state" and the end of your script.

Fin.

6.2 More About Masks

Here is a collection of random facts which may help you in your adventures with masks.

You may want to know what the names of variables are in XSG blocks provided by Xilinx. Open the "xbsIndex" library (usually by right-clicking a Xilinx block, and following "Link Options→Go To Library Block"), unlock the library (under the "Edit" menu), and then right-click "Edit Mask" the block you are interested in.

If you want all the dirty details about what variables are available in masked functions in general, you can get a list of them by typing:

```
get_param(gcb, 'ObjectParameters');
```

We have a convention that all new library blocks which are intended for general use be colored a pastel "ghoulish" green (CASPER, get it?). Just right-click "Background Color→Custom". Blocks which are just subblocks should be colored gray.

Read the init functions of other blocks for ideas. Don't forget to put your init function in the "Initialization" tab of your mask, calling it with the variable names listed in the "Parameters" tab.

And for goodness sakes, document.