2010 CASPER Workshop

Tutorial 1: Introduction to Simulink

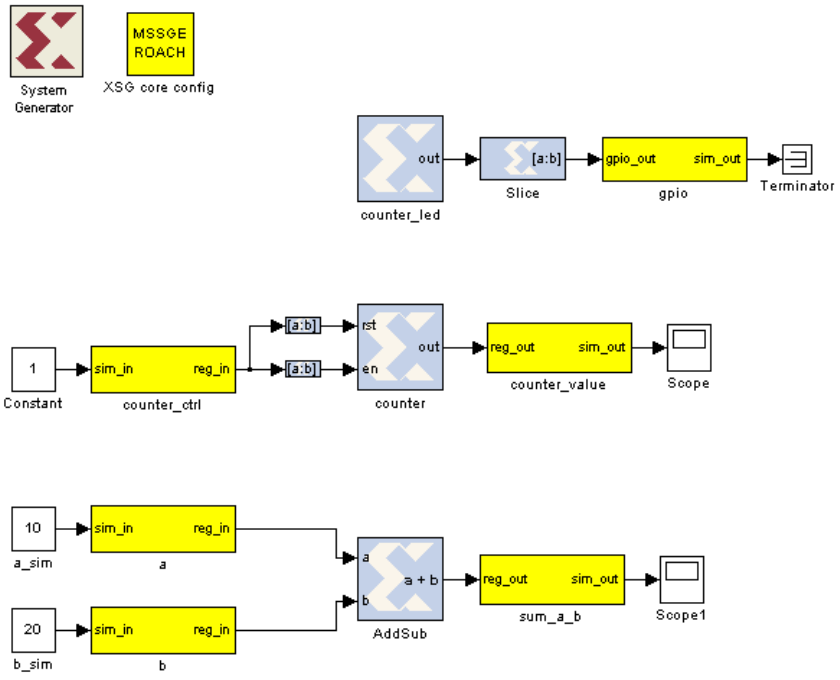
Author: Mark Wagner and Jason Manley

Expected completion time: 2hrs

SKA SOUTH AFRICA
SQUARE KILOMETRE ARRAY

Modified from Mark Wagner's original ROACH tutorial at casper.berkeley.edu/wiki/Roach_Tutorial.

Contents:



1 Introduction

In this tutorial, you will create a simple Simulink design using both standard Xilinx system generator blockset, as well as library blocks specific to ROACH. At the end of this tutorial, you will have a BORPH executable file (a BOF file) and you will know how to interact with your running hardware design using BORPH.

2 Setup

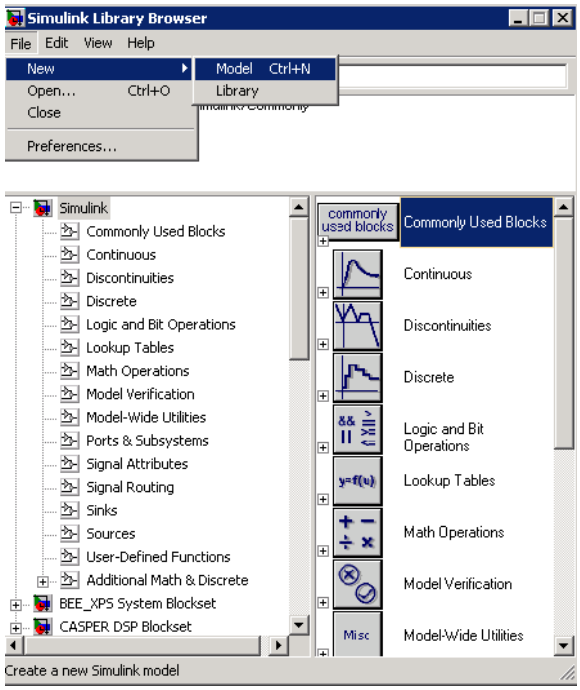
The lab at the workshop is preconfigured with the CASPER libraries, Matlab and Xilinx tools.

3 Creating Your Design

3.1 Create a New Model:

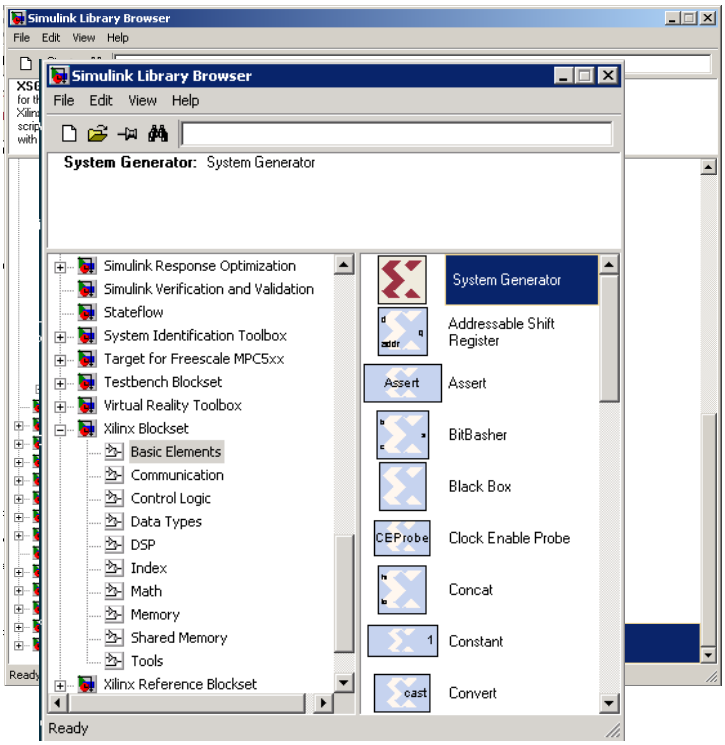
Start Matlab and open Simulink (either by typing *simulink* on the Matlab command line, or by click in the Simulink icon in the taskbar).

Create a new model:



3.2 Add Xilinx System Generator and XSG core config blocks:

Add a System generator block from the Xilinx library by locating the *Xilinx Blockset* library's *Basic Elements* subsection and dragging a *System Generator* token onto your new file. Do not configure it directly, but rather add an *XSG core config* from the *BEE XPS System Blockset* library to do this for you:



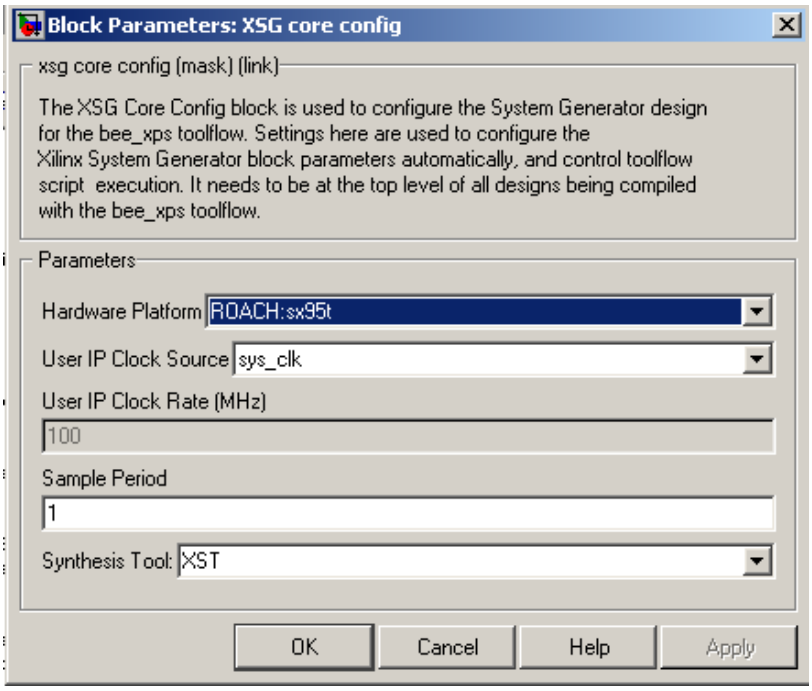
All hardware-related blocks are yellow and can be found in the *BEE_XPS* library. This library contains all the board-specific components colloquially called *Yellow Blocks.*

DSP related blocks are found in the CASPER DSP library and have other colours. Tutorial 3 will introduce you to these blocks.

Double click on the *XSG core config* block that you just added. Set it for *ROACH: SX95t* with *sys_clk* as the clock source. *sys_clk* is an onboard 100MHz crystal.

Leave everything else defaults and click *OK*.

Clocking options include:
•*sys_clk:* This is an onboard 100MHz crystal which is connected to the FPGA.
•*sys_clk2x*: This is the same sys_clk souce (100MHz onboard crystal), PLL'd up to 200MHz using a Digital Clock Manager (DCM) in the FPGA.
•*arb_clk*: Arbitrary clock using 100MHz onboard crystal with DCM on FPGA to produce any frequency (rounded to nearest available integer n/m in accordance with DCM abilities).
•*aux_clk* (*usr_clk* on older platforms): SMA input to board. PLL'd versions of these clocks are also available.
•*adcX_clk*: For use in conjuction with ADC boards, clock the FPGA off the ADC. For iADC and KATADC, this is 1/4 of sampling rate (ADCs demux internally). You need to use one of these clocks if you are using an ADC.



This will go off and configure the System Generator block which you previously added.

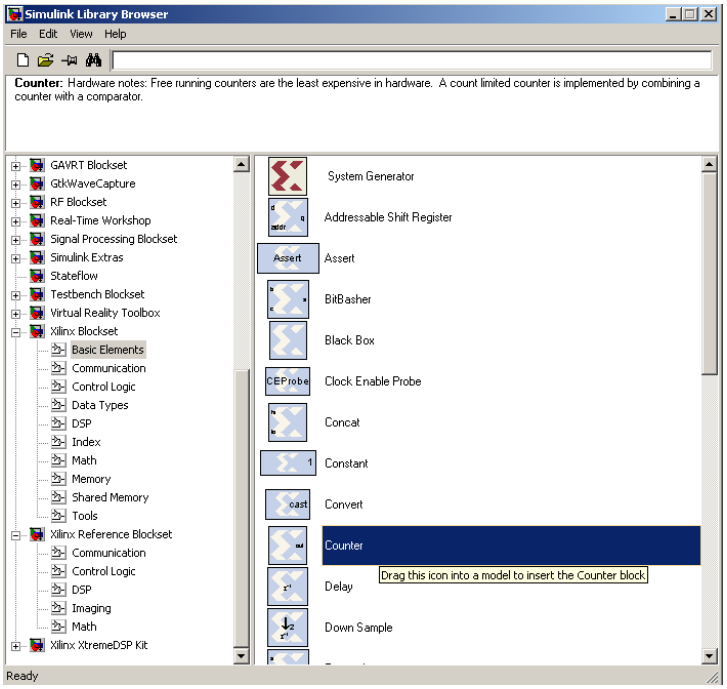
You need to add these two blocks for all CASPER designs.

3.3 Flashing LED

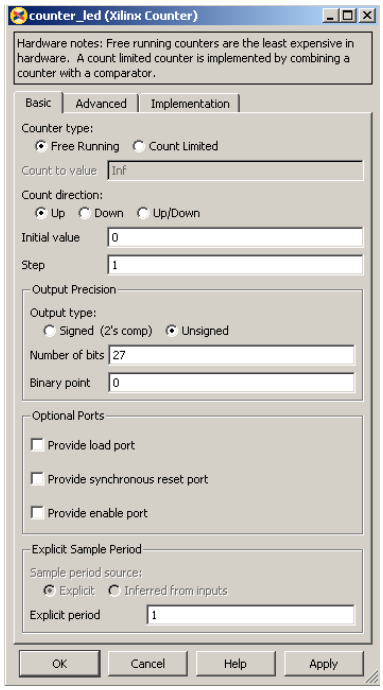
To demonstrate the basic use of hardware interfaces, we will make an LED flash. With the FPGA running at 100MHz, the most significant bit (msb) of a 27 bit counter will toggle every 0.745 seconds. We can output this bit to an LED on ROACH. ROACH has four green LEDs. We will now connect a counter to the first one.

3.3.1 Add a counter

Add a counter to your design by navigating to *Xilinx Blockset -> Basic Elements -> Counter* and dragging it onto your model.

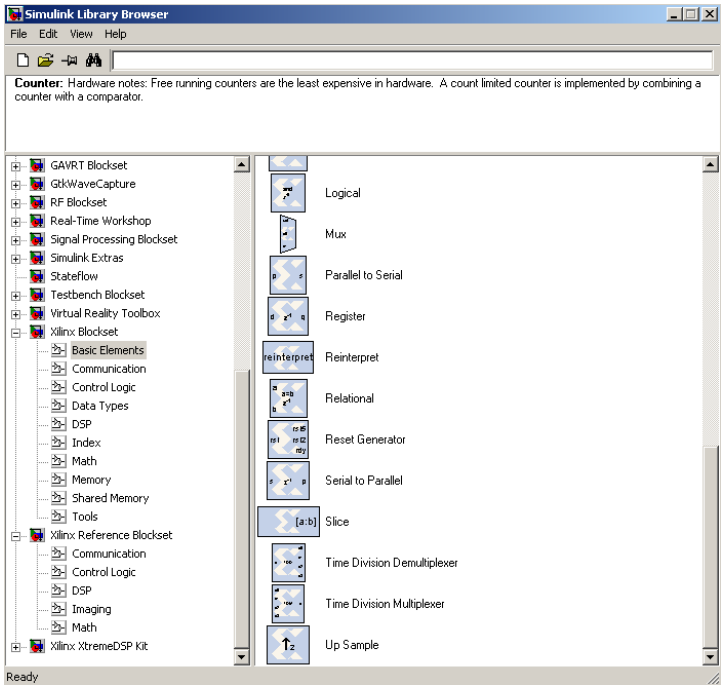


Double-click it and set it for free running, 27 bits, unsigned.



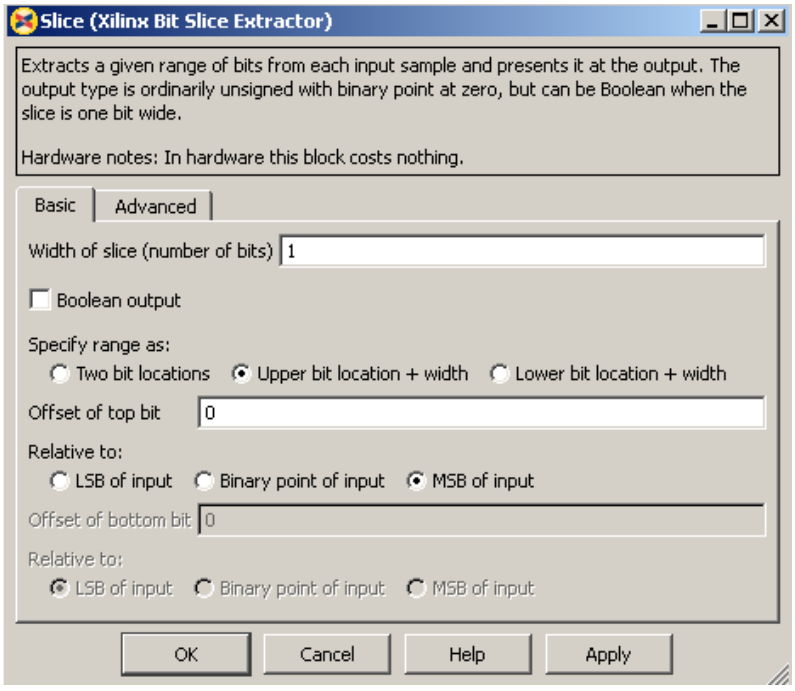
3.3.2 Add a slice block to select out the msb

We now need to select the most significant bit of the counter. We do this using a slice block, which Xilinx provides. *Xilinx Blockset -> Basic Elements -> Slice*.



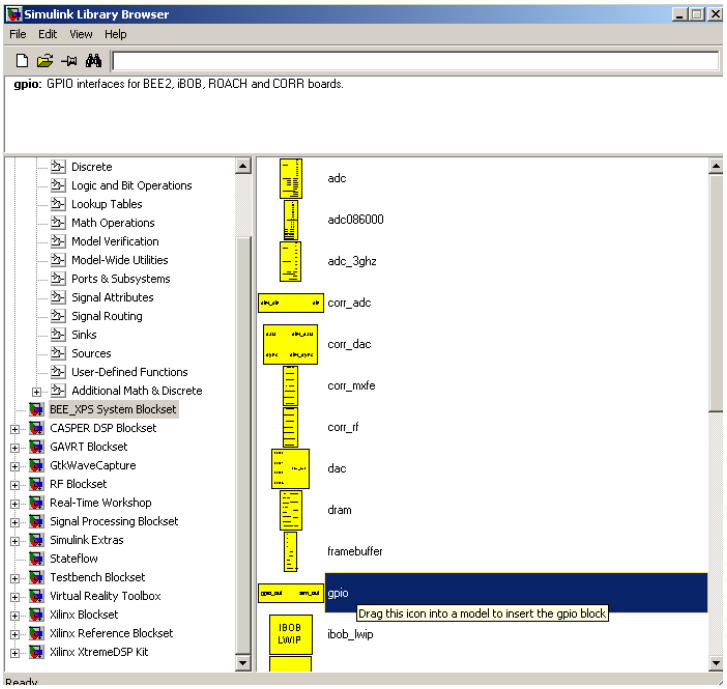
Double-click on the newly added slice block. There are multiple ways to select which bit(s) you want. In this case, I find it simplest to index from the upper end and select the first bit. If you wanted the lsb, you could also index from the lsb,. You can either select the width and offset, or two bit locations.

Set it for 1 bit wide with offset from top bit at zero.

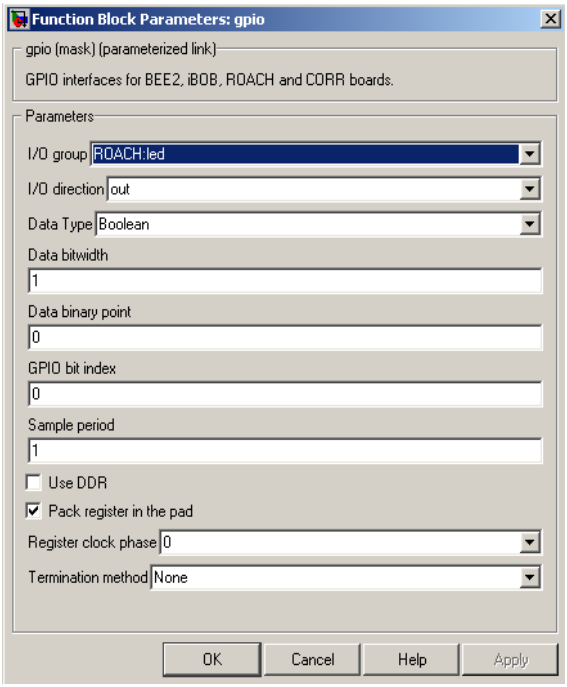


3.3.3 Add a GPIO block

(BEE_XPS library -> gpio).

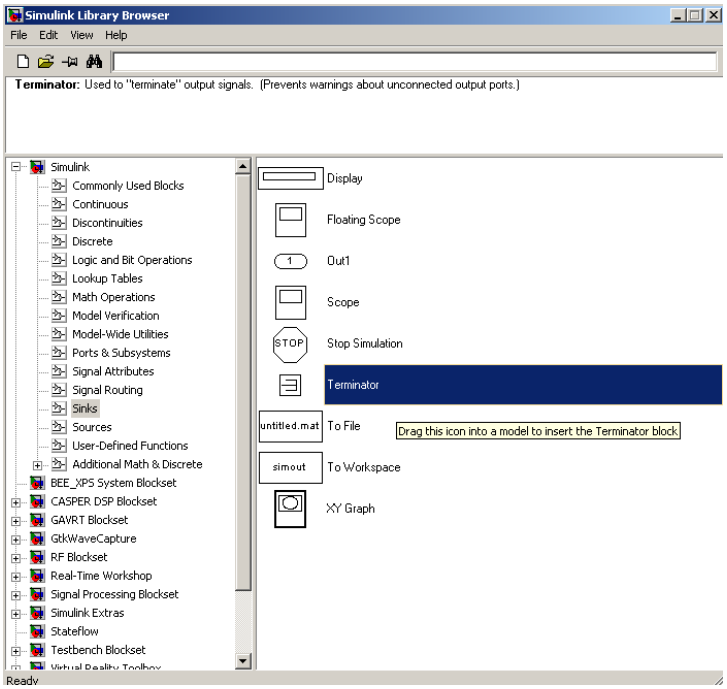


Set it to use ROACH's LED bank as output, GPIO bit index 0 (the first LED).



3.3.4 Add a terminator

To prevent warnings about unconnected outputs, terminate all unused outputs using a Terminator:



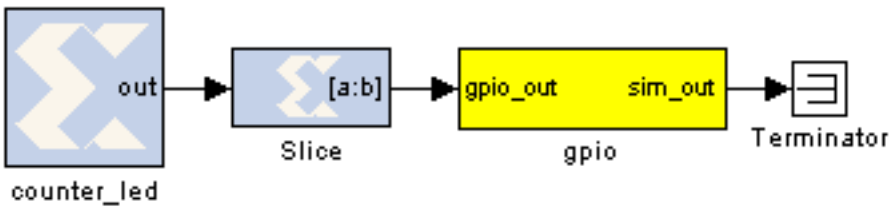
Note that all blocks from the "Simulink" library (usually white), will not be compiled into hardware. They are present for simulation only and expect continuous signals, not discreet.

Only Xilinx blocks (they are blue with Xilinx logo) will be compiled to hardware.

For this reason, you need to use *gateway* blocks whenever connecting a Simulink-provided block (like a scope or constant) for simulations. Some of the CASPER blocks (like the *GPIO* block) do this for you with "sim_in" and "sim_out". We will see later how to use a 'scope to monitor these lines.

3.3.5 Connect your design

It is a good idea to rename your blocks to something more sensible, like *counter_led* instead of just *counter*. Do this simply by double-clicking on the name of the block and editing the text appropriately.



It is a good time to **save** this new design. There are some Matlab limitations you should be aware-of:

Do not use spaces in your filenames, or anywhere in the file path as it will break the toolflow.

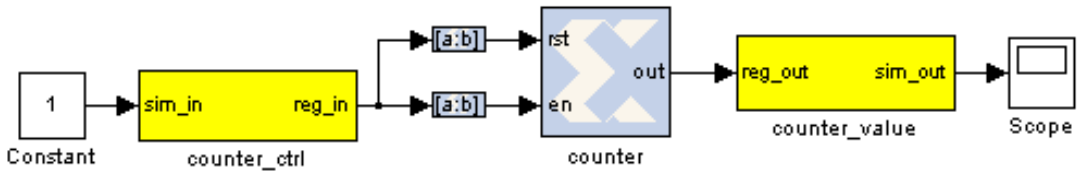
Total path length **cannot be more than 64 characters**. By “path”, I am refering to not only the file path, but also the path to any block within your design. For example, if you save this file to *c:\projects\myfile.mdl*, the longest Matlab-indexed path would be *c:\projects\myfile.mdl\counter_led*. While this is quite short, but there can be additional blocks hidden underneath some of your top level blocks. This is the case with GPIO, for example. This will become clearer later when we demonstrate the use of *SubSystems.* For now, try to keep your names short.

Please save your design in ***c:\projects\<YOUR_INITIALS>_tut1.mdl.***

3.3.6 Software control

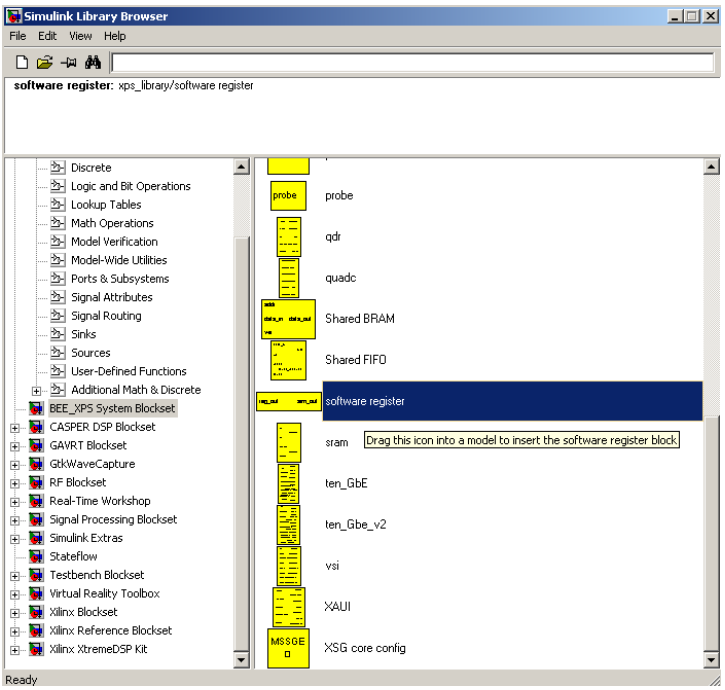
To demonstrate the use of software registers and control of the FPGA through the PPC, we will add a controllable counter to the above design. The counter can be started and stopped from software and also reset. We will be able to monitor the counter's current value too.

By the end of this section, you will create a system that looks like this:

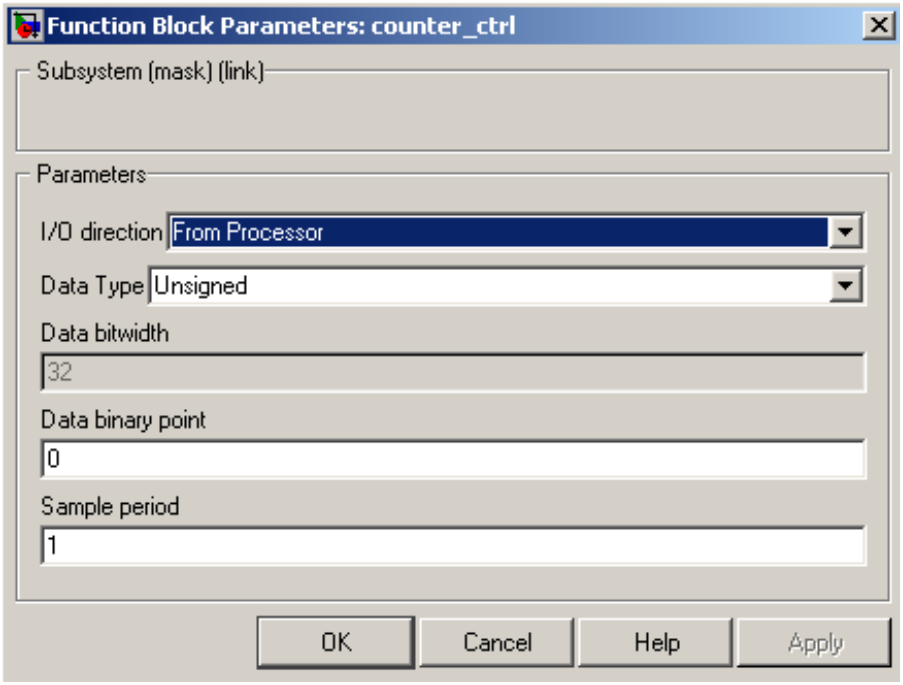


3.3.7 Add the software registers

We need two software registers. One to control the counter, and a second one to read its current value. From the *BEE_XPS System Blockset* library, drag two *Software Registers* onto your design.



Set the *I/O direction* to *From Processor* on the first one to enable dataflow from PowerPC to the FPGA fabric. Set it to *To Processor* on the second one.



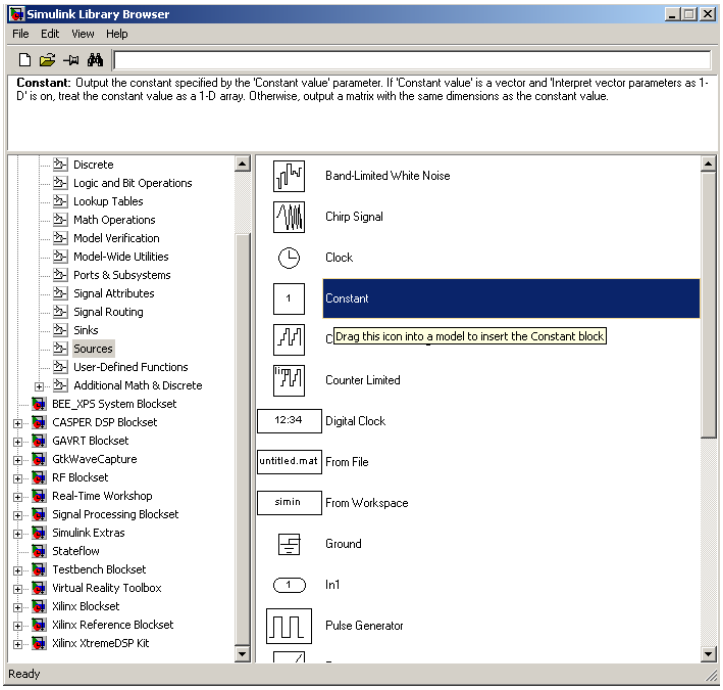
Note the field *Data bitwidth* is greyed out with a value 32. This is because all software registers have a fixed data bitwidth of 32 bits.

Rename the registers to something sensible, as these names are mapped to filenames in the PPC for controlling the design. Avoid using spaces, slashes and other funny characters in these names

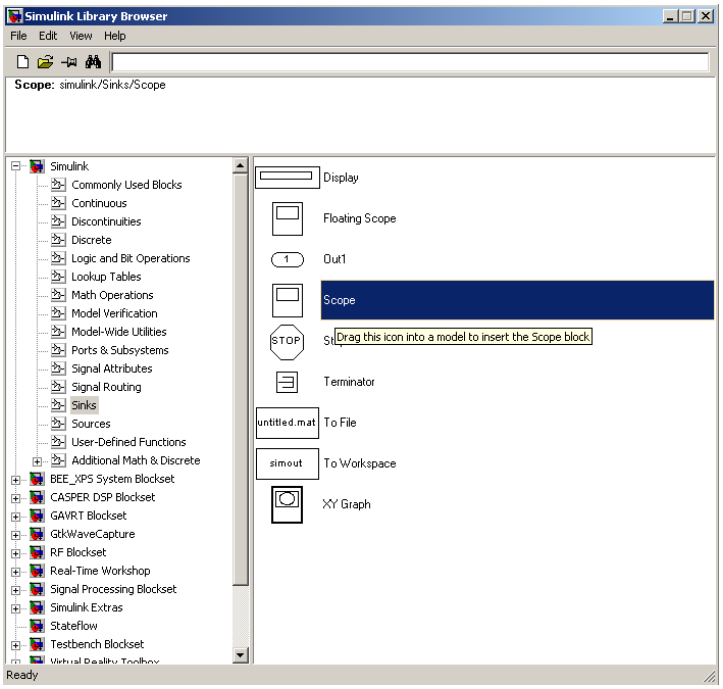
(spaces automatically get remapped to underscores anyway, but should be avoided for clarity). I suggest *counter_ctrl* and *counter_value,* to represent the control and output registers respectively.

Also note that the software registers have *sim_in* and *sim_out* ports. The input port provides a means of simulating this register's value (as would be set by the PPC) using the *sim_in* line. The output port provides a means to simulate this register's current FPGA-assigned value.

For now, set the *sim_in* port to constant one using a Simulink-type constant. This will enable the counter during simulations.



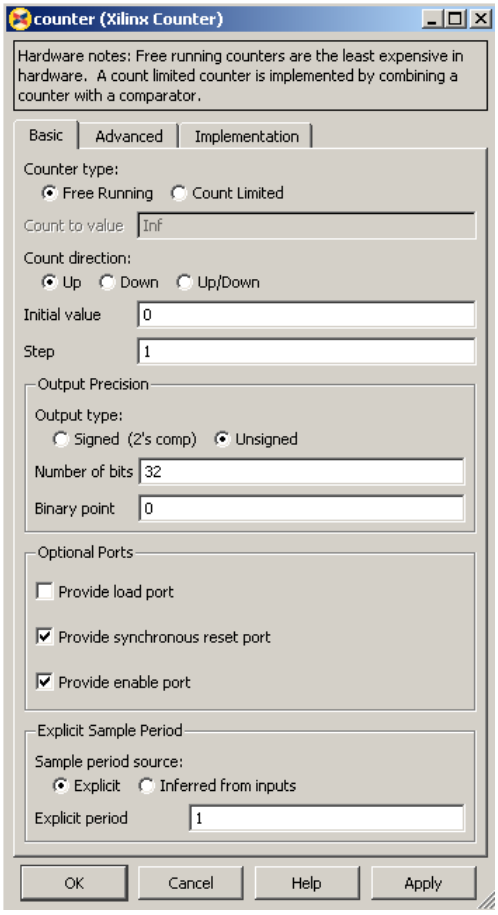
During simulation, we can monitor the counter's value using a scope:



3.3.8 Add the counter

You can do this either by copying your existing counter block (copy-paste, or ctrl-click-drag-drop) or by placing a new one from the library.

Configure it with a reset and enable port as follows:



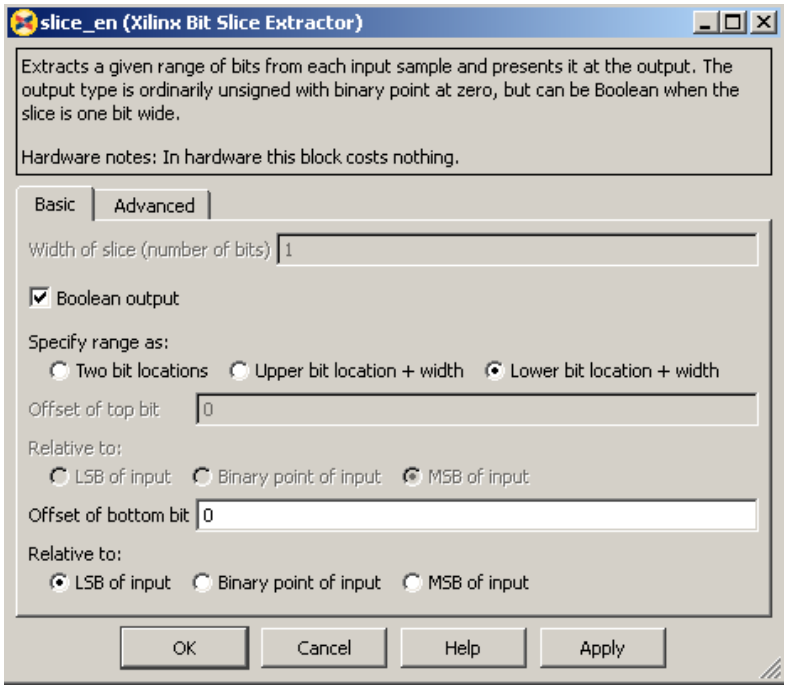
3.3.9 Add the slice blocks

Now we need some way to control the enable and reset ports of the counter. We could do this using two separate software registers, but this is wasteful since each register is 32 bits anyway.

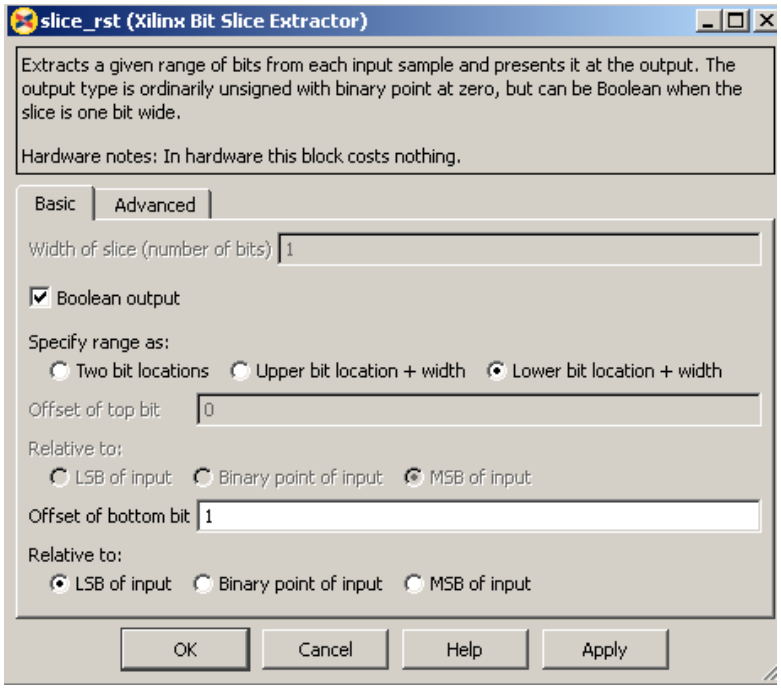
So we'll use a single register and slice out one bit for enabling the counter, and another bit for resetting it. Either copy your existing slice block (copy-paste it or hold ctrl while dragging/dropping it) or add two more from the library.

The enable and reset ports of the counter require boolean values (which Simulink interprets differently from ordinary 1-bit unsigned numbers). Configure the slices as follows:

Slice for enable:



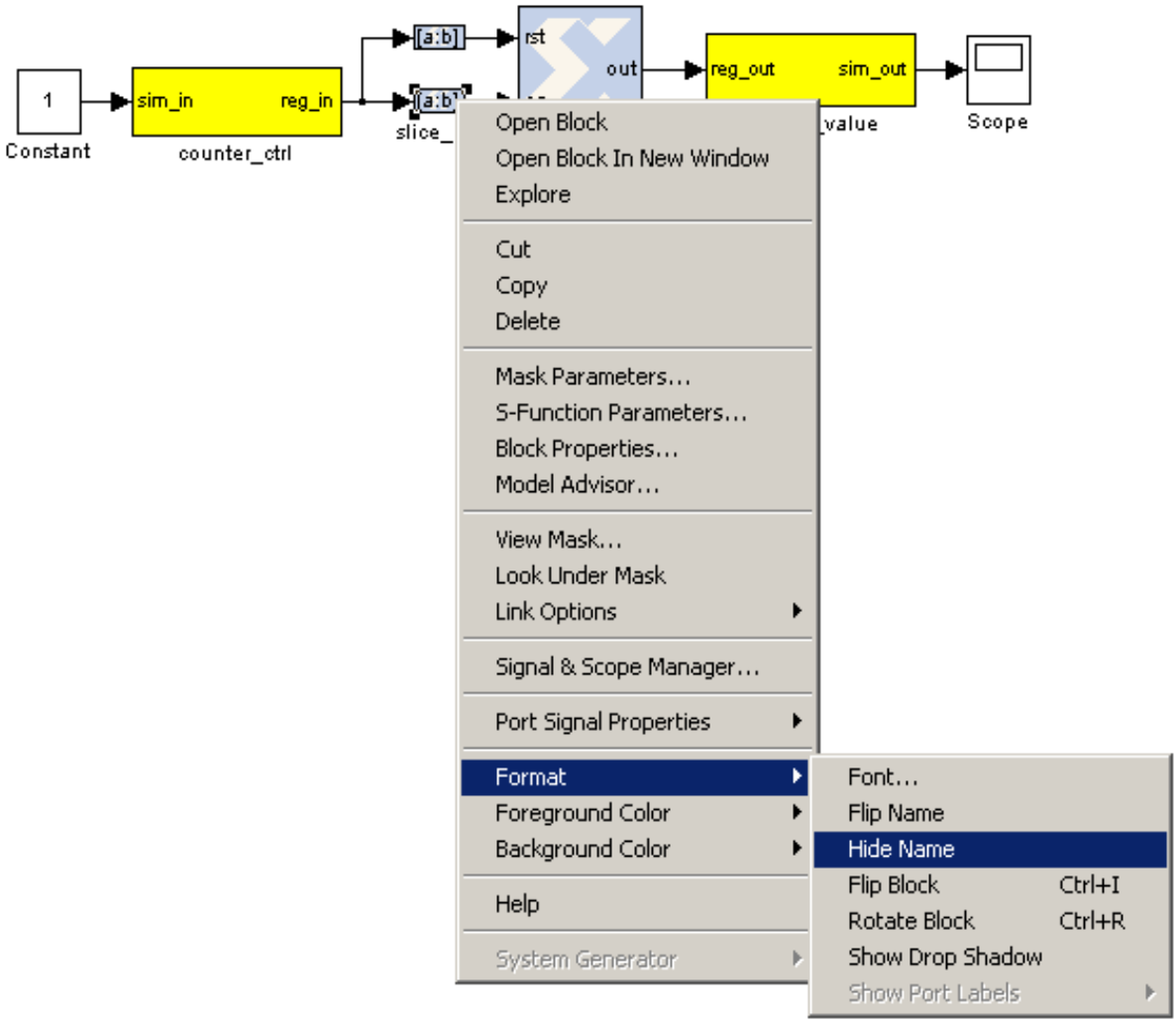
Slice for reset:



3.3.10Connect it all up

Now we need to connect all these blocks together. To neaten things up, consider resizing the slice blocks and hiding their names. Their function is clear enough from their icon without needing to see their names.

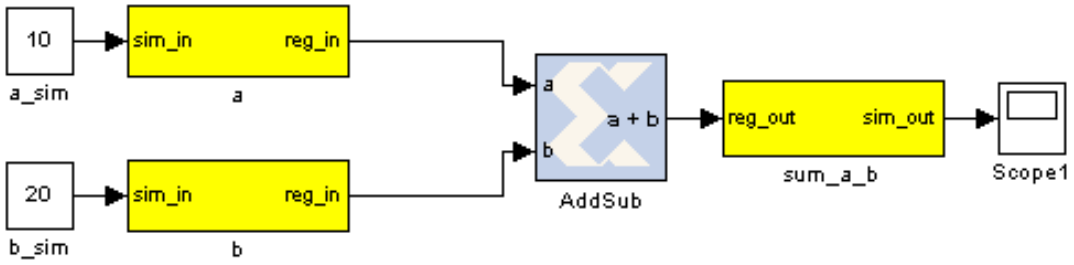
Do so by right-clicking and selecting Format → Hide Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



3.4 Adder

To demonstrate some simple mathematical operations, we will create an adder. It will add two numbers on demand and output the result to another software register. Almost all astronomy DSP is done using fixed-point (integer) notation, and this adder will be no different.

We will calculate *a+b=sum_a_b.*



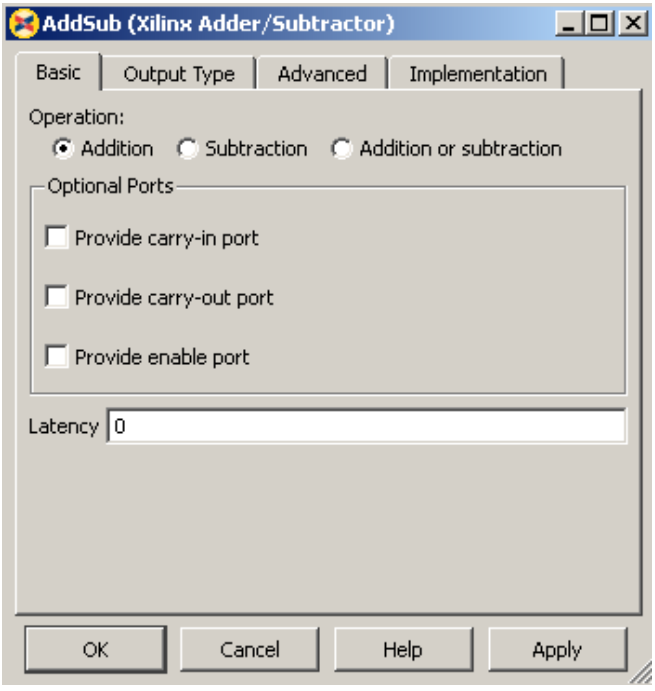
3.4.1 Add the software registers

Add two more input software registers. These will allow us to specify the two numbers to add. Add another output register for the sum output.

Either copy your existing software register blocks (copy-paste or holding ctrl while dragging/dropping it) or add three more from the library. Set the *I/O direction* to *From Processor* on the first two and set it to *To Processor* on the third one.

3.4.2 Add the adder block

Locate the adder/subtractor block, *Xilinx Blockset -> Math -> AddSub* and drag one onto your design. This block can optionally perform addition or subtraction. Let's leave it set at it's default, for addition.



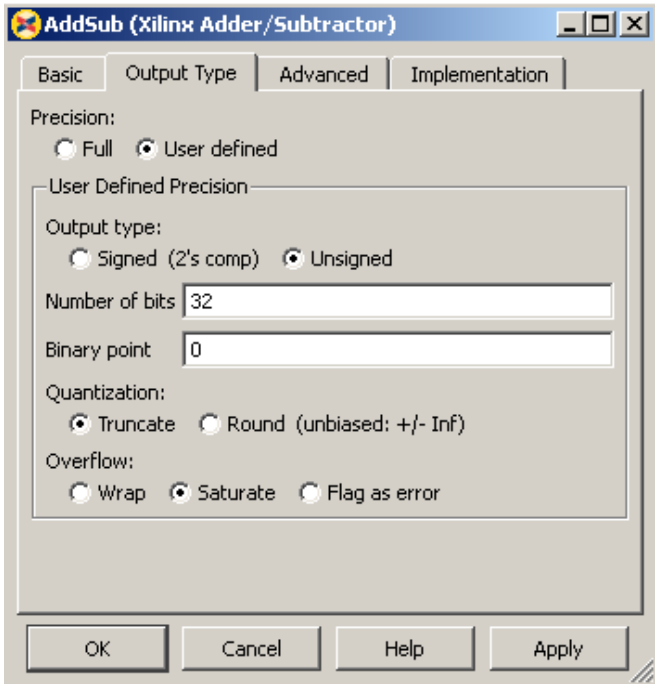
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

There are a number of ways of fixing this:

*) limit the input bitwidth(s) with slice blocks
*) limit the output bitwidth with slice blocks
*) create a 32 bit adder.

Since you have already seen slice blocks demonstrated, let's try to set the AddSub block to be a 32 bit saturating adder. On the second tab, set it for user-defined precision, unsigned 32 bits.

Also, under overflow, set it to saturate. Now if we add two very large numbers, it will simply return 2^32 -1.

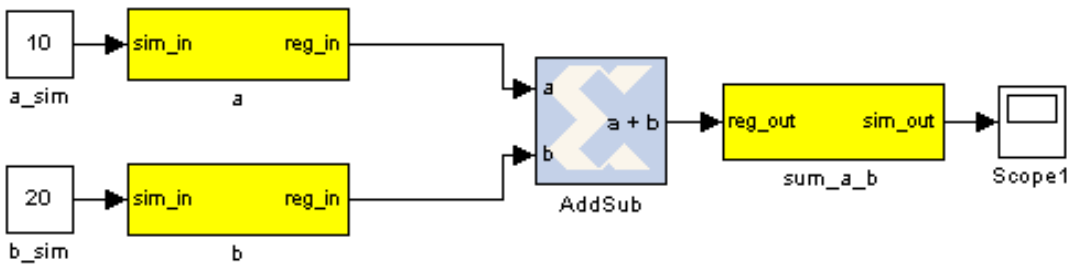


3.4.3 Add the scope and simulation inputs

Either copy your existing scope and simulation constants (copy-paste or ctrl-drag) or place a new one from the library as before. Set the values of the simulation inputs to anything you like.

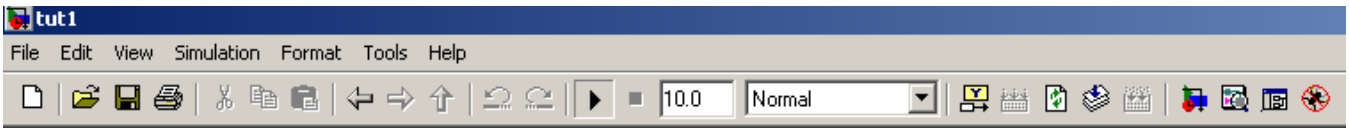
3.4.4 Connect it all together

Like this:



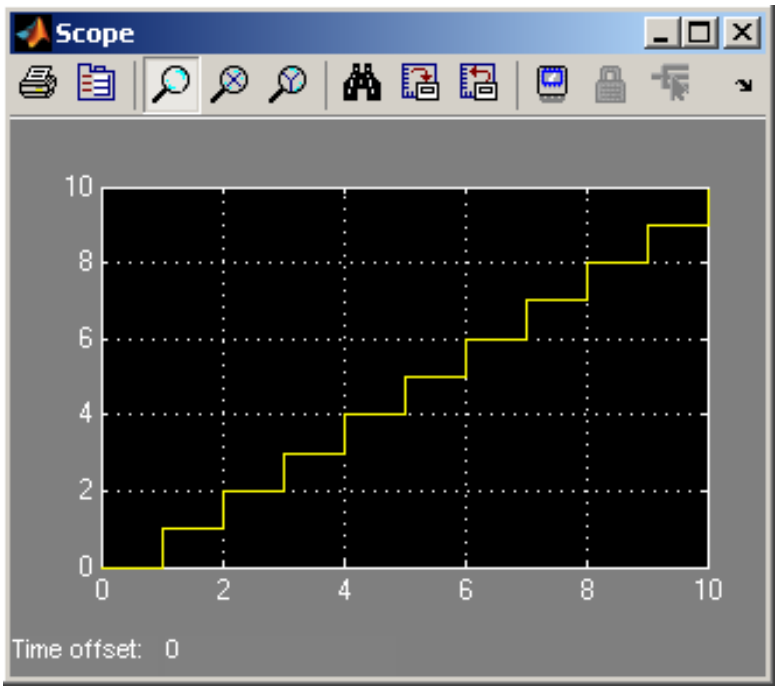
4 Simulating

The design can be simulated with clock-for-clock accuracy directly from within Simulink. Set the number of clock cycles that you'd like to simulate and press the play button in the top toolbar.

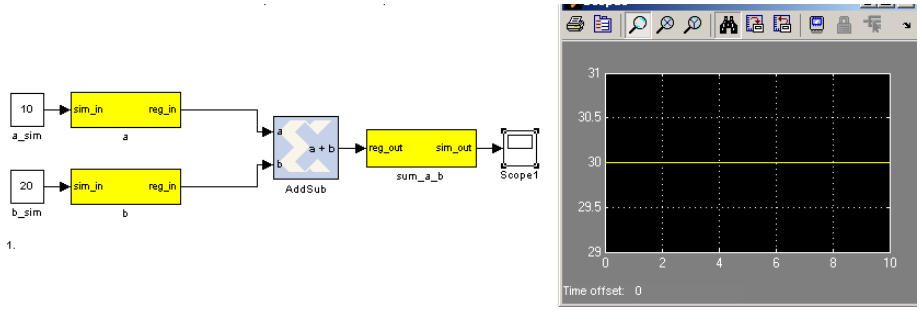


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

You can double-click on the scopes to see what the signals look like on those lines. For example, the one connected to the counter should look like this:



The one connected to your adder should return a constant, equal to the sum of the two numbers you entered. You might have to press the *Autoscale* button to scale the scope appropriately.

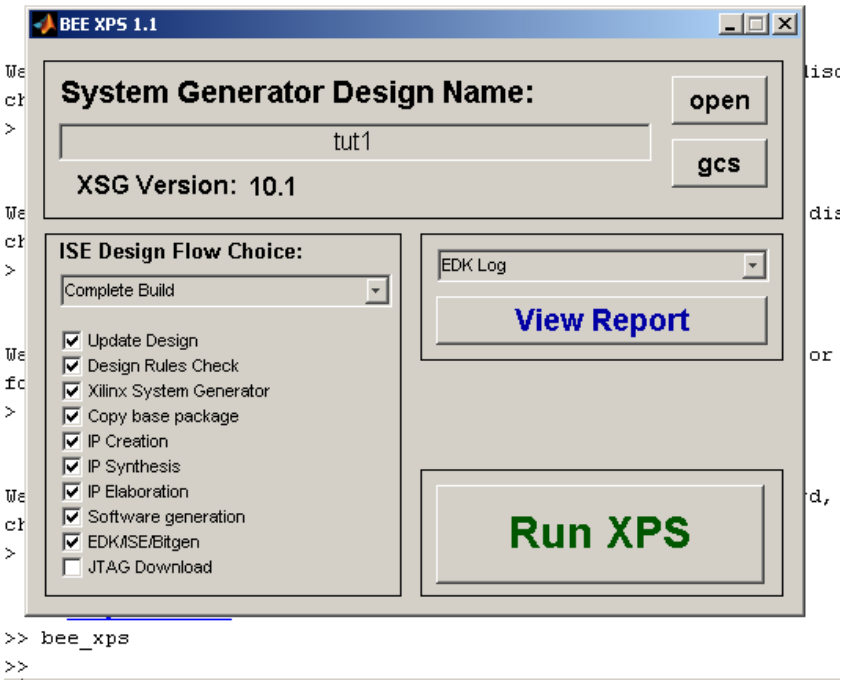


Once you have verified that that design functions as you'd like, you're ready to compile for the FPGA...

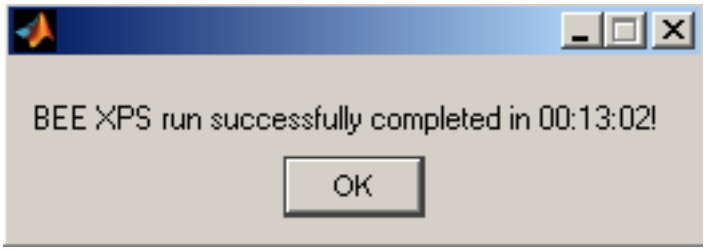
5 Compiling

Essentially, you have constructed three completely separate little instruments. You have a flashing LED, a counter which you can start/stop/reset from software and also an adder. These components are all clocked off the same 100MHz system clock crystal, but they will operate independently.

In order to compile this to an FPGA bitstream, type *bee_xps* on the Matlab command line. Leave all options on defaults. Ensure that the listed design is the one you want to compile (*System Generator Design Name).* If it is not, click anywhere on your design such that it is the highlighted window, then click *gcs.* To start the process, simply click *RUN XPS*.



Compile time is approximately 15 minutes on the little blue computers. When complete, you should receive a popup box like this:



6 Transferring you design to the FPGA

To transfer your design to the FPGA see the General Roach Instructions document.

7 Programming the FPGA

To program the FPGA see the General Roach Instructions Document.