
CASPER Tutorials Documentation

Release 0.1

Collaboration for Astronomy Signal Processing and Electronics R

Mar 10, 2020

SNAP Tutorials

1	Tutorial Instructions	3
2	Environment setup	291

Welcome to the CASPER tutorials page! Here you will find all the current tutorials for the ROACH, SNAP, SKARAB and Red Pitaya platforms.

It is recommended to start with the introduction tutorial for the platform of your liking, then do that platform's GBE tutorial and finally move onto the spectrometer or correlator tutorial or the next difficulty tutorial.

Currently there are five hardware platforms supported through the CASPER Community:

1. ROACH
2. ROACH2
3. SKARAB
4. SNAP
5. Red Pitaya

It is worth noting that even though SNAP, SKARAB and Red Pitaya require their firmwares to be developed using Xilinx's Vivado (as opposed to ISE), the **SNAP** tutorials are **very** similar to the ROACH/2 tutorials. In fact, the only real difference is the choice of hardware platform that is made in Simulink. This is done by selecting the **SNAP** Yellow Block in the Simulink library under *CASPER XPS Blockset -> Hardware Platforms*

Tutorial Instructions

If you are new to astronomy signal processing, here is [Tutorial 0: some basic introduction into astronomy signal processing](#). If you already have a lot of experience on it, you can go directly to the introduction tutorials below for CASPER FPGA design and implementation.

If you are a beginner, we recommend the Step-by-Step tutorials, however if you should get stuck, prefer a less tedious method of learning, or already have decent feel for these tools, links to Completed tutorials are available with commented models.

1.1 Vivado

SNAP

1. Introduction Tutorial: [Step-by-Step](#) or [Completed](#)
2. 10GbE Tutorial: [Step-by-Step](#) or [Completed](#)
3. Spectrometer Tutorial [Step-by-Step](#) or [Completed](#)
4. Correlator Tutorial [Step-by-Step](#) or [Completed](#)
5. Yellow Block Tutorial: [Bidirectional GPIO](#)

1.1.1 Tutorial 1: Introduction to Simulink

In this tutorial, you will create a simple Simulink design using both standard Xilinx system generator blockset, as well as library blocks specific to CASPER boards (so-called “Yellow Blocks”). At the end of this tutorial, you will know how to generate an fpg file, program it to a CASPER FPGA board, and interact with your running hardware design using [casperfpga](#) via a Python Interface.

Creating Your Design

Create a New Model

Start Matlab via executing the startsg command, as described [here](#). This ensures that necessary Xilinx and CASPER libraries are loaded into your by Simulink. When MATLAB starts up, open Simulink by typing simulink on the MATLAB command line. Start a new model, and save it with an appropriate name. **With Simulink, it is very wise to save early, and often.**

There are some Matlab limitations you should be aware-of right from the start:

- **Do not use spaces in your filenames** or anywhere in the file path as it will break the toolflow.
- **Do not use capital letters in your filenames** or anywhere in the file path as it will break the toolflow.
- **Beware block paths that exceed 64 characters.** This refers to not only the file path, but also the path to any block within your design.
 - For example, if you save a model file with a name ~/some_really_long_filename.slx, and have a block called in a submodule the longest block path would be: some_really_long_filename_submodule_block.
 - If you use lots of subsystems, this can cause problems.

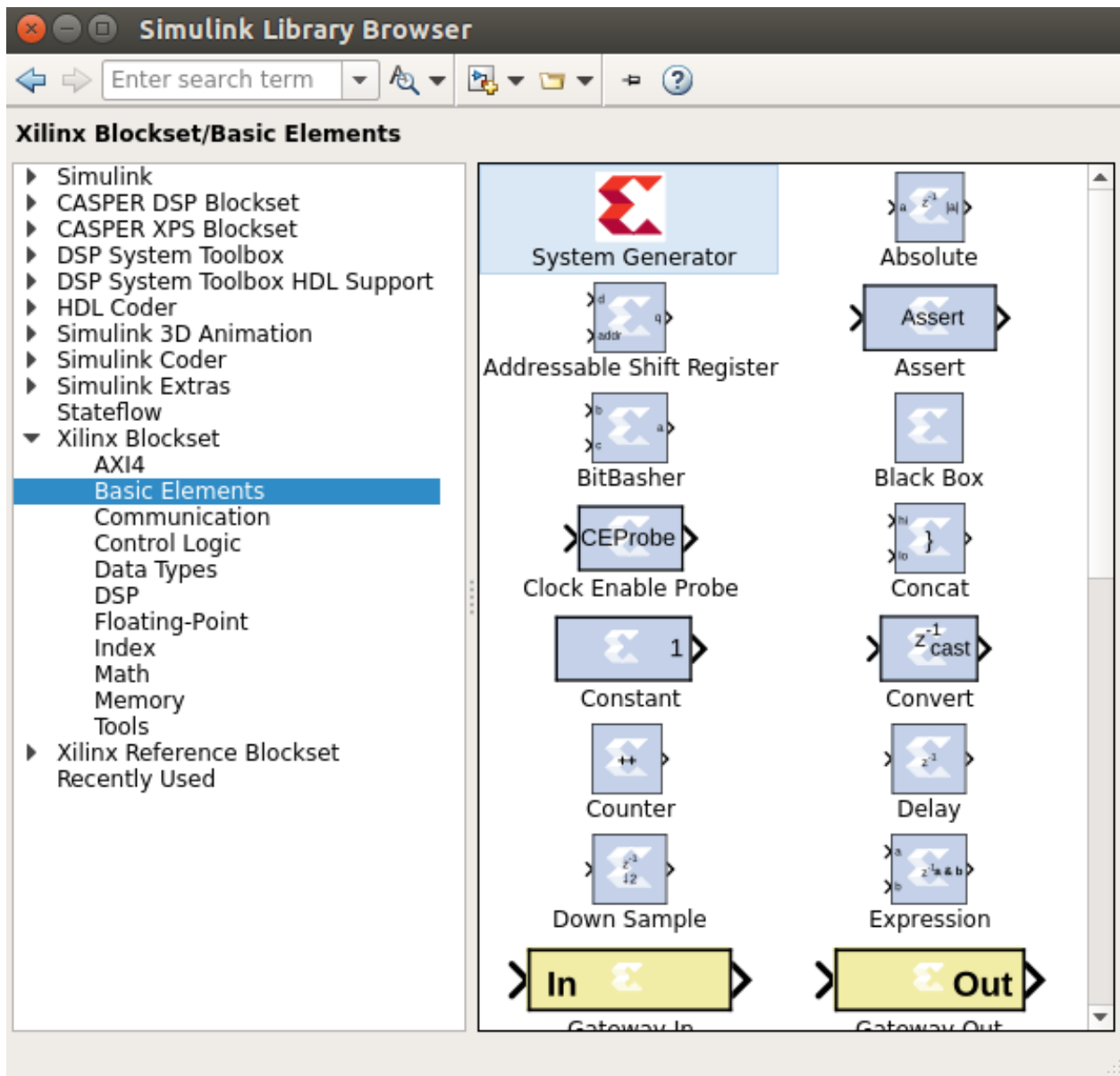
Library organization

There are three libraries which you will use when you design firmware in Simulink.

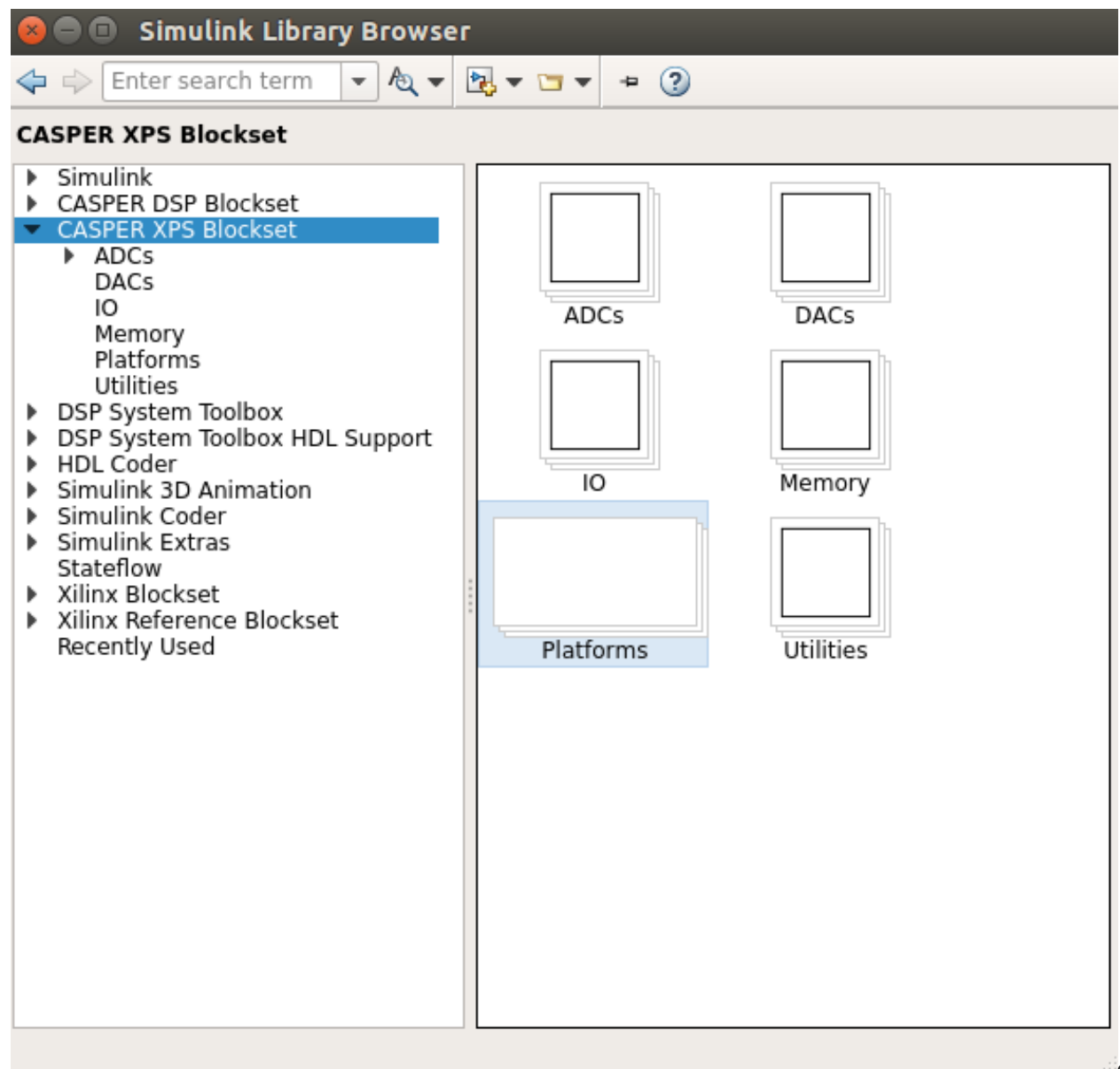
1. The **CASPER XPS Library** contains “Yellow Blocks” – these are blocks which encapsulate interfaces to hardware (ADCs, Memory chips, CPUs, Ethernet ports, etc.)
2. The **CASPER DSP Library** contains (mostly green) blocks which implement DSP functions such as filters, FFTs, etc.
3. The **Xilinx Library** contains blue blocks which provide low-level functionality such as multiplexing, delay-ing, adding, etc. The Xilinx library also contains the super-special System Generator block, which contains information about the type of FPGA you are targeting.

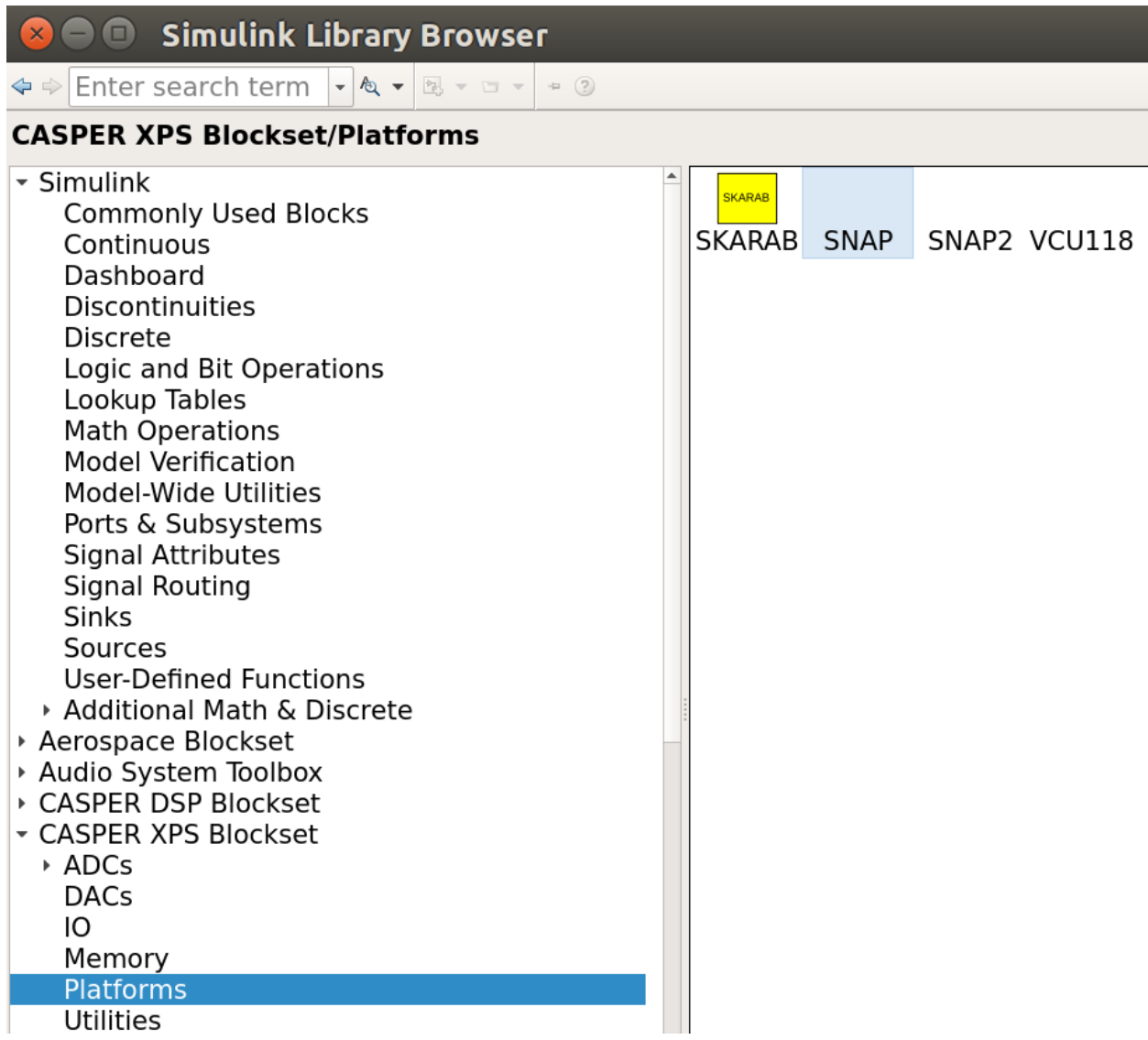
Add Xilinx System Generator and XSG core config blocks

Add a System generator block from the Xilinx library by locating the Xilinx Blockset library’s Basic Elements sub-section and dragging a System Generator token onto your new file.



Do not configure it directly, but rather add a platform block representing the system you are compiling for. These can be found in the CASPER XPS System Blockset library. For SNAP (and later) platforms, you need a block which matches the platform name, which can be found in the library under “platforms”, as shown below.





casper_xps_select

Double click on the platform block that you just added. The Hardware Platform parameter should match the platform you are compiling for. Once you have selected a board, you need to choose where it will get its clock. In designs including ADCs you probably want the FPGA clock to be derived from the sampling clock, but for this simple design (which doesn't include an ADC) you should use the platform's on-board clock. To do this, set the User IP Clock Source to `sys_clk`. The `sys_clk` rate is 100 MHz, so you should set this for *User IP Clock Rate* in the block.

The configuration yellow block knows what FPGA corresponds to which platform, and so it will automatically configure the System Generator block which you previously added.

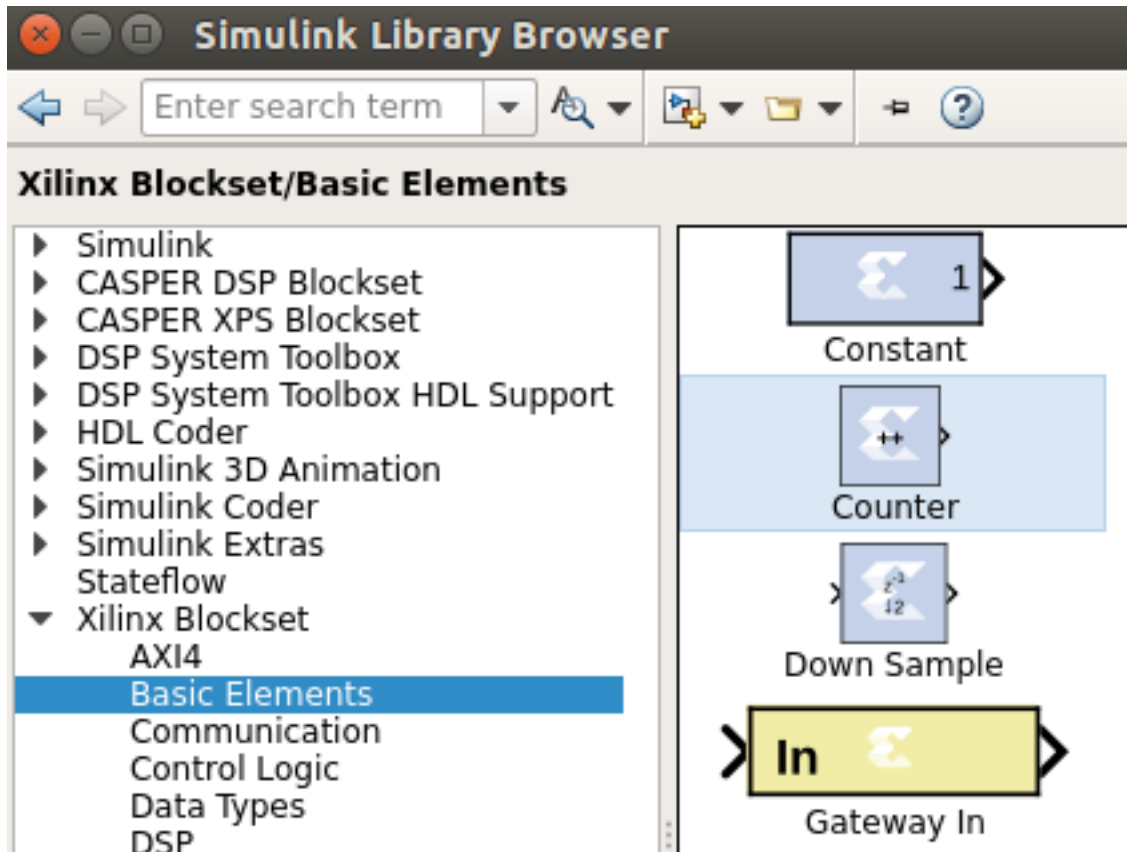
The System Generator and XPS Config blocks are required by all CASPER designs

Flashing LED

To demonstrate the basic use of hardware interfaces, we will make an LED flash. With the FPGA running at ~100MHz (or greater), the most significant bit (msb) of a 27 bit counter will toggle approximately every 0.67 seconds. We can output this bit to an LED on your board. Most (all?) CASPER platforms have at least four LEDs, with the exact configuration depending on the board. We will make a small circuit connecting the top bit of a 27 bit counter to one of these LEDs. When compiled this will make the LED flash with a 50% duty cycle approximately once a second.

Add a counter

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



xilinx_select_counter.png

Double-click it and set it for free running, 27 bits, unsigned. This means it will count from 0 to $2^{27} - 1$, and will then wrap back to zero and continue.

counter_led (Xilinx Counter)

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☐ Provide synchronous reset port

☐ Provide enable port

Explicit Sample Period

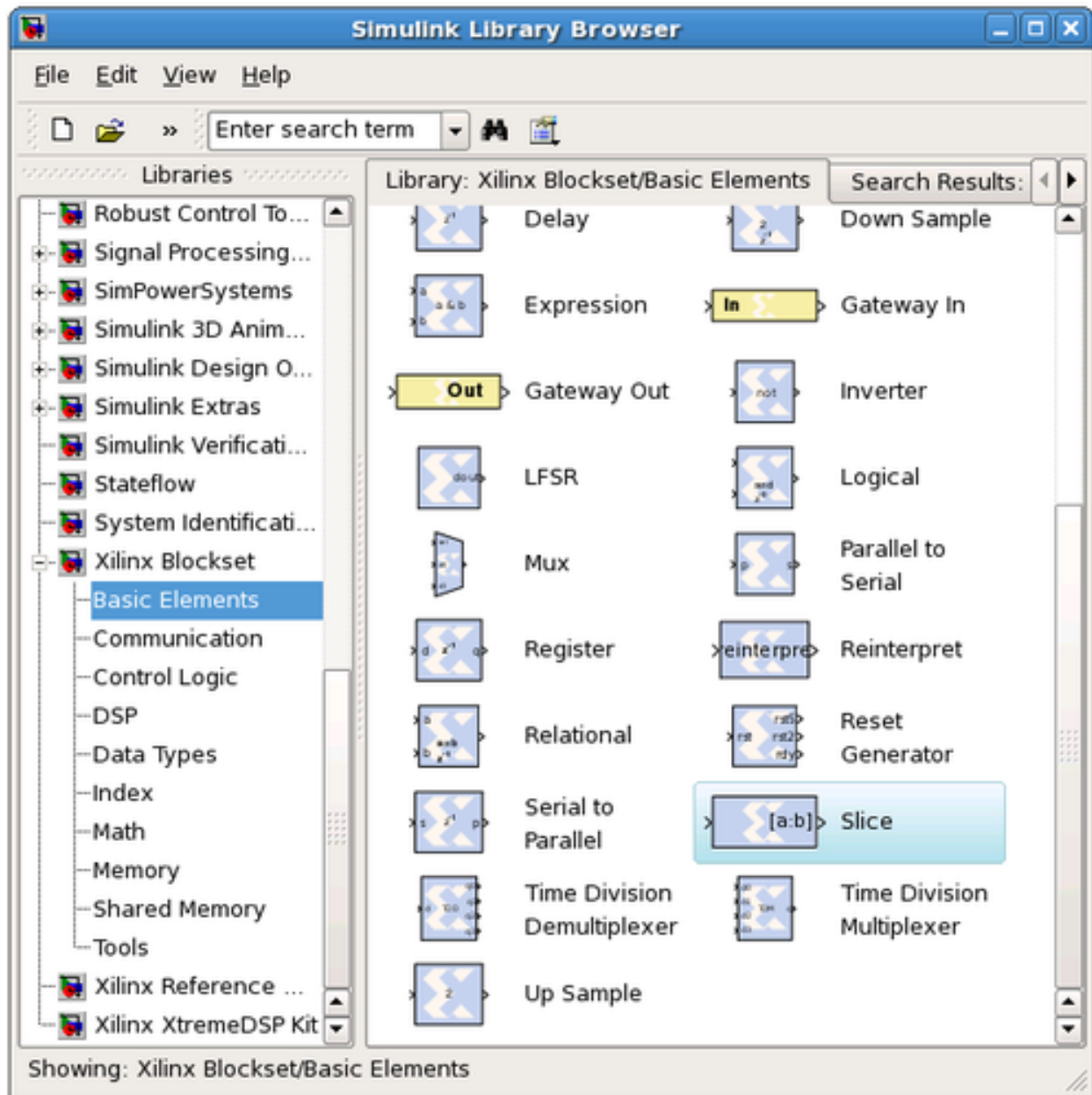
Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

OK **Cancel** **Help** **Apply**

Add a slice block to select out the msb

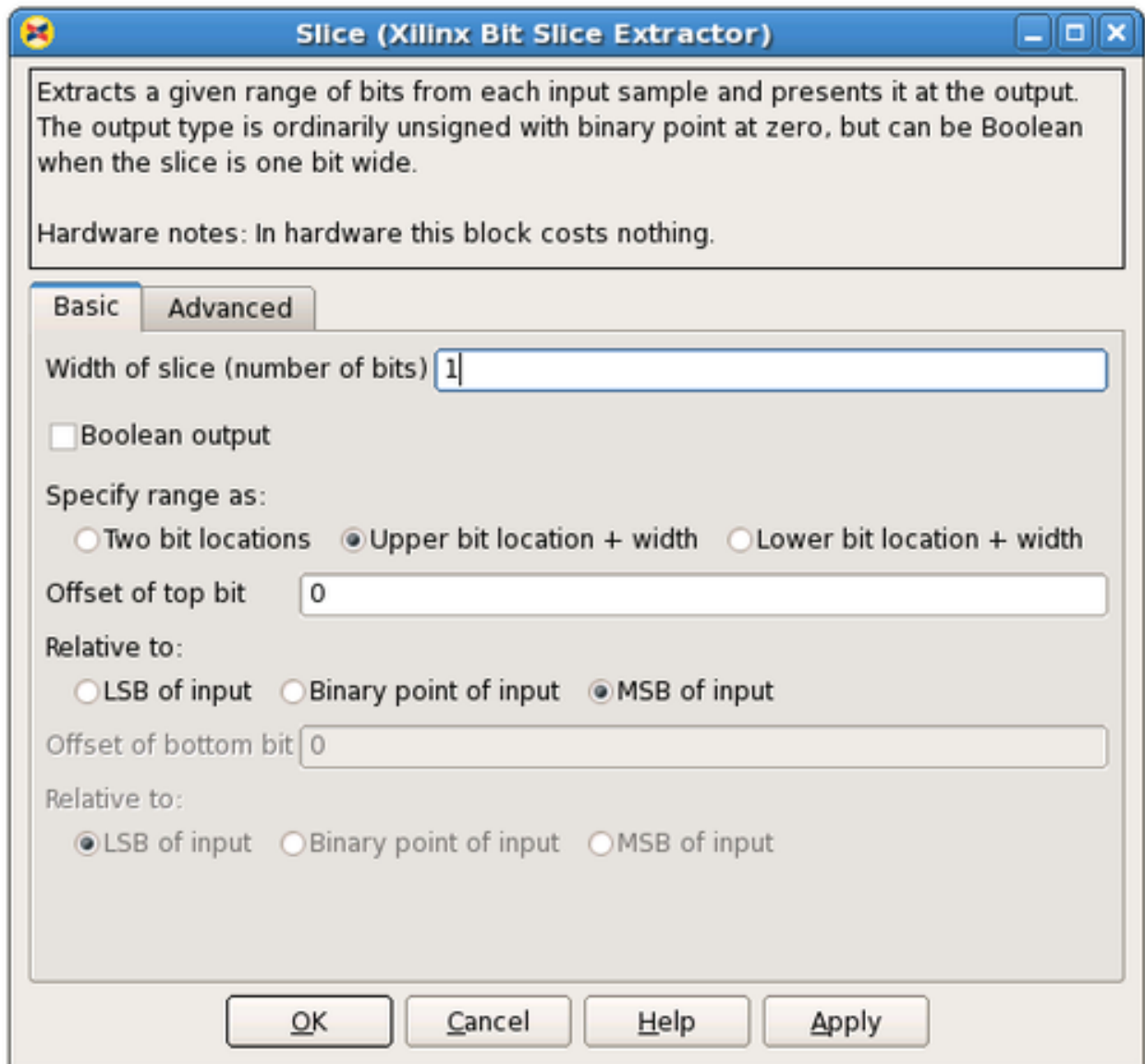
We now need to select the **most significant bit (msb)** of the counter. We do this using a slice block, which Xilinx provides. Xilinx Blockset -> Basic Elements -> Slice.



Slice_select.png

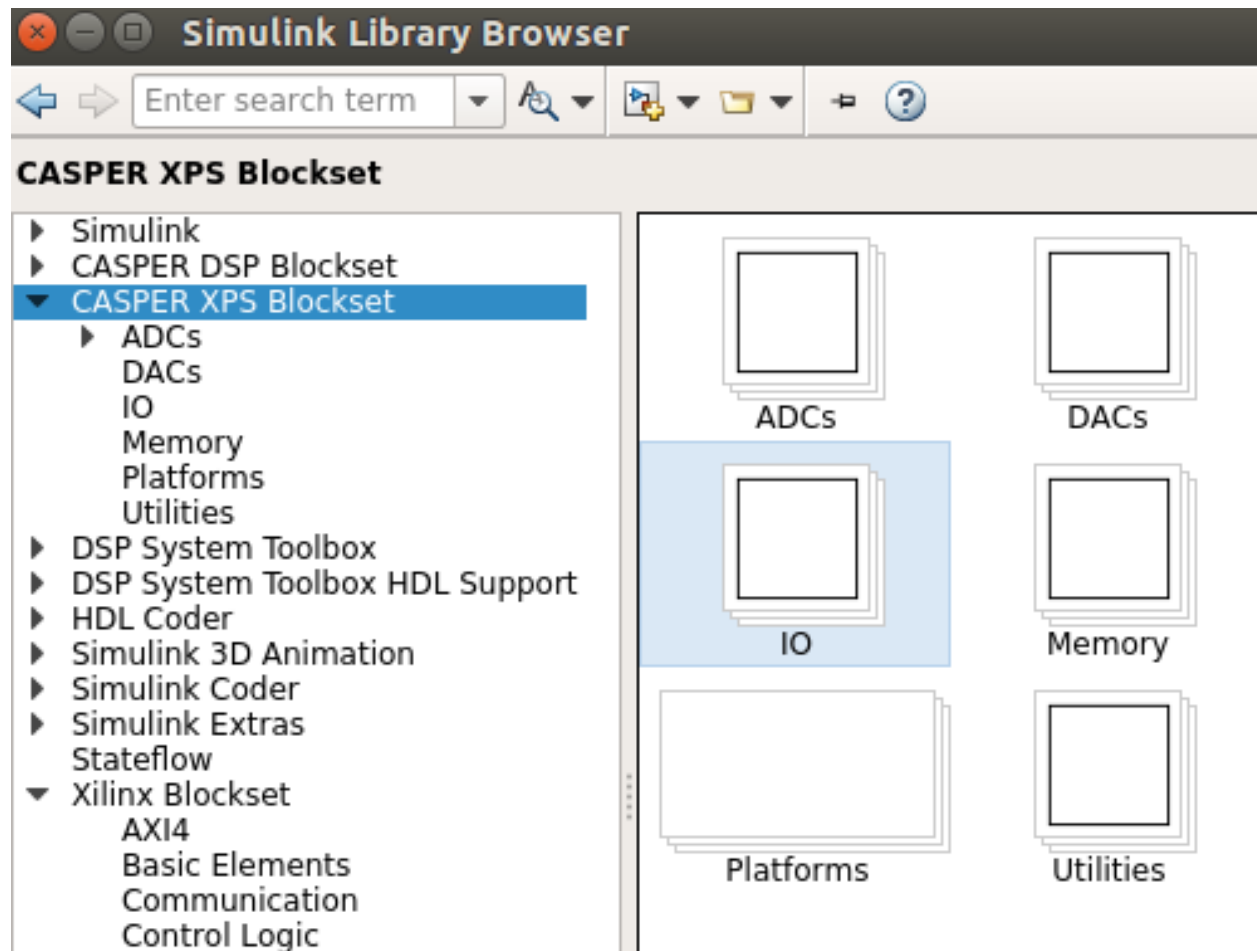
Double-click on the newly added slice block. There are multiple ways to select which bit(s) you want. In this case, it is simplest to index from the upper end and select the first bit. If you wanted the **least significant bit (lsb)**, you can also index from that position. You can either select the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

Set it for 1 bit wide with offset from top bit at zero. As you might guess, this will take the 27-bit input signal, and output just the top bit.

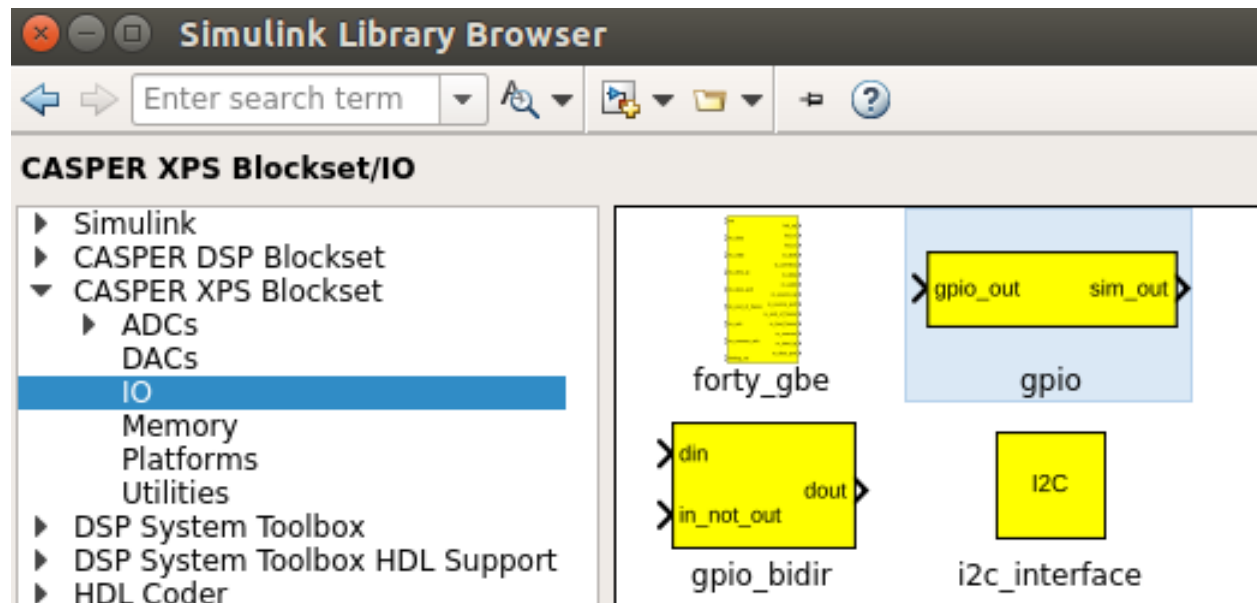


Add a GPIO block

From: CASPER XPS library -> IO -> gpio.



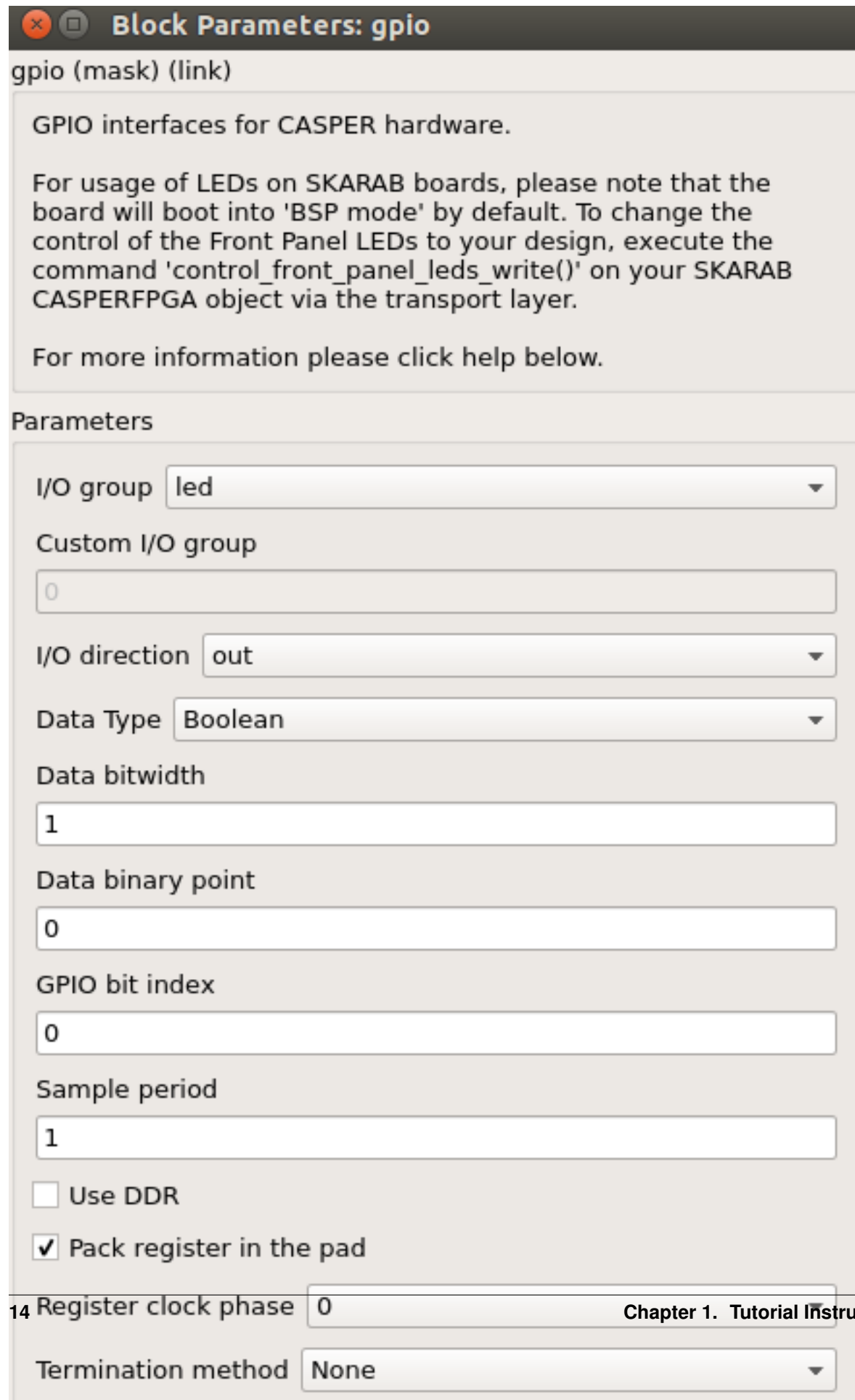
casper_xps_select



casper_xps_select

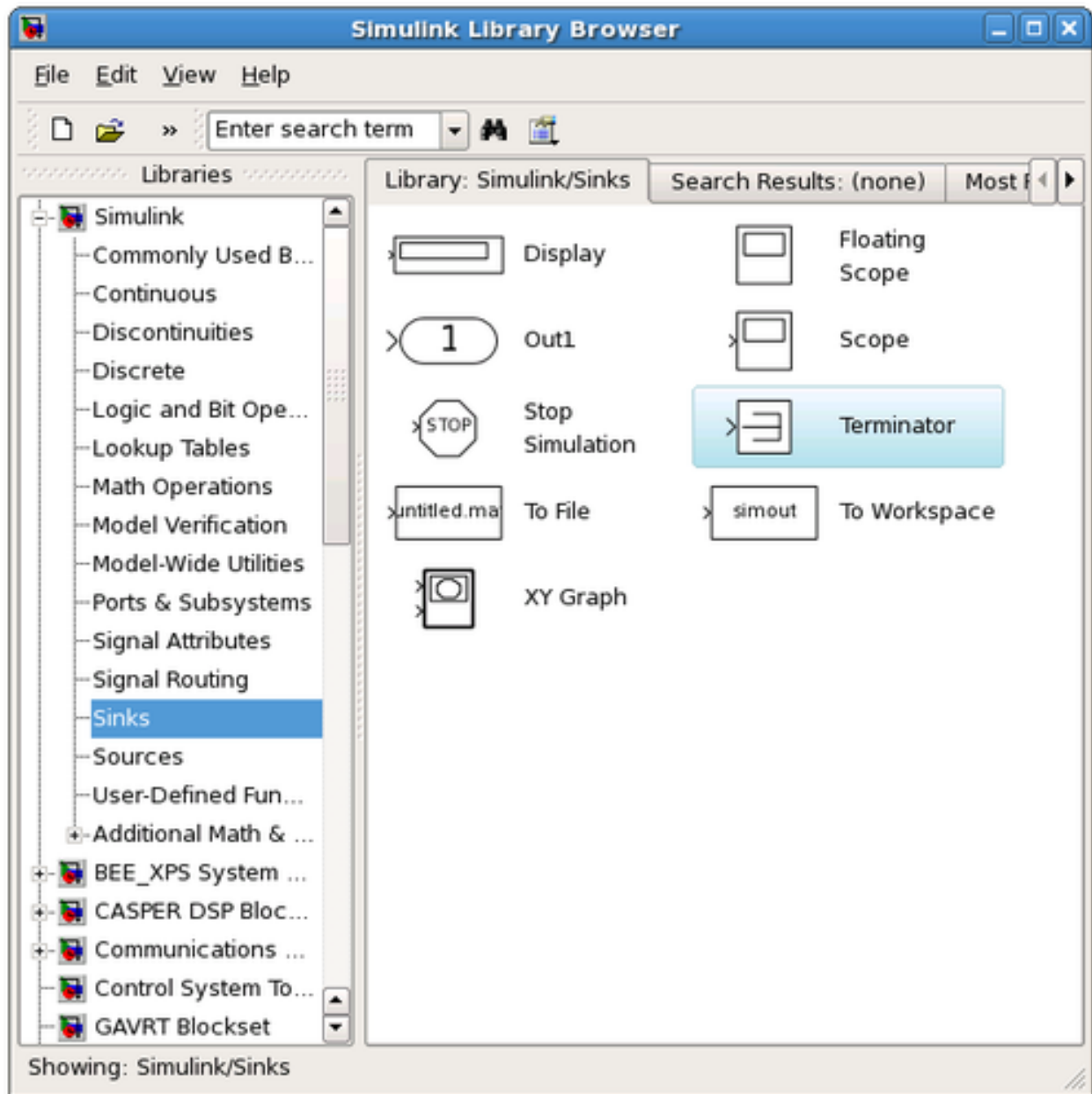
In order to send the 1 bit signal you have sliced off to an LED, you need to connect it to the right FPGA output pin. To do this you can use a GPIO (general-purpose input/output) block from the XPS library, this allows you to route a signal from Simulink to a selection of FPGA pins, which are addressed with user-friendly names. Set it to use SNAP's LED bank as output. Once you've chosen the LED bank, you need to pick *which* LED you want to output to. Set the

GPIO bit index to 0 (the first LED) and the data type to Boolean with bitwidth 1. This means your simulink input is a 1 bit Boolean, and the output is LED0.



Add a terminator

To prevent warnings (from MATLAB & Simulink) about unconnected outputs, terminate all unused outputs using a *Terminator*:



From: Simulink -> Sinks -> Terminator

You can also use the Matlab function `XIAddTerms`, run in the MATLAB prompt, to automatically terminate your unused outputs.

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.

To connect the blocks simply click and drag from the ‘output arrow’ on one block and drag it to the ‘input arrow’ of another block. Connect the blocks together: Counter -> Slice -> gpio as showing in digram below.

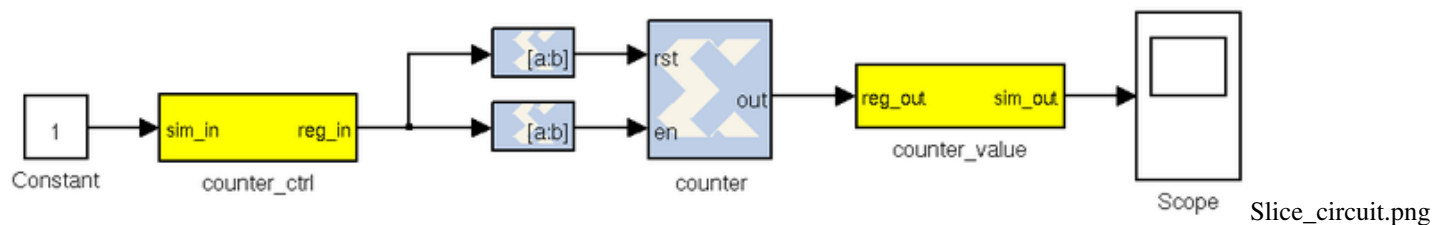


Remember to save your design often.

Software control

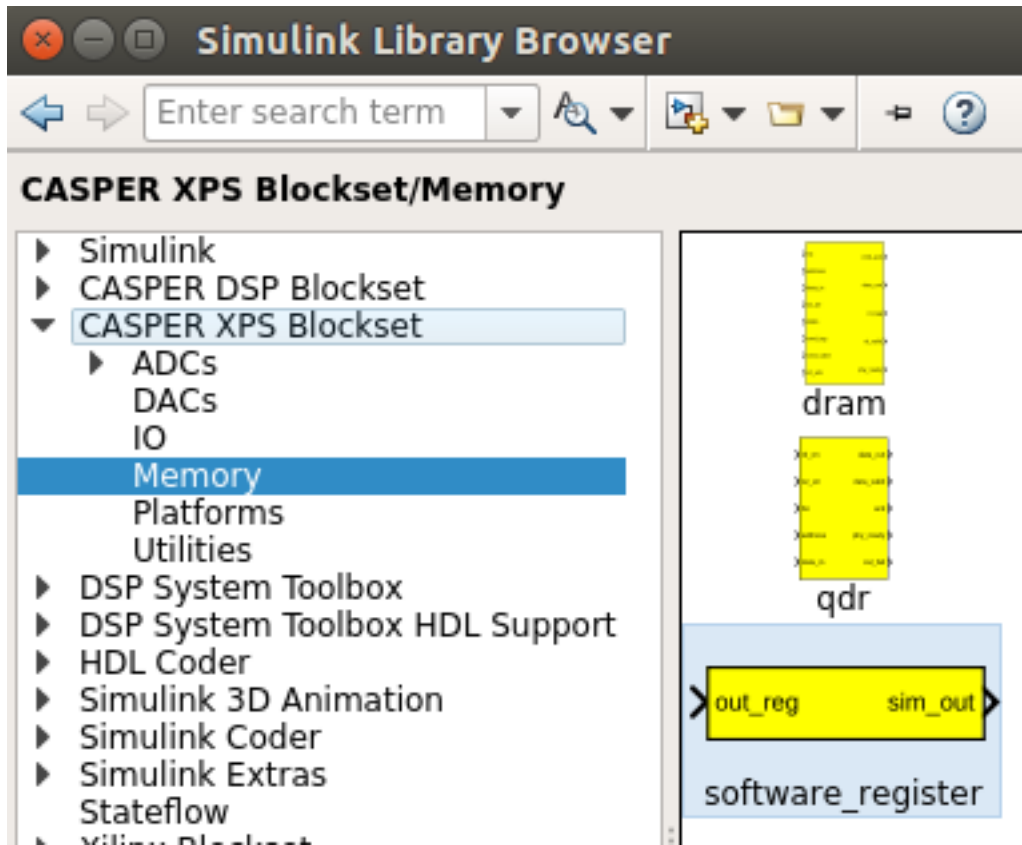
To demonstrate the use of software registers to control the FPGA from a computer, we will add a registers so that the counter in our design can be started, stopped, and reset from software. We will also add a register so that we can monitor the counter’s current value too.

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





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 CASPER XPS System Blockset library, drag two Software Registers onto your design.



casper_xps_select_memory_swreg.png

Set the I/O direction to *From Processor* on the first one (counter control) to enable a value to be set from software and sent *to* your FPGA design. Set it to *To Processor* on the second one (counter value) to enable a value to be sent *from* the FPGA to software. Set both registers to a bitwidth of 32 bits.

 **Block Parameters: counter_ctrl**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction From Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts



0

Bitfield types, ufix=0, fix=1, bool=2

0

☒ Provide sim input/output?

☐ Print format string?

 **Block Parameters: counter_value**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction To Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts

0

Bitfield types, ufix=0, fix=1, bool=2

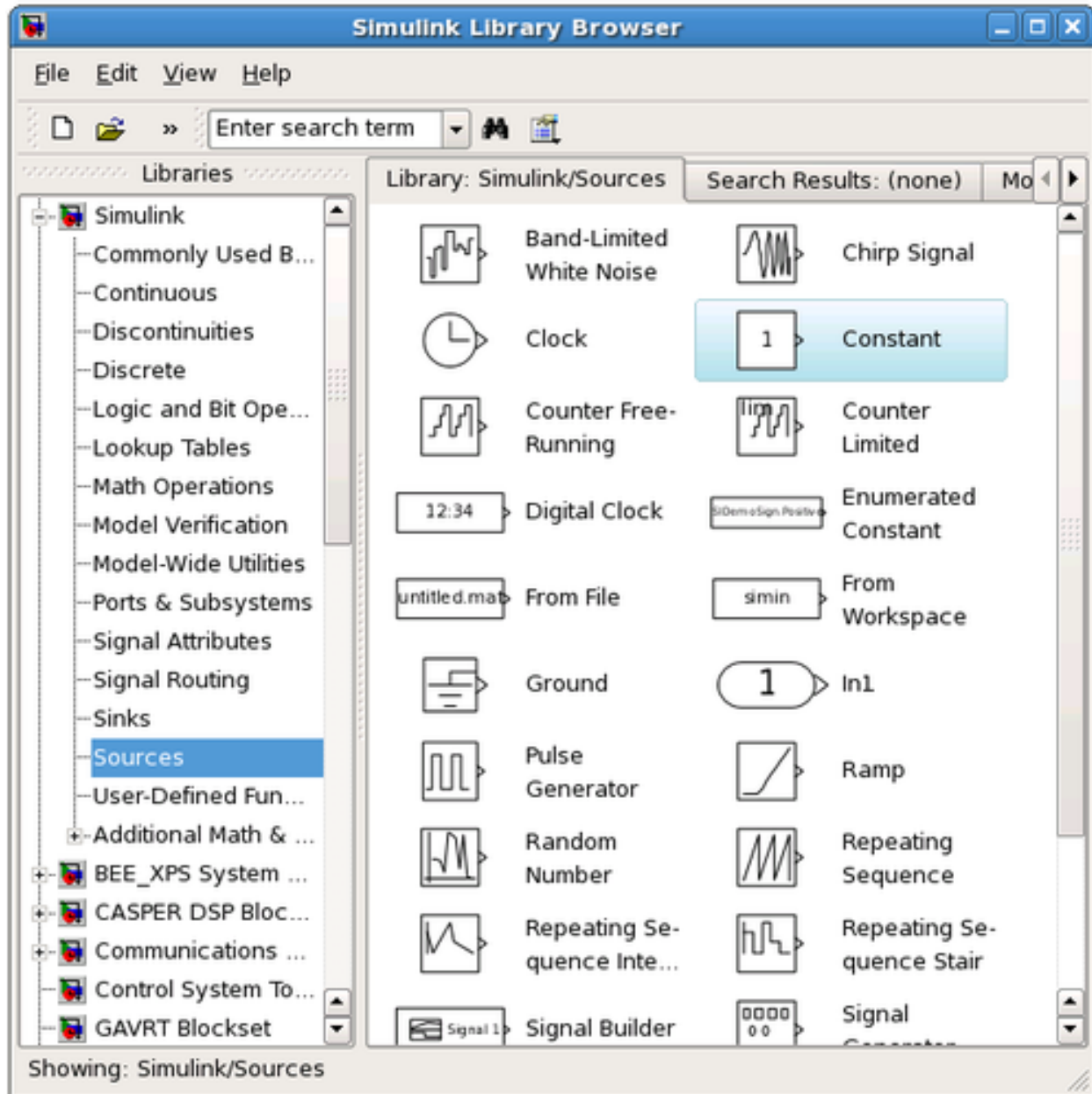
1.1. Vivado

0

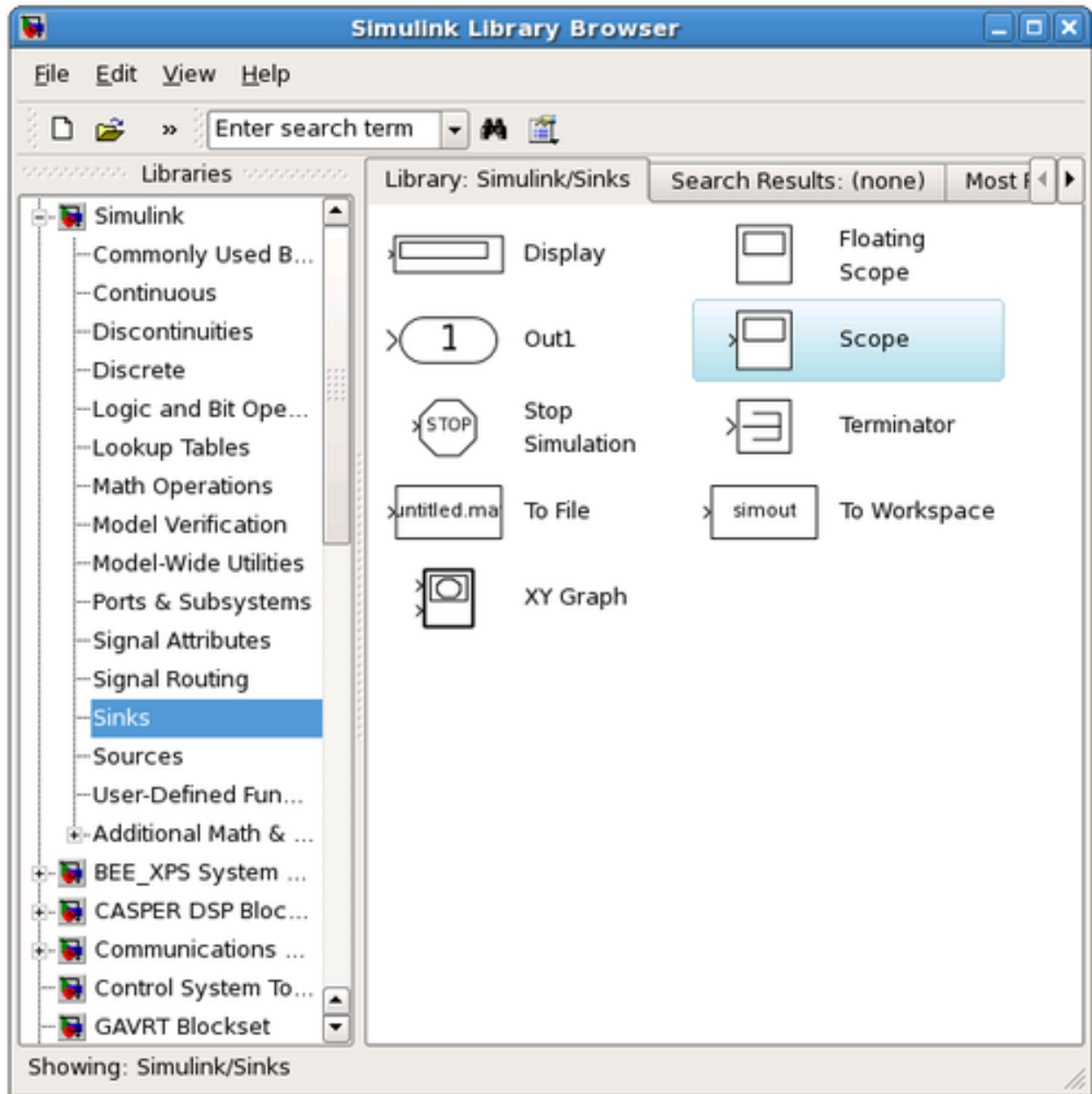
Rename the registers to something sensible. The names you give them here are the names you will use to access them from software. Do not use spaces, slashes and other funny characters in these. Perhaps *counter_ctrl* and *counter_value*, to represent the control and output registers respectively.

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

For now, set the *sim_reg* port to constant one using a Simulink-type constant. Found in *Simulink* -> *Sources*. This will enable the counter during simulations.



During simulation, we can monitor the counter's value using a scope (*Simulink* -> *Sinks*):



Here is a good point to note that all blocks from the *Simulink* library (usually white), will not be compiled into hardware. They are present for simulation only.





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

You need to use *gateway* blocks whenever connecting a Simulink-provided block (like a scope or sine-wave generator) to a from a Xilinx block, this will sample and quantize the simulink signals so that they are compatible with the Xilinx world. Some blocks (like the software register) provide a gateway internally, so you can feed the input of a software register with a xilinx signal, and monitor its output with a Simulink scope. However, in general, you must manually insert these gateways where appropriate. Simulink will issue warnings for any direct connections between the Simulink and Xilinx worlds.

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:

 **counter (Xilinx Counter)**   

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Advanced** **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☒ Provide synchronous reset port

☒ Provide enable port

Explicit Sample Period

Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

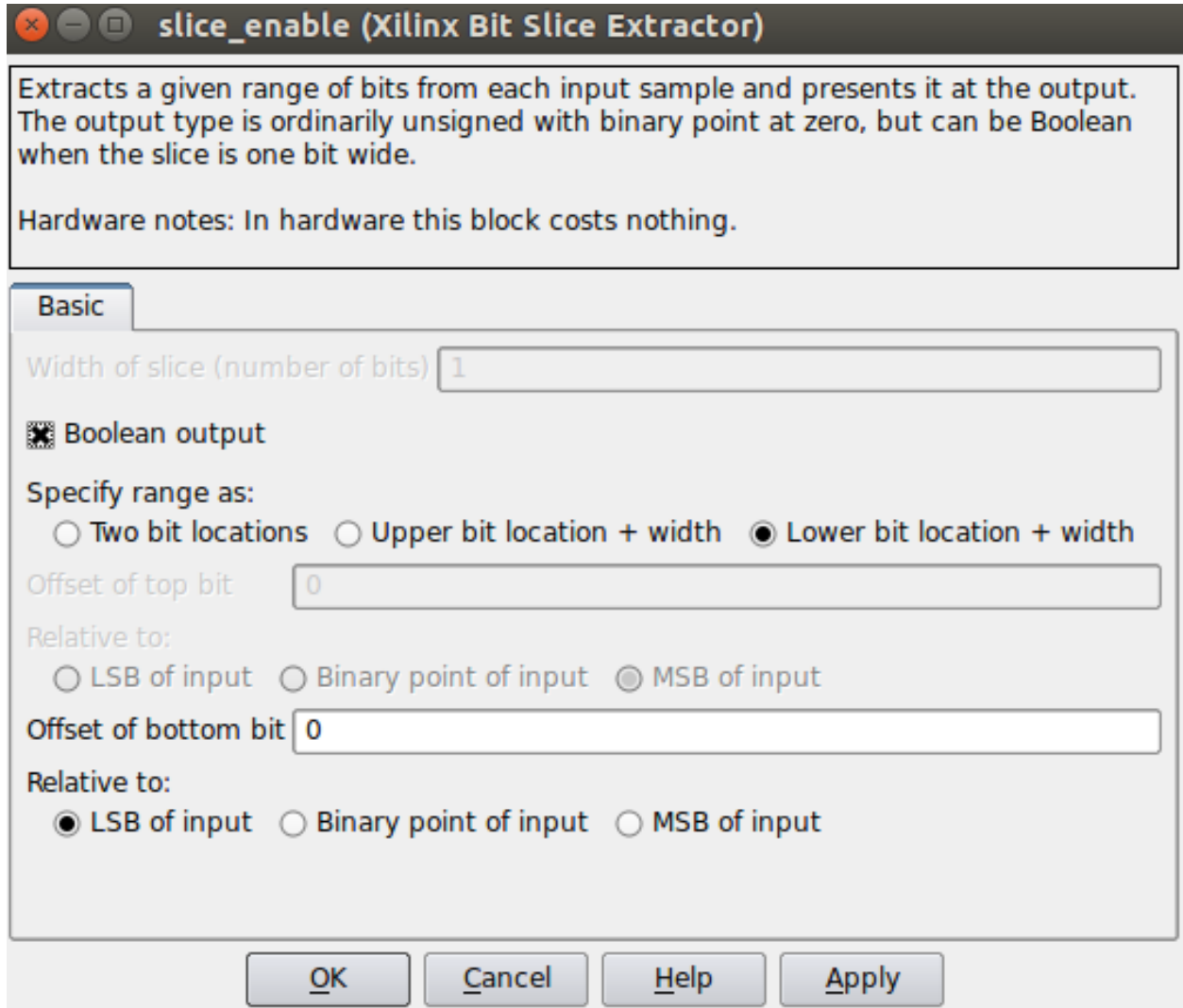
Add the slice blocks

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.

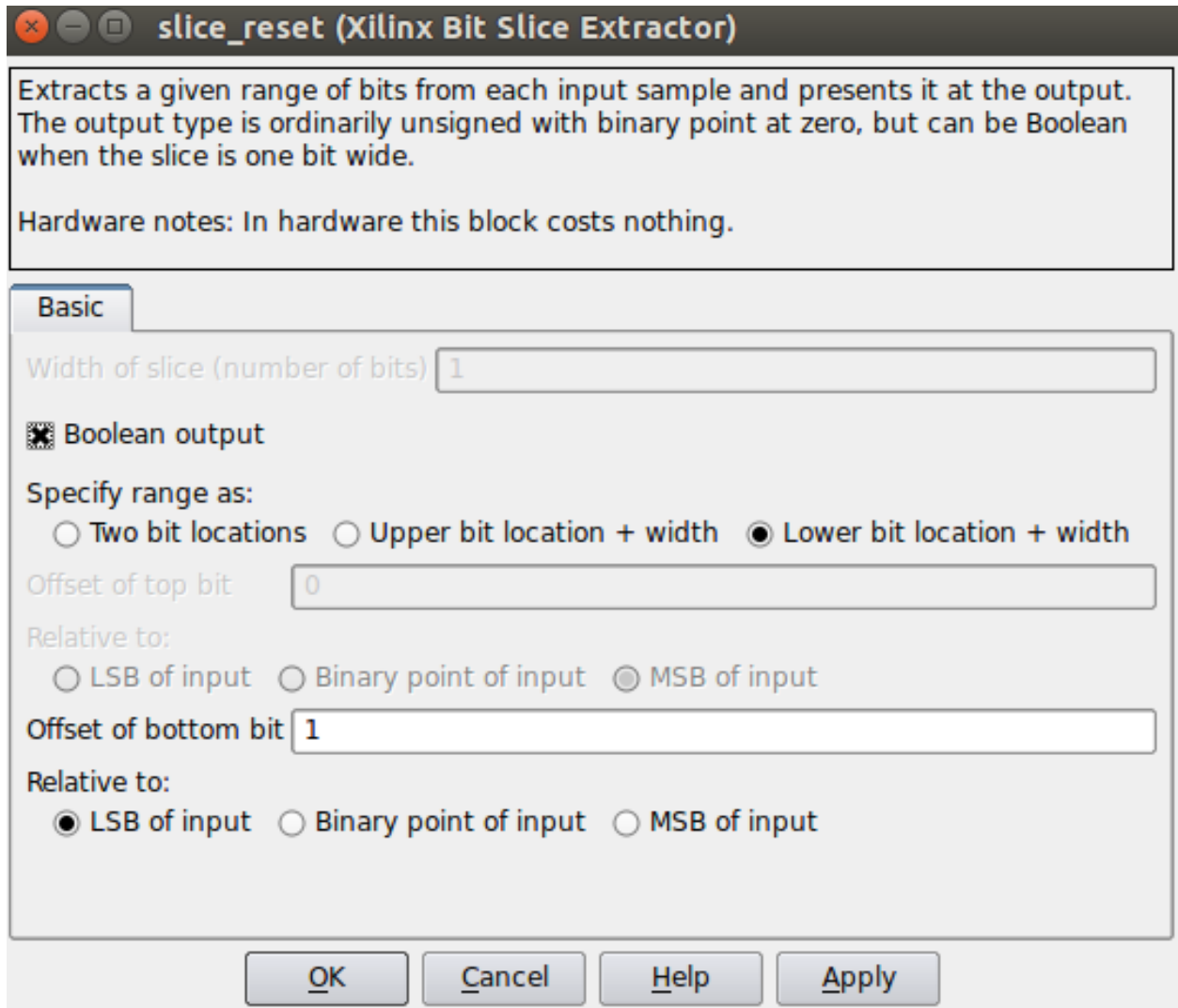
The enable and reset ports of the counter require boolean values (which Simulink interprets differently from ordinary 1-bit unsigned numbers). Configure the slices as follows:

Slice for enable:



casper_xps_param

Slice for reset:

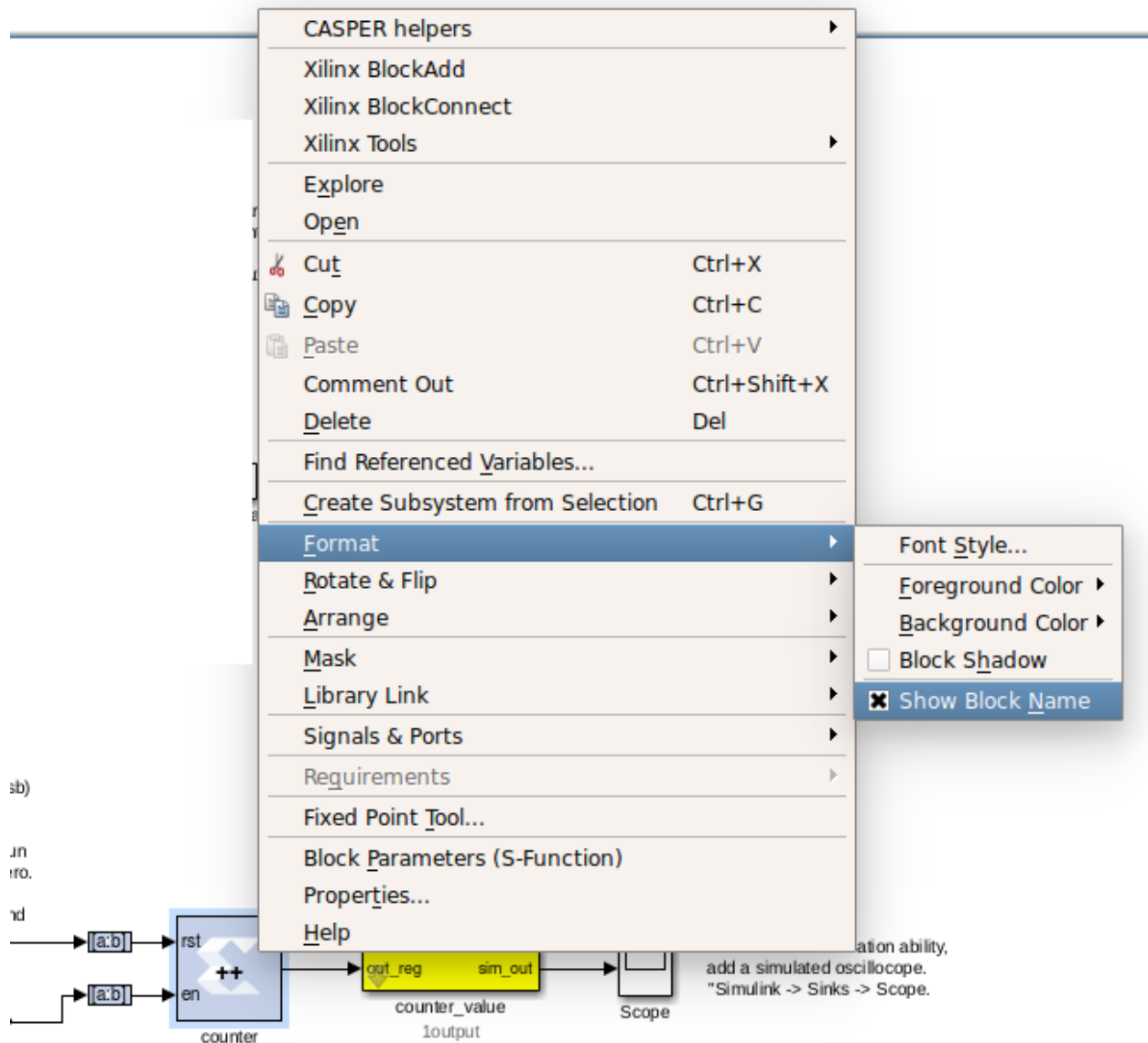


casper_xps_param

Connect 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.

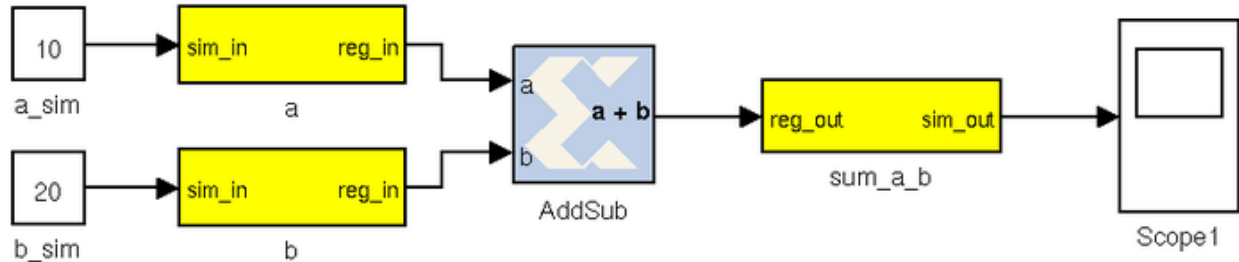
Do so by right-clicking and unchecking Format → Show Block Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



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 $a+b = \text{sum_a_b}$.



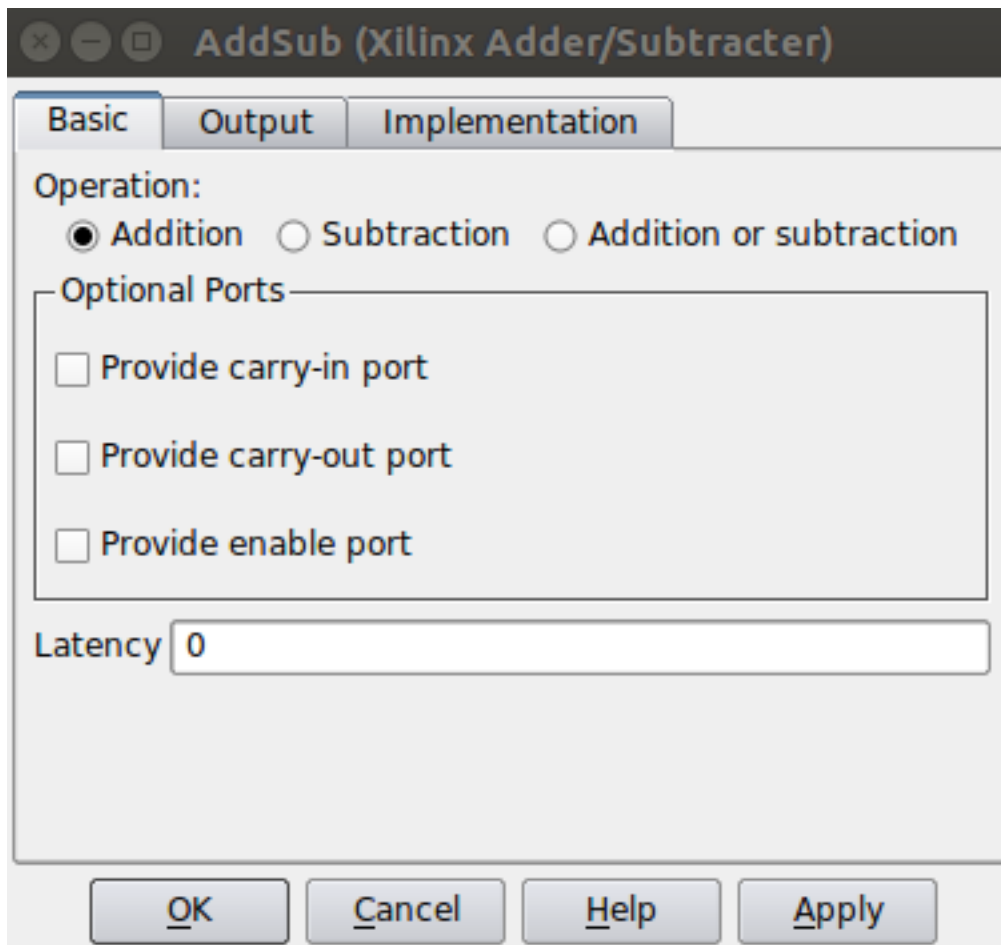
Add the software registers

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.

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.



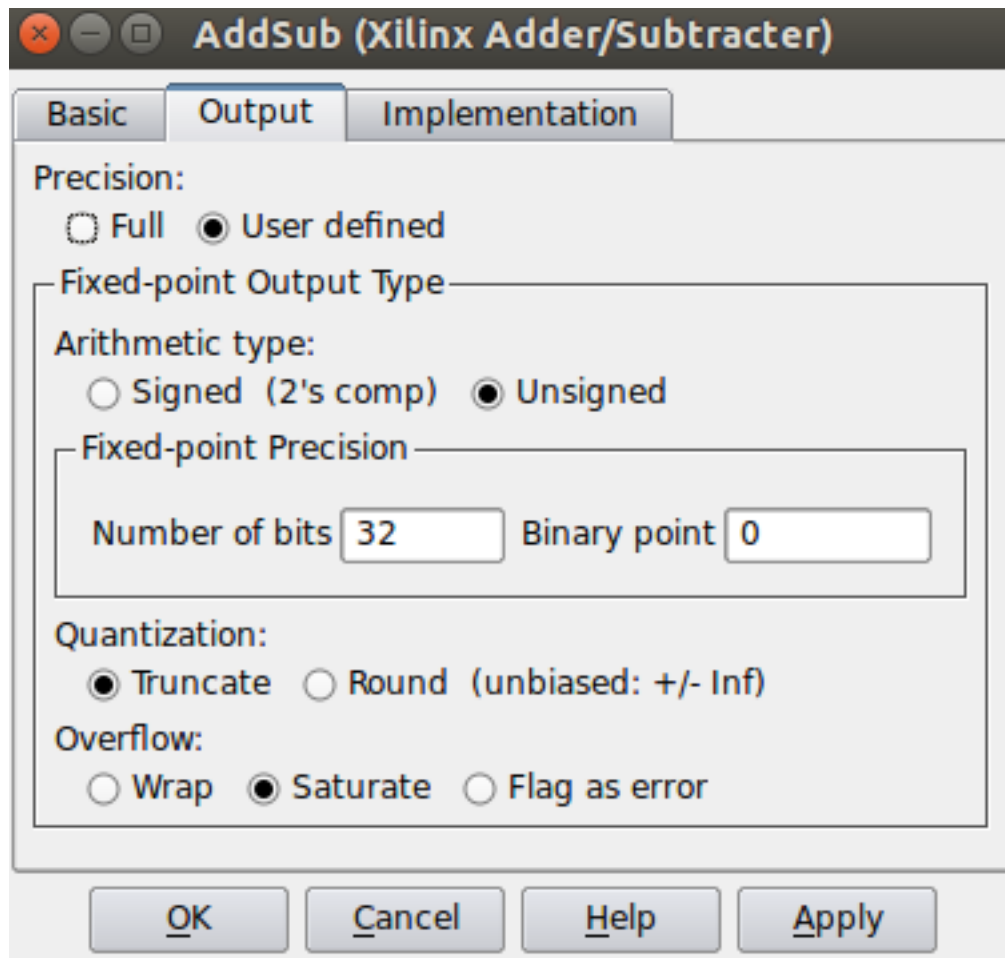
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

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 $2^{32} - 1$.

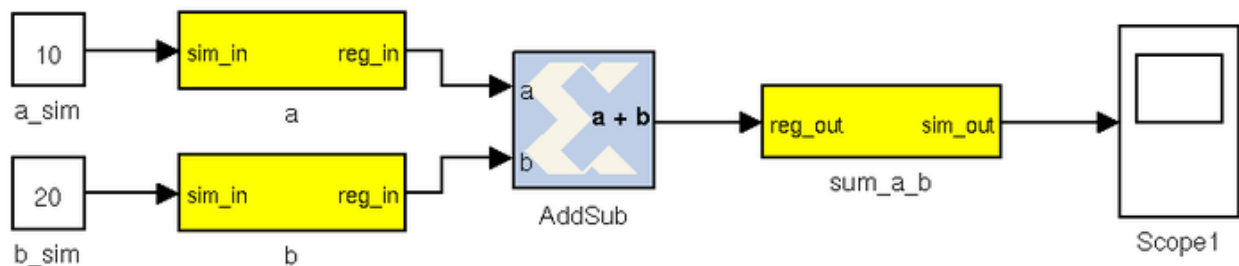


Add the scope and simulation inputs

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.

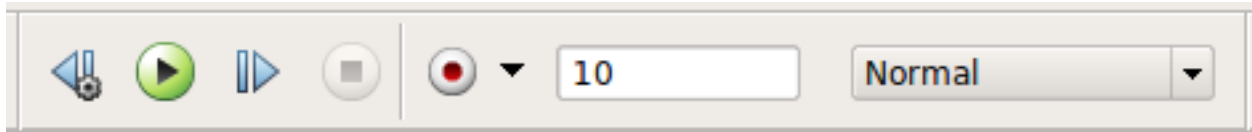
Connect it all together

Like this:



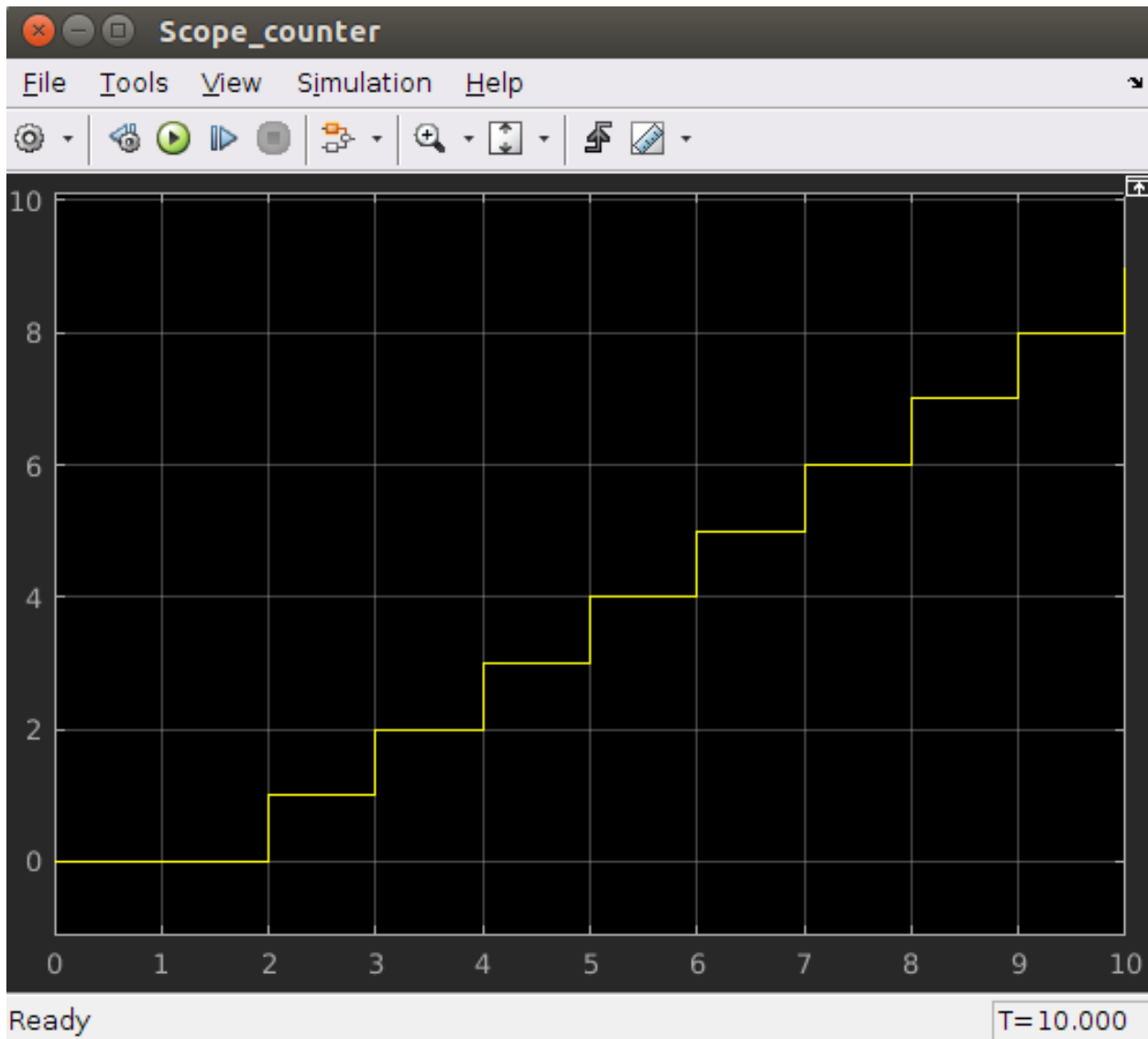
Simulating

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.

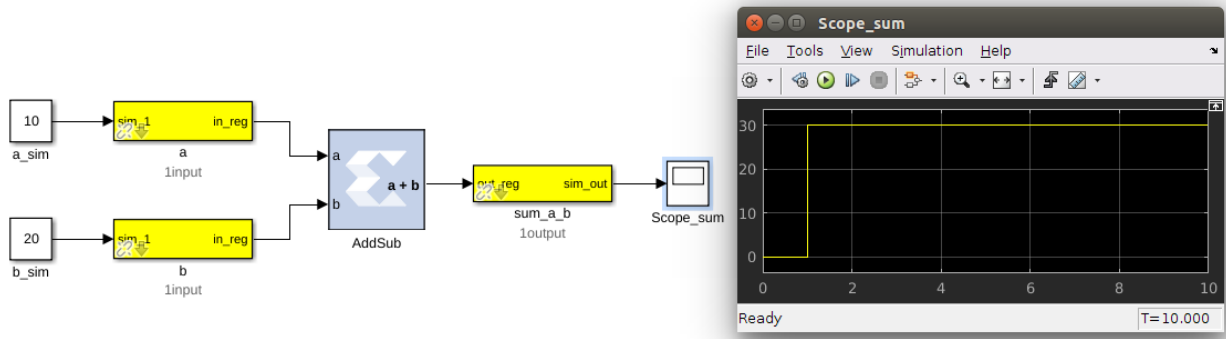


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

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...

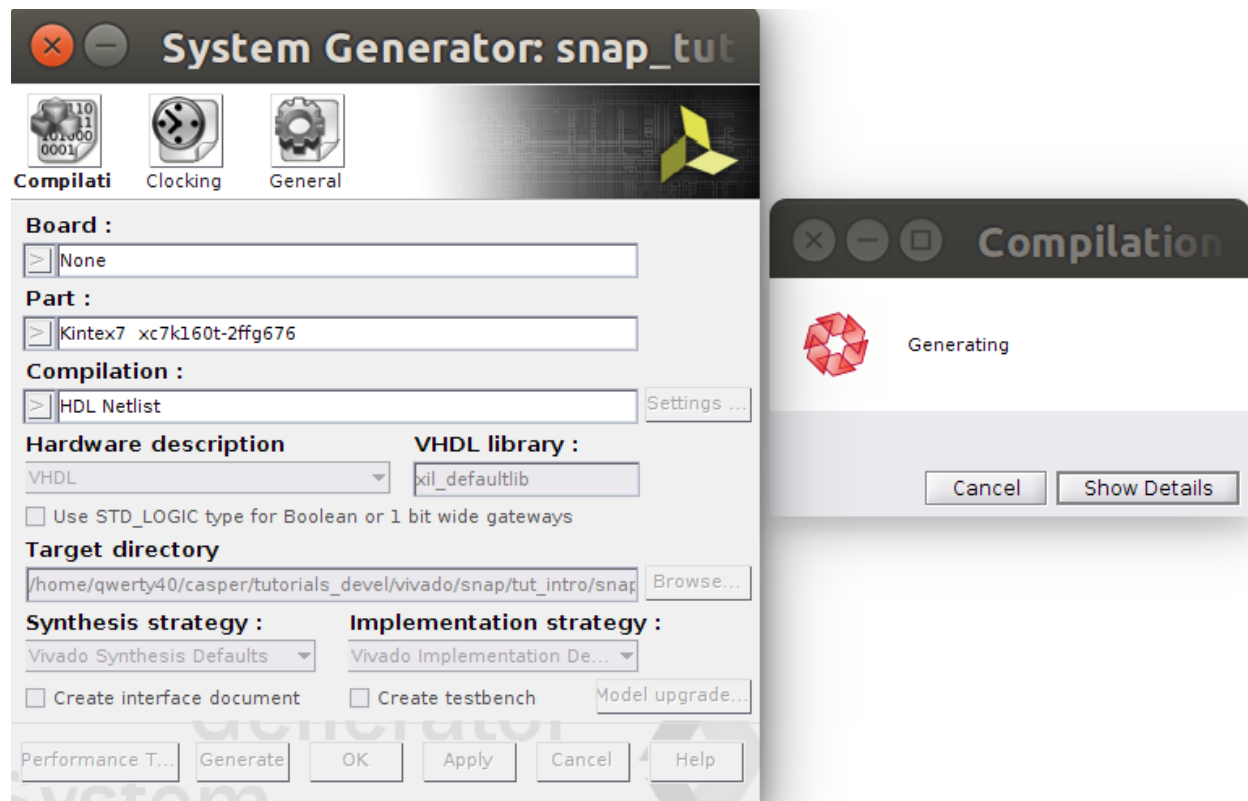
Compiling

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 156.25MHz system clock crystal and to your specified User IP Clock Rate, but they will operate independently.

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window. **THIS COMMAND DEPENDS WHICH PLATFORM YOU ARE TARGETING:**

```
>> jasper
```

When a GUI pops up, click "Compile!". This will run the complete build process, which consists of two stages. The first involving Xilinx's System Generator, which compiles any Xilinx blocks in your Simulink design to a circuit which can be implemented on your FPGA. While System Generator is running, you should see the following window pop up:



After this, the second stage involves synthesis of your design through Vivado, which goes about turning your design into a physical implementation and figuring out where to put the resulting components and signals on your FPGA. Finally the toolflow will create the final output fpg file that you will use to program your FPGA. This file contains the bitstream (the FPGA configuration information) as well as meta-data describing what registers and other yellow blocks are in your design. This file will be created in the 'outputs' folder in the working directory of your Simulink model. **Note: Compile time is approximately 15-20 minutes.**

```
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/
myproj/ outputs/ sysgen/
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/outputs/
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$ ls -lt
total 8336
-rw-rw-r-- 1 amish amish 1249412 Jul 25 14:18 tut_1_2017-7-25_1355.fpg
-rw-rw-r-- 1 amish amish 7280615 Jul 25 14:18 tut_1_2017-7-25_1355.bof
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$
```

Advanced Compiling

Once you are familiar with the CASPER toolflow, you might find you want to run the two stages of the compile separately. This means that MATLAB will become usable sooner, since it won't be locked up by the second stage of the compile. If you want to do this, you can run the first stage of the compile from the MATLAB prompt with

```
>> jasper_frontend
```

After this is completed, the last message printed will tell you how to finish the compile. It will look something like:

```
$ python /path_to/mlib_devel/jasper_library/exec_flow.py -m /home/user/path_to/snap/
↳tut_intro/snap_tut_intro.slx --middleware --backend --software
```

You can run this command in a separate terminal, after sourcing appropriate environment variables. Not recommended for beginners.

Programming the FPGA

Reconfiguration of CASPER FPGA boards is achieved using the casperfpga python library, created by the SA-SKA group.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration. However, should you want to run these tutorials on your own machines, you should download the latest casperfpga libraries from [here](#).

Copy your .fpg file to your Server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server. Instructions to do this are available [here](#)

Connecting to the board

SSH into the server that the SNAP board is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
$ ipython
```

Now import the fpga control library. This will automatically pull-in the KATCP library and any other required communications libraries.

```
import casperfpga
```

To connect to the board we create a CasperFpga instance; let's call it fpga. The CasperFpga constructor requires just one argument: the IP hostname or address of your FPGA board.

```
fpga = casperfpga.CasperFpga('SNAP hostname or ip_address')
```

The first thing we do is program the FPGA with the .fpg file which your compile generated.

```
fpga.upload_to_ram_and_program('your_fpgfile.fpg')
```

Should the execution of this command return true, you can safely assume the FPGA is now configured with your design. You should see the LED on your board flashing. Go check! All the available/configured registers can be displayed using:

```
fpga.listdev()
```

The adder and counter can be controlled by **writing to** and **reading from** registers added in the design using:

```
fpga.write_int('a',10)
fpga.write_int('b',20)
fpga.read_int('sum_a_b')
```

With any luck, the sum returned by the FPGA should be correct.

You can also try writing to the counter control registers in your design. You should find that with appropriate manipulation of the control register, you can make the counter start, stop, and return to zero.

```
fpga.write_int('counter_ctrl',10')
fpga.read_uint('counter_value')
```

Conclusion

This concludes the first CASPER Tutorial. You have learned how to construct a simple Simulink design, program an FPGA board and interact with it with Python using `casperfpga`. Congratulations!

While the design you made might not be very complicated, you now have the basic skills required to build more complex designs which are covered in later tutorials.

1.1.2 Tutorial 2: 10GbE Interface

Introduction

In this tutorial, you will create a simple Simulink design which uses the SNAP's 10GbE ports to send data at high speeds to another port. This could just as easily be another FPGA board or a computer with a 10GbE network interface card. In addition, we will learn to control the design remotely, using a supplied Python library for KATCP.

In this tutorial, a counter will be transmitted through one SFP+ port and back into another. This will allow a test of the communications link. This test can be used to test the link between boards and the effect of different cable lengths on communication robustness.

Background

SNAP boards have two on-board SFP+ ports. The Ethernet interface is driven by an on-board 156.25MHz crystal oscillator. This clock is then multiplied up on the FPGA by a factor of 66. Thus, the speed on the wire is actually $156.25\text{MHz} \times 66 = 10.3125\text{ Gbps}$. However, 10GbE over single-lane SFP+ connectors uses 64b/66b encoding, which means that for every 66 bits sent, 64 bits are actually transmitted. This is to ensure proper clocking, since the receiver recovers and locks-on to the transmitter's clock and requires edges in the data. Imagine transmitting a string of 0xFF or 0b11111111... which would otherwise generate a DC level on the line, now an extra two bits are introduced which includes a zero bit which the receiver can use to recover the clock and byte endings. See [here](#) for more information.

For this reason, we actually get 10Gbps usable data rate. CASPER's 10GbE Simulink core sends and receives UDP over IPv4 packets. These IP packets are wrapped in Ethernet frames. Each Ethernet frame requires a 38 byte header, IPv4 requires another 20 bytes and UDP a further 16. So, for each packet of data you send, you will incur a cost of at least 74 bytes. I say at least, because the core will zero-pad some headers to be on a 64-bit boundary. You will thus never achieve 10Gbps of usable throughput, though you can get close. It pays to send larger packets if you are trying to get higher throughputs.

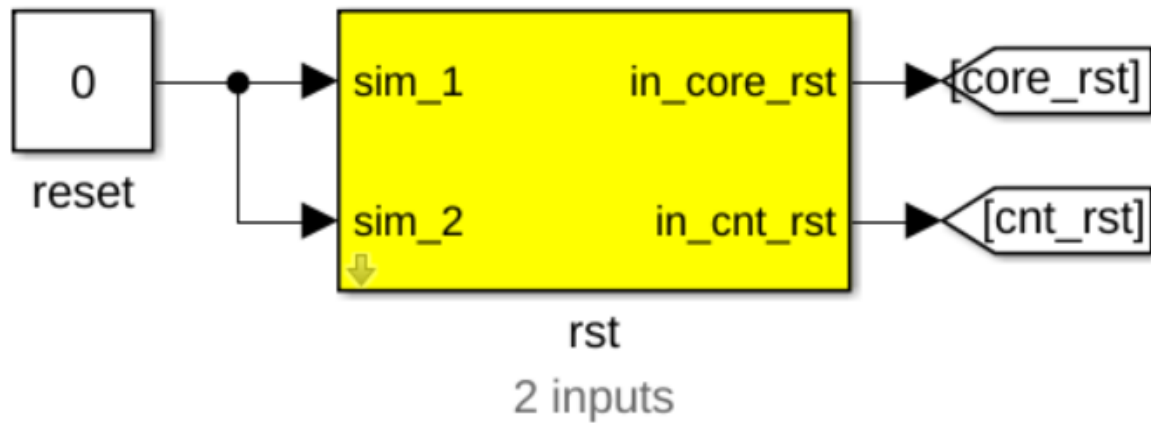
The maximum payload length of the CASPER 10GbE core is 8192 bytes (implemented in BRAM) plus another 512 (implemented in distributed RAM) which is useful for an application header. These ports (and hence part of the 10 GbE cores) run at 156.25MHz, while the interface to your design runs at the FPGA clock rate (sys_clk, adcX_clk etc). The interface is asynchronous, and buffers are required at the clock boundary. For this reason, even if you send data between two SNAP boards which are running off the same hard-wired clock, there will be jitter in the data. A second consideration is how often you clock values into the core when you try to send data. If your FPGA is running faster than the core, and you try and clock data in on every clock cycle, the buffers will eventually overflow. Likewise for receiving, if you send too much data to a board and cannot clock it out of the receive buffer fast enough, the receive buffers will overflow and you will lose data. In our design, we are clocking the FPGA at 100 MHz, with the cores running at 156.25MHz. We can thus clock data into the TX buffer continuously without worrying about overflows.

Create a new model

Start Matlab and open Simulink (either by typing 'simulink' on the Matlab command line, or by clicking on the Simulink icon in the taskbar). A template is provided for Tut2 with a pre-created packet generator in the tutorials_devel git repository. Get a copy of this template and save it. You will need the SNAP block in the Platforms subdirectory of the xps_library. Specify a clock frequency of 100 MHz and the clock source "sys_clock".

Add reset logic

A very important piece of logic to consider when designing your system is how, when and what happens during reset. In this example we shall control our resets via a software register. We shall have two independent resets, one for the 10GbE cores which shall be used initially, and one to reset the user logic which may be used more often to restart the user part of the system. Construct reset circuitry as shown below.



Add a software register

Use a software register yellow block from the CASPER XPS System Blockset for the rst block. Rename it to rst.

It used to be that every register you inserted had to be natively 32-bits, and you were responsible for slicing these 32 bits into different signals if you want to control multiple flags. The latest block can implicitly break the 32-bit registers out into separate names signals, so we'll use that. The downside is there are a bunch of settings to configure – you need to set up the names and data types of your register subfields. You can configure the register as follows:

Block Parameters: rst

swreg (mask)

A 32-bit software-accessible register.
Can be divided into bitfields of
varying widths and types using the
fields in the block mask.

Setup

I/O direction

I/O delay

Initial Value

Sample period

Bitfield names [msb...lsb]

Bitfield widths

Add Goto blocks

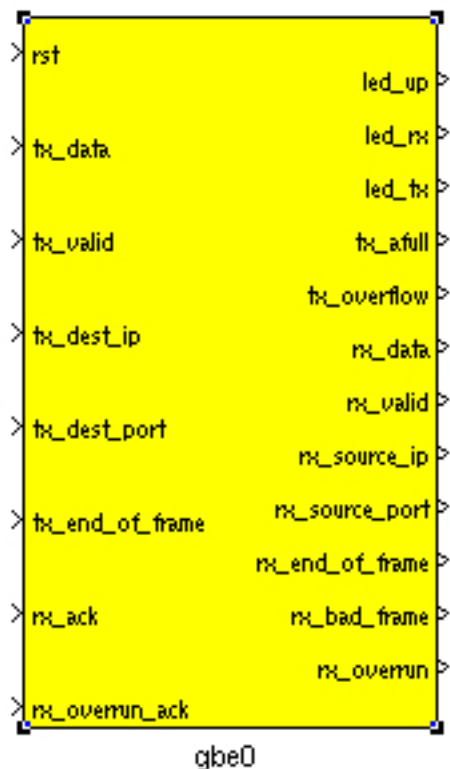
Add two Goto blocks from Simulink->Signal Routing. Configure them to have the tags as shown (core_rst and cnt_rst). These tags will be used by associated From (also found in Simulink->Signal Routing) blocks in other parts of the design. These help to reduce clutter in your design and are useful for control signals that are routed to many destinations. They should not be used a lot for data signals as it reduces the ease with which data flow can be seen through the system.

Add 10GbE and associated registers for data transmission

We will now add the 10GbE block to transmit a counter at a programmable rate.

Add a 10GbE block for data transmission

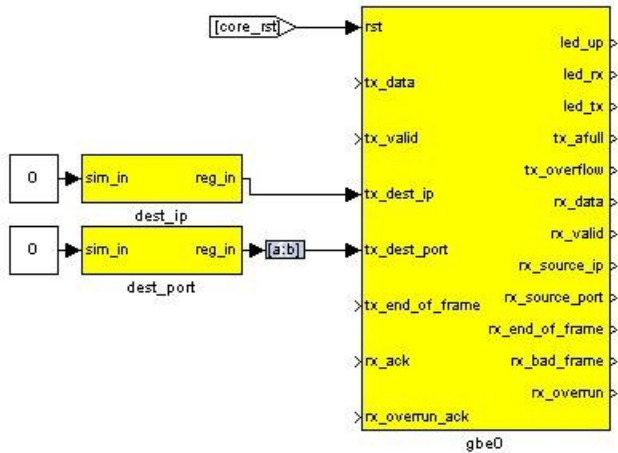
Add a ten_GbE yellow block from the CASPER XPS System Blockset. It will be used to transmit data and we shall add another later to receive data. Rename it gbe0. Double click on the block to configure it and set it to be associated with SFP+ port 0. If your application can guarantee that it will be able to use received data straight away (as our application can), shallow receive buffers can be used to save resources. This optimisation is not necessary in this case as we will use a small fraction of resources in the FPGA.



Add registers to provide the target IP address and port number

Add two yellow-block software registers to provide the destination IP address and port number for transmission with the data. Name one dest_ip and the other dest_port. The registers should be configured to receive their values from

the processor. Connect them to the appropriate inputs of the gbe0 10GbE block as shown. A Slice block is required to use the lower 16 bits of data from the dest_port register. Constant blocks from Simulink->Sources with 0 values are attached to the simulation inputs of the software registers. The destination port and IP address are not important in this system as it is a loopback example. Add a From block from Simulink->Signal Routing and set the tag to use core_rst, this enables one to reset the block.



Create a subsystem to generate a counter to transmit as data

We will now implement logic to generate a counter to transmit as data. This is already included in the Template for Tut 2. Some details are provided here for completeness.

Construct a subsystem for data generation logic

It is often useful to group related functionality and hide the details. This reduces drawing space and complexity of the logic on the screen, making it easier to understand what is happening. Simulink allows the creation of Subsystems to accomplish this.

These can be copied to places where the same functionality is required or even placed in a library for use in other projects and by other people. To create a subsystem, one can highlight the logical elements to be encapsulated, then right-click and choose Create Subsystem from the list of options. You can also simply add a Subsystem block from Simulink->Ports & Subsystems.

Subsystems inherit variables from their parent system. Simulink allows one to create a variable whose scope is only a particular subsystem. To do this, right-click on a subsystem and choose the Create Mask option. The mask created for that particular subsystem allows one to add parameters that appear when you double-click on the icon associated with the subsystem.

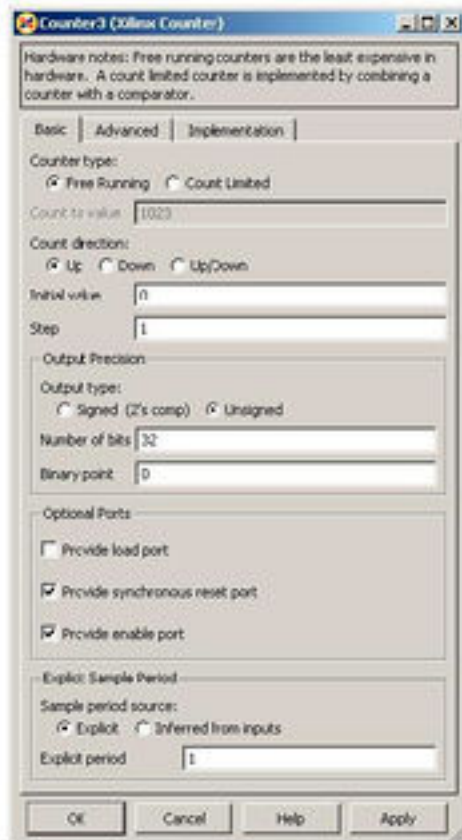
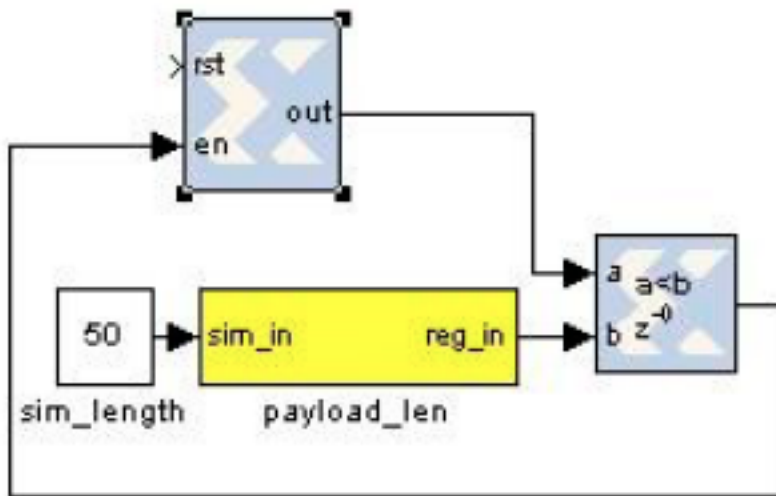
The mask also allows you to associate an initialisation script with a particular subsystem. This script is called every time a mask parameter is modified and the Apply button clicked. It is especially useful if the internal structure of a subsystem must change based on mask parameters. Most of the interesting blocks in the CASPER library use these initialisation scripts.

Drop a subsystem block into your design and rename it pkt_sim. Then double-click on it to add logic.

Add a counter to generate a certain amount of data

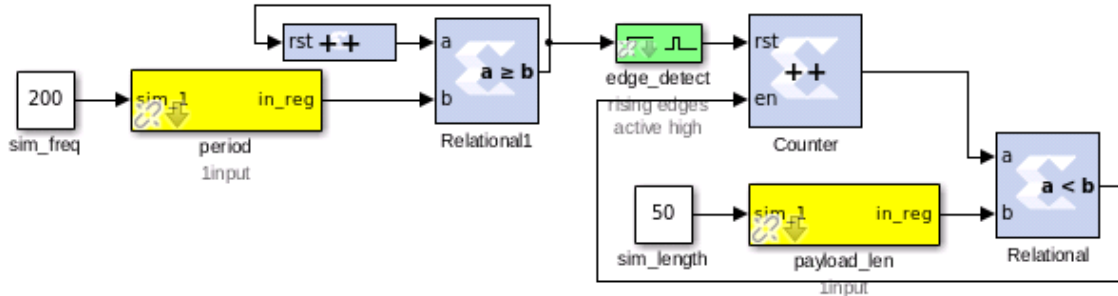
Add a Counter block from Xilinx Blockset->Basic Elements and configure it to be unsigned, free-running, 32-bits, incrementing by 1 as shown. Add a Relational block, software register and Constant block as shown. In simulation

this circuit will generate a counter from 0 to 49 and then stop counting. This will allow us to generate 50 data elements before stopping.



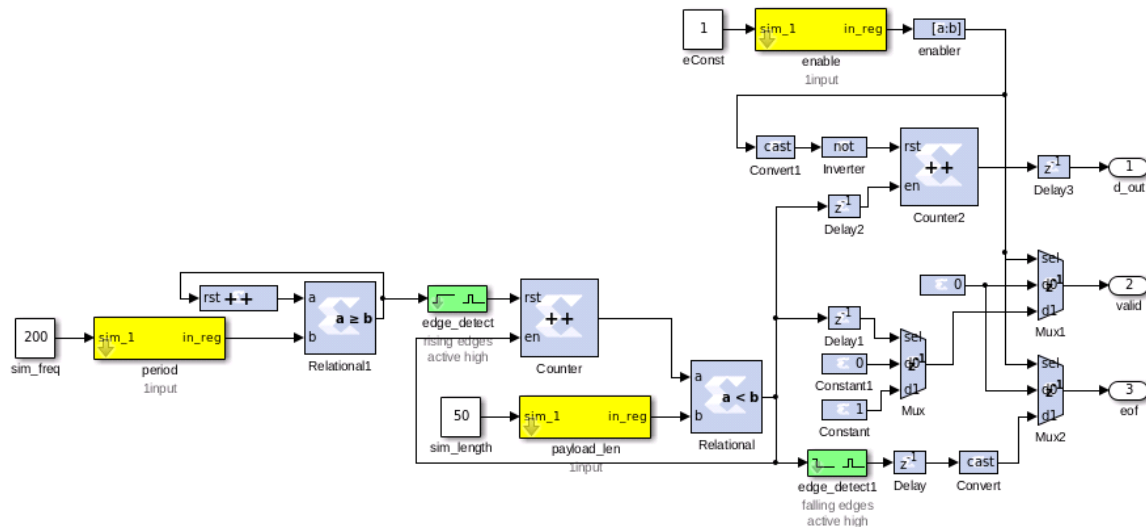
Add a counter to limit the data rate

As mentioned earlier in this tutorial, it is impossible to supply data to the 10GbE transmission block at the full clock rate. This would mean transmitting a 64-bit word at 200MHz, and the 10GbE standard only supports up to 156.25MHz data transmission. We thus want to generate data in bursts such that the transmission FIFOs do not overflow. We thus add circuitry to limit the data rate as shown below. The logic that we have added on the left generates a reset at a fixed period determined by the software register. This will trigger the generation of a new packet of data as before. In simulation this allows us to limit the data rate to $50/200 * 200\text{MHz} = 50\text{MHz}$. Using these values in actual hardware would limit the data rate to $(50/(8/10 * 156.25)) = 4\text{Gbps}$.



Finalise logic including counter to be used as data

We will now finalise the data generation logic as shown below. To save time, use the existing logic provided with the tutorial. Counter1 in the illustration generates the actual data to be transmitted and the enable register allows this data stream to the transmitting 10GbE core to be turned off and on. Logic linked to the eof output port provides an indication to the 10GbE core that the final data word for the frame is being sent. This will trigger the core to begin transmission of the frame of data using the IP address and port number specified.



Receive blocks and logic

The receive logic is composed of another 10GbE yellow block with the transmission interface inputs all tied to 0 as no transmission is to be done, however Simulink requires all inputs to be connected. Connecting them to 0 should ensure that during synthesis the transmission logic for this 10GbE block is removed. Double click on the block to configure it and set it to be associated with SFP+ port 1.

Buffers to capture received and transmitted data

The casperfpga Python package contains all kinds of methods to interact with your 10GbE cores. For example, grabbing packets from the TX and RX stream, or counting the number of packets sent and received are all supported, as long as you turn on the appropriate functionality in the 10GbE yellow block. The settings we'll use are –

Block Parameters: gbe0

ten_gbe (mask)

This block sends and receives UDP frames (packets). It accepts a 64 bit wide data stream with user-determined frame breaks. The data stream is wrapped in a UDP frame for transmission. Incoming UDP packets are unwrapped and the data presented as a 64 bit wide stream.

Core Debug counters

Port 0

☐ Shallow RX Fifo (Beware overruns!)

☒ Enable Large TX Frames (8k+512)

☒ ----- Show Implementation Parameters -----

Pre-emphasis 3

Differential swing 800

☒ Enable fabric on startup

Fabric MAC Address

hex2dec('123456780000')

Fabric IP Address

$192 \cdot (2^{24}) + 168 \cdot (2^{16}) + 5 \cdot (2^8) + 20 \cdot (2^0)$

Fabric UDP Port

10000

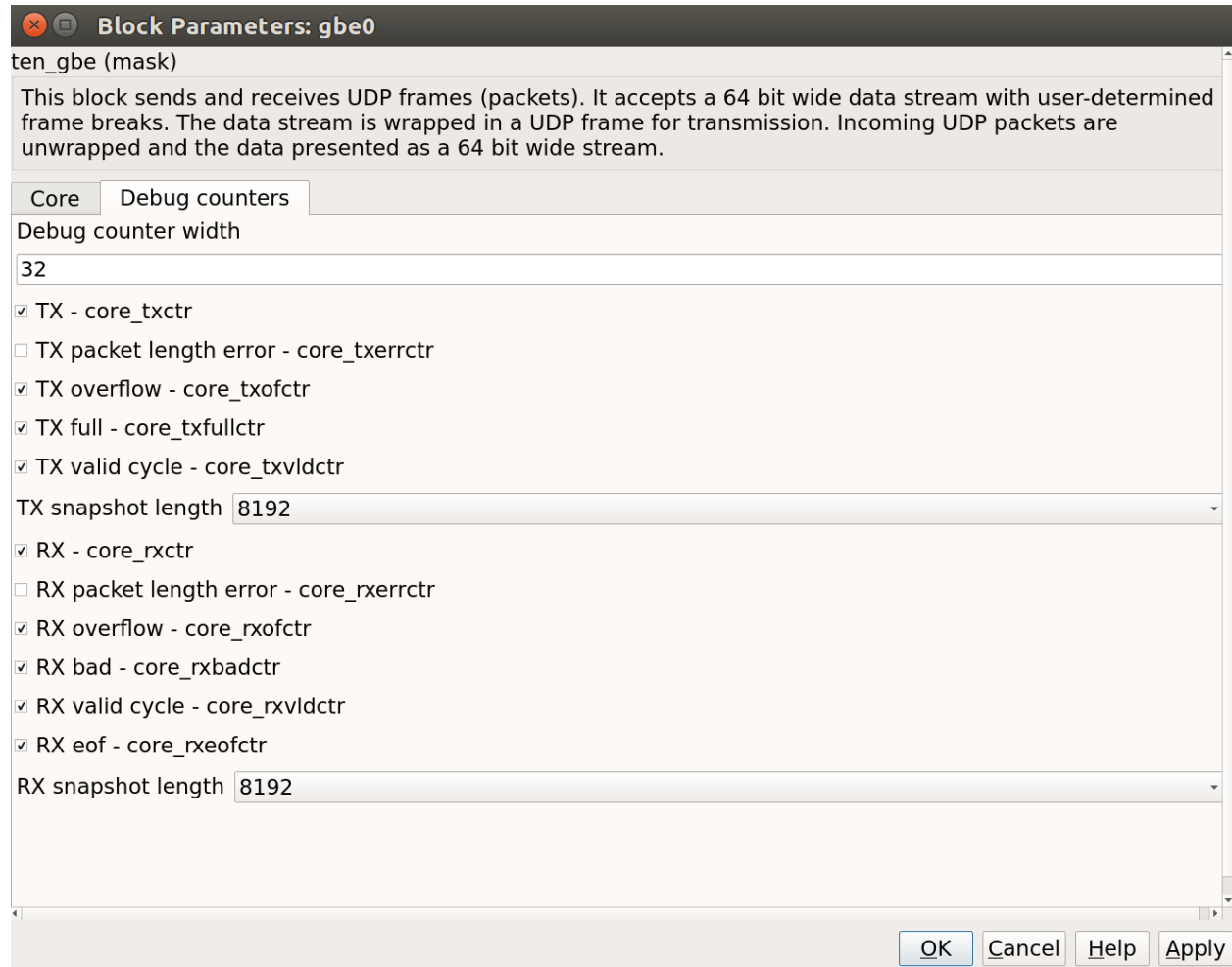
Fabric Gateway

1

☒ Enable CPU RX

☒ Enable CPU TX

OK Cancel Help Apply



You can see how to use these functions in the software that accompanies this tutorial.

LEDs and status registers

You can also sprinkle around other registers or LEDs to monitor status of core parameters, or give visual feedback that the design is doing something sane. Check out the reference model for some examples of potentially useful monitoring circuitry.

Compilation

Compiling this design takes approximately 20 to 30 minutes. A pre-compiled binary (.fpg file) is made available to save time.

Programming and interacting with the FPGA

A pre-written python script, “`snap_tut_tge.py`” is provided. This script programs the fpga with your complied design (.fpg file) configures the 10GbE Ports and initiates data transfer. The script is run using:

```
./snap_tut_tge.py <SNAP_IP_ADDRESS>
```

If everything goes as expected, you should see a whole bunch of lines running across your screen as the code sets up the IP/MAC parameters of the 10GbE cores and checks their status, and that the data the cores are sending and receiving are consistent. Have a look at this code to see how one uses the more advanced (i.e. more complex than `read_int`, and `write_int`) methods `casperfpga` makes available. Documentation for `casperfpga` is still a work in progress(!) but the basic idea is that when you instantiate a `CasperFpga`, the software intelligently builds python objects into this instance, based on what you put in your design. For example, your Ethernet cores should show up as objects `CasperFpga.gbes.<simulink_block_name>` (or `CasperFpga.gbes['simulink_block_name']`) which have useful methods like “`setup`”, which sets the core’s IP/MAC address, or “`print_10gbe_core_details`” which will print out useful status information, like the current state of the core’s ARP cache. iPython and tab-complete are your friend here, there are lots of handy methods to discover. (I’m still discovering them now :))

The control software should be(!) well-commented, to explain what’s going on behind the scene as the software interacts with your FPGA design.

Conclusion

This concludes Tutorial 2. You have learned how to utilize the 10GbE ports on a SNAP to send and receive UDP packets. You also learned how to further use the Python to program the FPGA and control it remotely using some of the OOP goodies available in `casperfpga`.

1.1.3 Tutorial 3: Wideband Spectrometer

Introduction

A spectrometer is something that takes a signal in the time domain and converts it to the frequency domain. In digital systems, this is generally achieved by utilising the FFT (Fast Fourier Transform) algorithm. However, with a little bit more effort, the signal to noise performance can be increased greatly by using a Polyphase Filter Bank (PFB) based approach.

When designing a spectrometer for astronomical applications, it’s important to consider the science case behind it. For example, pulsar timing searches will need a spectrometer which can dump spectra on short timescales, so the rate of change of the spectra can be observed. In contrast, a deep field HI survey will accumulate multiple spectra to increase the signal to noise ratio. It’s also important to note that “bigger isn’t always better”; the higher your spectral and time resolution are, the more data your computer (and scientist on the other end) will have to deal with. For now, let’s skip the science case and familiarize ourselves with an example spectrometer.

Setup

This tutorial comes with a completed model file, a compiled bitstream, ready for execution on ROACH, as well as a Python script to configure the ROACH and make plots. [Here](#)

Spectrometer Basics

When designing a spectrometer there are a few main parameters of note:

- **Bandwidth:** The width of your frequency spectrum, in Hz. This depends on the sampling rate; for complex sampled data this is equivalent to:

$$BW = \text{sampling rate} = \frac{1}{\text{sampling period}}$$

In contrast, for real or Nyquist sampled data the rate is half this:

$$BW = \frac{\text{sampling rate}}{2} = \frac{1}{2 \times \text{sampling period}}$$

as two samples are required to reconstruct a given waveform .

- **Frequency resolution:** The frequency resolution of a spectrometer, Δf , is given by

$$\Delta f = \frac{BW}{\text{no. channels}},$$

and is the width of each frequency bin. Correspondingly, Δf is a measure of how precise you can measure a frequency.

- **Time resolution:** Time resolution is simply the spectral dump rate of your instrument. We generally accumulate multiple spectra to average out noise; the more accumulations we do, the lower the time resolution. For looking at short timescale events, such as pulsar bursts, higher time resolution is necessary; conversely, if we want to look at a weak HI signal, a long accumulation time is required, so time resolution is less important.

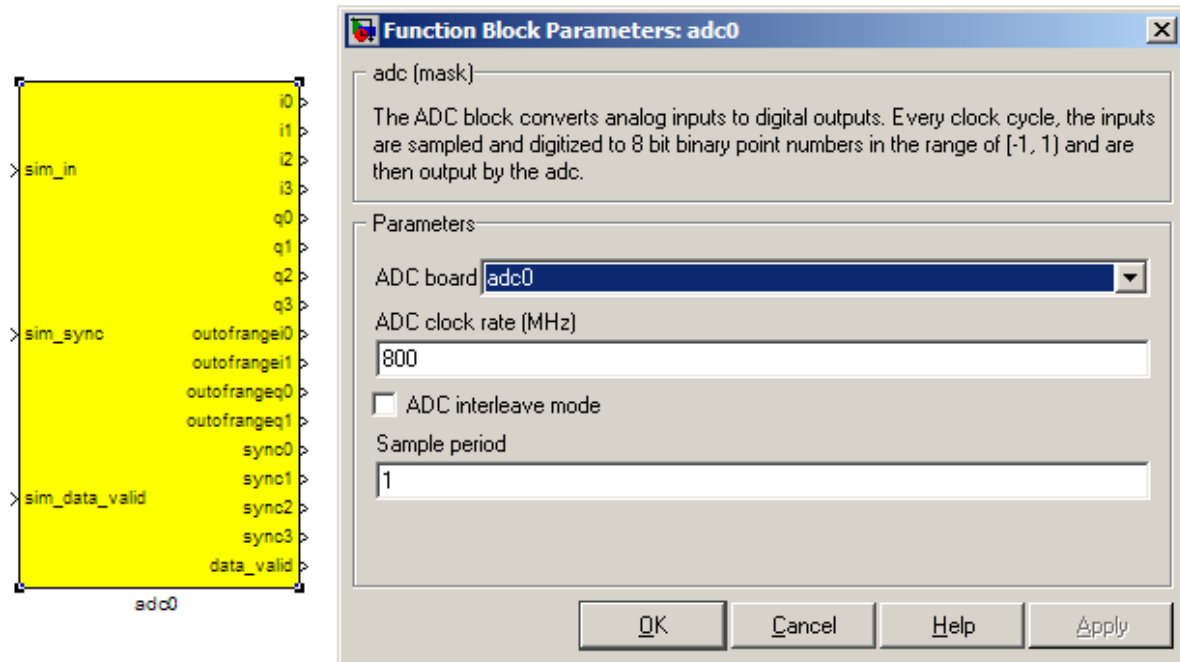
Simulink / CASPER Toolflow

Simulink Design Overview

If you're reading this, then you've already managed to find all the tutorial files. Jason has gone to great effort to create an easy to follow simulink model that compiles and works. By now, I presume you can open the model file and have a vague idea of what's happening. The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some "blockumentation" in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- The all important Xilinx token is placed to allow System Generator to be called to compile the design.
- In the MSSGE block, the hardware Platform is set to "SNAP:xc7k160t" and clock rate is specified as 200MHz.
- The input signal is digitised by the ADC, resulting in four parallel time samples of 8 bits each clock cycle. The ADC runs at 800MHz, which gives a 400MHz nyquist sampled spectrum. The output range is a signed number in the range -1 to +1 (ie 7 bits after the decimal point). This is expressed as fix_8_7.
- The four parallel time samples pass through the pfb_fir_real and fft_wideband_real blocks, which together constitute a polyphase filter bank. We've selected 212=4096 points, so we'll have a 211=2048 channel filter bank.
- You may notice Xilinx delay blocks dotted all over the design. It's common practice to add these into the design as it makes it easier to fit the design into the logic of the FPGA. It consumes more resources, but eases signal timing-induced placement restrictions.
- The real and imaginary (sine and cosine value) components of the FFT are plugged into power blocks, to convert from complex values to real power values by squaring. They are also scaled by a gain factor before being quantised...
- The requantized signals then enter the vector accumulators, vacc0 and vacc1, which are simple_bram_vacc 64 bit vector accumulators. Accumulation length is controlled by the acc_cntrl block.
- The accumulated signal is then fed into software registers, odd and even.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

adc

The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter. In Simulink, the ADC daughter board is represented by a yellow block. Work through the “iADC tutorial” if you’re not familiar with the iADC card.

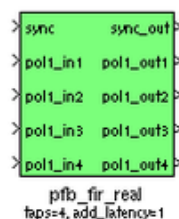
The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 8 bit binary point numbers in the range of -1 to 1 and are then output by the ADC. This is achieved through the use of two’s-complement representation with the binary point placed after the seven least significant bits. This means we can represent numbers from -128/128 through to 127/128 including the number 0. Simulink represents such numbers with a `fix_8_7` moniker.

ADCs often internally bias themselves to halfway between 0 and -1. This means that you’d typically see the output of an ADC toggling between zero and -1 when there’s no input. It also means that unless otherwise calibrated, an ADC will have a negative DC offset.

The ADC has to be clocked to four times that of the FPGA clock. In this design the ADC is clocked to 800MHz, so the ROACH will be clocked to 200MHz. This gives us a bandwidth of 400MHz, as Nyquist sampling requires two samples (or more) each second.

INPUTS**OUTPUTS**

The ADC outputs two main signals: *i* and *q*, which correspond to the coaxial inputs of the ADC board. In this tutorial, we’ll only be using input *i*. As the ADC runs at 4x the FPGA rate, there are four parallel time sampled outputs: *i0*, *i1*, *i2* and *i3*. As mentioned before, these outputs are 8.7 bit.

pfb_fir_real

Function Block Parameters: pfb_fir_real

pfb_fir_real (mask)

Fold adders into DSPs: Causes adders to be absorbed into DSP blocks (supported in Virtex5)
 Adder implementation: Cores using Fabric or DSP48 or behavioral HDL

Parameters

Size of PFB: (2[?] pnts)
 12

Total Number of Taps:
 4

Windowing Function: hamming

Number of Simultaneous Inputs: (2[?])
 2

Make Biplex
 0

Input Bitwidth:
 8

Output Bitwidth:
 18

Coefficient Bitwidth:

OK Cancel Help Apply

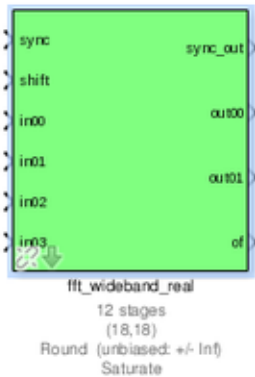
There are two main blocks required for a polyphase filter bank. The first is the pfb_fir_real block, which divides the signal into parallel ‘taps’ then applies finite impulse response filters (FIR). The output of this block is still a time-domain signal. When combined with the FFT_wideband_real block, this constitutes a polyphase filterbank.

INPUTS/OUTPUTS

As the ADC has four parallel time sampled outputs: i0, i1, i2 and i3, we need four parallel inputs for this PFB implementation.

PARAMETERS

fft_wideband_real



The dialog box is titled 'Function Block Parameters: fft_wideband_real'. It contains the following settings:

- fft_wideband_real (mask)**: A real-sampled wideband FFT.
- Basic** tab is selected.
- Number simultaneous streams**: 1
- Size of FFT: (2^? pts)**: 12
- Number of Simultaneous Inputs: (2^?)**: 2
- Input Bit Width**: 18
- Input binary point**: 17
- Coefficient Bit Width**: 18
- ☒ **Unscramble output (ie, put channels in canonical order)**
- ☐ **Asynchronous operation**

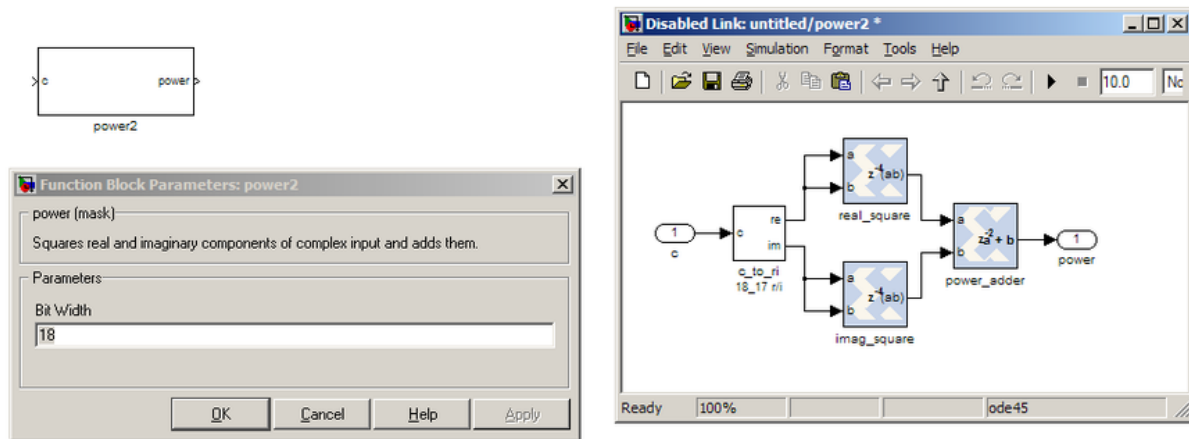
Buttons at the bottom: OK, Cancel, Help, Apply.

The FFT_wideband_real block is the most important part of the design to understand. The cool green of the FFT block hides the complex and confusing FFT butterfly biplex algorithms that are under the hood. You do need to have a working knowledge of it though, so I recommend reading Chapter 8 and Chapter 12 of Smith's free online DSP guide at (<http://www.dspguide.com/>). Parts of the documentation below are taken from the [[Block_Documentation | block documentation]] by Aaron Parsons and Andrew Martens.

INPUTS/OUTPUTS

PARAMETERS

power

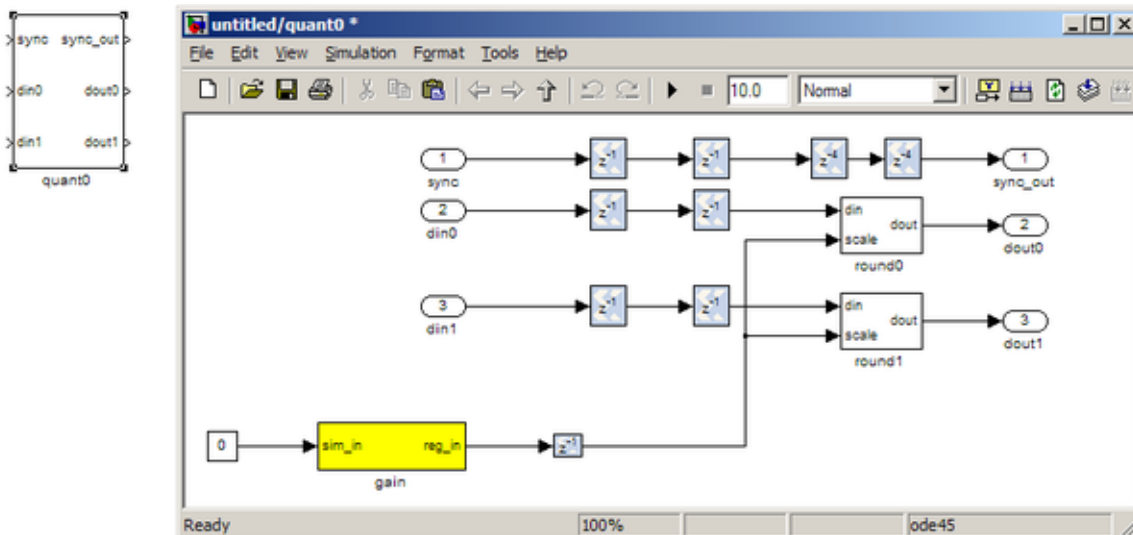


The power block computes the power of a complex number. The power block typically has a latency of 5 and will compute the power of its input by taking the sum of the squares of its real and imaginary components. The power block is written by Aaron Parsons and online documentation is by Ben Blackman. In our design, there are two power blocks, which compute the power of the odd and even outputs of the FFT. The output of the block is 36.34 bits; the next stage of the design re-quantizes this down to a lower bitrate.

INPUTS/OUTPUTS

PARAMETERS

quant



The quant0 was written by Jason Manley for this tutorial and is not part of the CASPER blockset. The block re-quantizes from 36.34 bits to 6.5 unsigned bits, in preparation for accumulation by the 32 bit bram_vacc block. This block also adds gain control, via a software register. The tut3.py script sets this gain control. You would not need to re-quantize if you used a larger vacc block, such as the 64bit one, but it's illustrative to see a simple example of

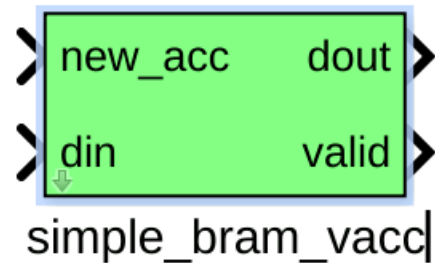
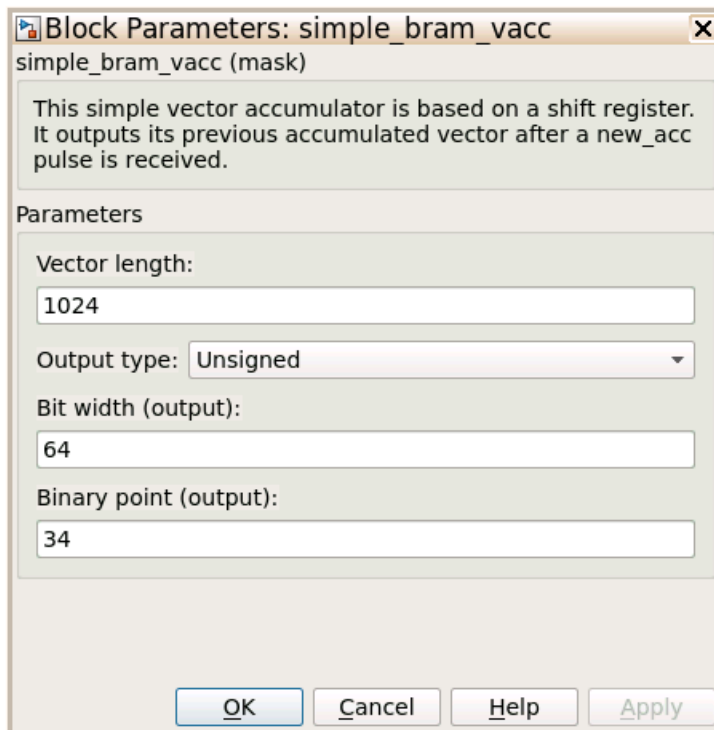
re-quantization, so it's in the design anyway. Note that the sync_out port is connected to a block, acc_ctrl, which provides accumulation control.

INPUTS/OUTPUTS

PARAMETERS

None.

simple_bram_vacc



The simple_bram_vacc block is used in this design for vector accumulation. Vector growth is approximately 28 bits each second, so if you wanted a really long accumulation (say a few hours), you'd have to use a block such as the qdr_vacc or dram_vacc. As the name suggests, the simple_bram_vacc is simpler so it is fine for this demo spectrometer. The FFT block outputs 1024 cosine values (odd) and 1024 sine values, making 2048 values in total. We have two of these bram vacc's in the design, one for the odd and one for the even frequency channels. The vector length is thus set to 1024 on both.

PARAMETERS

INPUTS/OUTPUTS

Even and Odd BRAMs



Block Parameters: even [X]

Subsystem (mask)

Parameters

Output Data Type: Unsigned

Address width: 10

Data Width: 64

☒ Register Primitive Output

☒ Register Core Output

Optimization: Minimum_Area

Data Binary Point: 0

Initial values (simulation only): [0:2^10-1]

Sample rate: 1

[OK] [Cancel] [Help] [Apply]

The final blocks, odd and even, are shared BRAMs, which we will read out the values of using the tut3.py script.

PARAMETERS

INPUTS/OUTPUTS

Software Registers

There are a few **control registers**, led blinkers, and **snap** block dotted around the design too:

- **cnt_rst**: Counter reset control. Pulse this high to reset all counters back to zero.
- **acc_len**: Sets the accumulation length. Have a look in tut3.py for usage.
- **sync_cnt**: Sync pulse counter. Counts the number of sync pulses issued. Can be used to figure out board uptime and confirm that your design is being clocked correctly.
- **acc_cnt**: Accumulation counter. Keeps track of how many accumulations have been done.

- **led0_sync**: Back on topic: the led0_sync light flashes each time a sync pulse is generated. It lets you know your ROACH is alive.
- **led1_new_acc**: This lights up led1 each time a new accumulation is triggered.
- **led2_acc_clip**: This lights up led2 whenever clipping is detected.

There are also some [snap](#) blocks, which capture data from the FPGA fabric and makes it accessible to the Power PC. This tutorial doesn't go into these blocks (in its current revision, at least), but if you have the inclination, have a look at their [documentation](#). In this design, the snap blocks are placed such that they can give useful debugging information. You can probe these through [KATCP](#), as done in [Tutorial 1](#), if interested. If you've made it to here, congratulations, go and get yourself a cup of tea and a biscuit, then come back for part two, which explains the second part of the tutorial – actually getting the spectrometer running, and having a look at some spectra.

Configuration and Control

Hardware Configuration

The tutorial comes with a pre-compiled fpg file, which is generated from the model you just went through (snap_tut_spec.fpg) Load up your SNAP. You don't need to telnet in to the SNAP; all communication and configuration will be done by the python control script called snap_tut_spec.py.

Next, you need to set up your SNAP. Switch it on, making sure that:

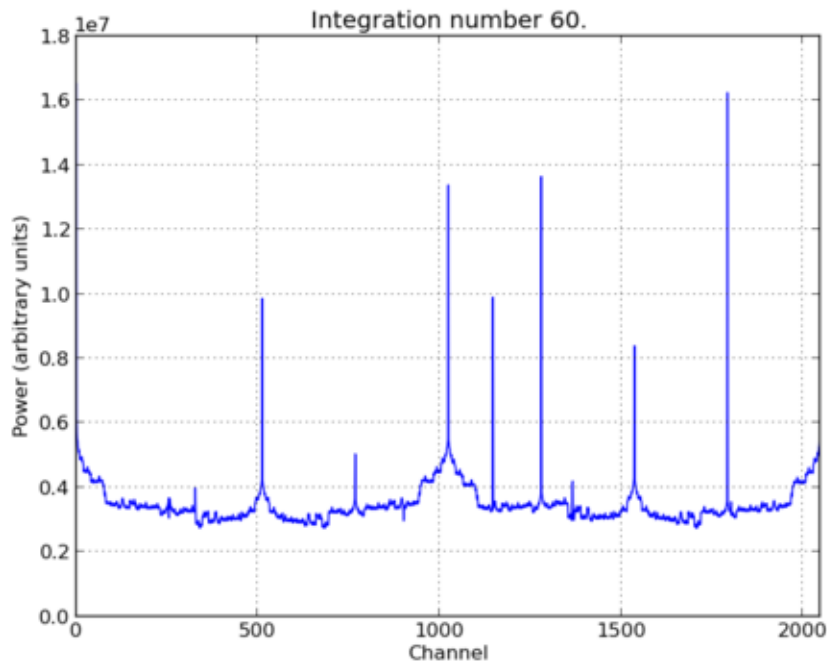
- You have some signal on your ADC input.
- You have your 10MHz reference clock connected to the appropriate SNAP input.

The snap_tut_spec.py spectrometer script

Once you've got that done, it's time to run the script. If you're in linux, browse to where the snap_tut_spec.py file is in a terminal and at the prompt type

```
./snap_tut_spec.py <SNAP IP or hostname> -b <fpgfile name>
```

replacing with the IP address of your SNAP and with your fpgfile. You should see a spectrum like this:



In the plot, there should be a fixed DC offset spike; and if you're putting in a tone, you should also see a spike at the correct input frequency. If you'd like to take a closer look, click the icon that is below your plot and third from the right, then select a section you'd like to zoom in to. The digital gain (-g option) is set to maximum (0xffff_ffff) by default to observe the ADC noise floor. Reduce the gain (decrease the value (for a -10dBm input 0x100)) when you are feeding the ADC with a tone, as not to saturate the spectrum.

Now you've seen the python script running, let's go under the hood and have a look at how the FPGA is programmed and how data is interrogated. To stop the python script running, go back to the terminal and press ctrl + c a few times.

iPython walkthrough

The tut3.py script has quite a few lines of code, which you might find daunting at first. Fear not though, it's all pretty easy. To whet your whistle, let's start off by operating the spectrometer through iPython. Open up a terminal and type:

```
ipython
```

and press enter. You'll be transported into the magical world of iPython, where we can do our scripting line by line, similar to MATLAB. Our first command will be to import the python packages we're going to use:

```
import corr,time,numpy,struct,sys,logging,pylab
```

Next, we set a few variables:

```
katcp_port = 7147
```

Which we can then use in FpgaClient() such that we can connect to the ROACH and issue commands to the FPGA:

```
fpga = casperfpga.CasperFpga(snap)
```

We now have an fpga object to play around with. To check if you managed to connect to your ROACH, type:

```
fpga.is_connected()
```

Let's set the bitstream running using the progdev() command:

```
fpga.upload_to_ram_and_program('snap_tut_spec.fpg')
```

Now we need to configure the accumulation length and gain by writing values to their registers. For two seconds and maximum gain: accumulation length, $2 \cdot (2^{28}) / 2048$, or just under 2 seconds:

```
fpga.write_int('acc_len', 2 * (2**28) / 2048)
fpga.write_int('gain', 0xffffffff)
```

Finally, we reset the counters:

```
fpga.write_int('cnt_rst', 1)
fpga.write_int('cnt_rst', 0)
```

To read out the integration number, we use fpga.read_uint():

```
acc_n = fpga.read_uint('acc_cnt')
```

Do this a few times, waiting a few seconds in between. You should be able to see this slowly rising. Now we're ready to plot a spectrum. We want to grab the even and odd registers of our PFB:

```
a_0=struct.unpack('>1024i', fpga.read('even', 1024*4, 0))
a_1=struct.unpack('>1024i', fpga.read('odd', 1024*4, 0))
```

These need to be interleaved, so we can plot the spectrum. We can use a for loop to do this:

```
interleave_a=[]

for i in range(1024):
    interleave_a.append(a_0[i])
    interleave_a.append(a_1[i])
```

This gives us a 2048 channel spectrum. Finally, we can plot the spectrum using pyLab:

```
pylab.figure(num=1, figsize=(10, 10))
pylab.plot(interleave_a)
pylab.title('Integration number %i.' % acc_n)
pylab.ylabel('Power (arbitrary units)')
pylab.grid()
pylab.xlabel('Channel')
pylab.xlim(0, 2048)
pylab.show()
```

Voila! You have successfully controlled the SNAP spectrometer using python, and plotted a spectrum. Bravo! You should now have enough of an idea of what's going on to tackle the python script. Type exit() to quit ipython. snap_tut_spec.py notes ==

Now you're ready to have a closer look at the snap_tut_spec.py script. Open it with your favorite editor. Again, line by line is the only way to fully understand it, but to give you a head start, here's a few notes:

Connecting to the SNAP

To make a connection to the SNAP, we need to know what port to connect to, and the IP address or hostname of our SNAP. The connection is made on line 73:

```
fpga = casperfpga.CasperFpga(snap)
```

The `katcp_port` variable is set on line 13, and the `roach` variable is passed to the script at the terminal (remember that you typed `python snap_tut_spec.py roachname`). We can check if the connection worked by using `fpga.is_connected()`, which returns `true` or `false`:

```
if fpga.is_connected():
```

The next step is to get the right bitstream programmed onto the FPGA fabric. The bitstream is set on line 68, from the options you entered on cli:

```
bitstream = opts.fpgfile
```

Then the `progdev` command is issued on line 108:

```
fpga.upload_to_ram_and_program(bitstream)
```

Passing variables to the script

Starting from line 51, you'll see the following code:

```
from optparse import OptionParser

p = OptionParser()
p.set_usage('spectrometer.py <ROACH_HOSTNAME_or_IP> [options]')
p.set_description(__doc__)
p.add_option('-l', '--acc_len', dest='acc_len', type='int', default=2*(2**28)/2048,
             help='Set the number of vectors to accumulate between dumps. default is 2*(2^28)/\n2048, or just under 2 seconds.')
p.add_option('-s', '--skip', dest='skip', action='store_true',
             help='Skip reprogramming the FPGA and configuring EQ.')
p.add_option('-b', '--fpg', dest='fpgfile', type='str', default='',
             help='Specify the fpg file to load')
opts, args = p.parse_args(sys.argv[1:])

if args==[]:
    print 'Please specify a SNAP board. Run with the -h flag to see all options.'
    exit()
else:
    snap = args[0]
if opts.fpgfile != '':
    bitstream = opts.fpgfile
```

What this code does is set up some default parameters which we can pass to the script from the command line. If the flags aren't present, it will default to the values set here.

Conclusion

If you have followed this tutorial faithfully, you should now know:

- What a spectrometer is and what the important parameters for astronomy are.
- Which CASPER blocks you might want to use to make a spectrometer, and how to connect them up in Simulink.
- How to connect to and control a SNAP spectrometer using python scripting.

In the following tutorials, you will learn to build a correlator, and a polyphase filtering spectrometer using an FPGA in conjunction with a Graphics Processing Unit (GPU).

1.1.4 Tutorial 4: Wideband Pocket Correlator

Introduction

In this tutorial, you will create a simple Simulink design which uses the [iADC](#) board on [ROACH](#) and the CASPER DSP blockset to process a wideband (400MHz) signal, channelize it and output the visibilities through ROACH's PPC.

By this stage, it is expected that you have completed [tutorial 1](#) and [tutorial 2](#) and are reasonably comfortable with Simulink and basic Python. We will focus here on higher-level design concepts, and will provide you with low-level detail preimplemented.

Background

Some of this design is similar to that of the previous tutorial, the Wideband Spectrometer. So completion of [tutorial 3](#) is recommended.

Interferometry

In order to improve sensitivity and resolution, telescopes require a large collection area. Instead of using a single, large dish which is expensive to construct and complicated to maneuver, modern radio telescopes use interferometric arrays of smaller dishes (or other antennas). Interferometric arrays allow high resolution to be obtained, whilst still only requiring small individual collecting elements.

Correlation

Interferometric arrays require the relative phases of antennas' signals to be measured. These can then be used to construct an image of the sky. This process is called correlation and involves multiplying signals from all possible antenna pairings in an array. For example, if we have 3 antennas, A, B and C, we need to perform correlation across each pair, AB, AC and BC. We also need to do auto-correlations, which will give us the power in each signal. ie AA, BB, CC. We will see this implemented later. The complexity of this calculation scales with the number of antennas squared. Furthermore, it is a difficult signal routing problem since every antenna must be able to exchange data with every other antenna.

Polarization

Dish type receivers are typically dual polarized (horizontal and vertical feeds). Each polarization is fed into separate ADC inputs. When correlating these antennae, we differentiate between full Stokes correlation or a half Stokes method. A full Stokes correlator does cross correlation between the different polarizations (ie for a given two antennas, A and B, it multiplies the horizontal feed from A with the vertical feed from B and vice-versa). A half stokes correlator only correlates like polarizations with each other, thereby halving the compute requirements.

The Correlator

The correlator we will be designing is a 3 input correlator which uses a SNAP board with each ADC operating in maximum speed mode.

Creating Your Design

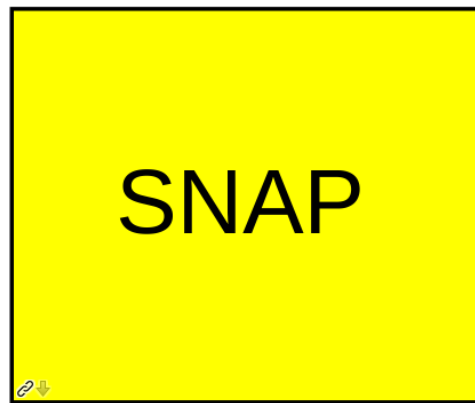
Create a new model

Having started Matlab, open Simulink (either by typing `simulink` on the Matlab command line, or by clicking the Simulink icon in the taskbar). Create a new model and add the Xilinx System Generator and SNAP platform blocks as before in Tutorial 1.

System Generator and Platform Blocks



System
Generator

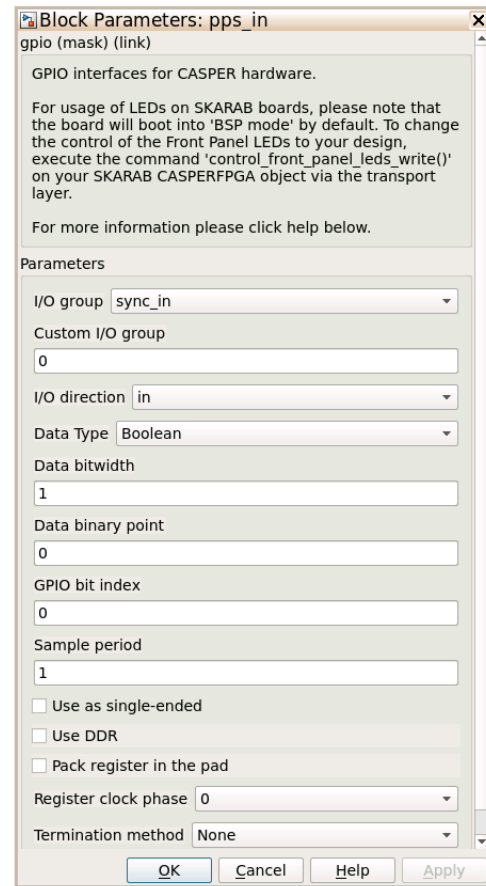
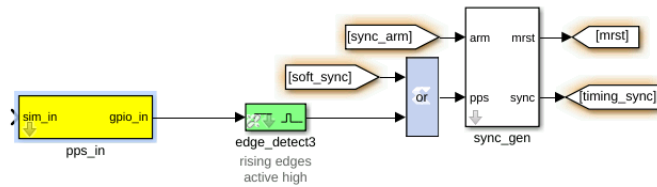


SNAP

By now you should have used these blocks a number of times. Pull the System Generator block into your design from the Xilinx Blockset menu under Basic Elements. The settings can be left on default.

The SNAP platform block can be found under the CASPER XPS System Blockset: Platform subsystem. Set the Clock Source to `adc0_clk` and the rest of the configuration as the default.

Sync Generator

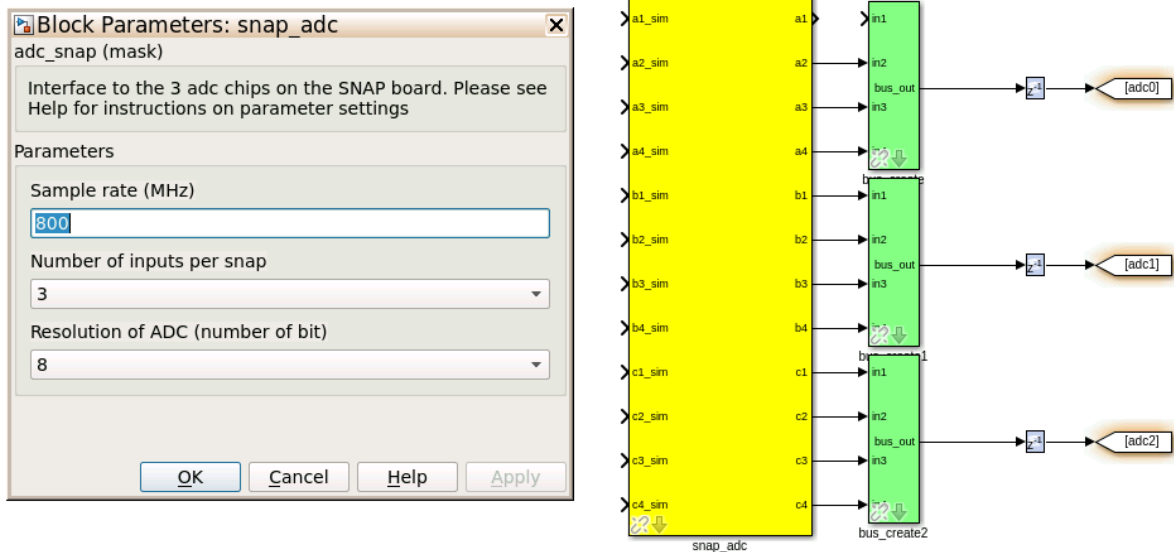


The Sync Generator puts out a sync pulse which is used to synchronize the blocks in the design. See the CASPER memo on sync pulse generation for a detailed explanation.

This sync generator is able to synchronize with an external trigger input. Typically we connect this to a GPS's 1pps output to allow the system to reset on a second boundary after a software arm. This enables us to know precisely the time at which an accumulation was started. It also allows multiple boards to be synchronized which is vital if we are using a signal which correlates digitizers hosted on separate boards. To synchronize from an external PPS we can drive the sync generator logic with the SNAP's sync_in GPIO input.

Logic is also provided to generate a sync manually via a software input. This allows the design to be used even in the absence of a 1 pps signal. However, in this case, the time the sync pulse occurs depends on the latency of software issuing the sync command and the FPGA signal triggering. This introduces some uncertainty in the timestamps associated with the correlator outputs.

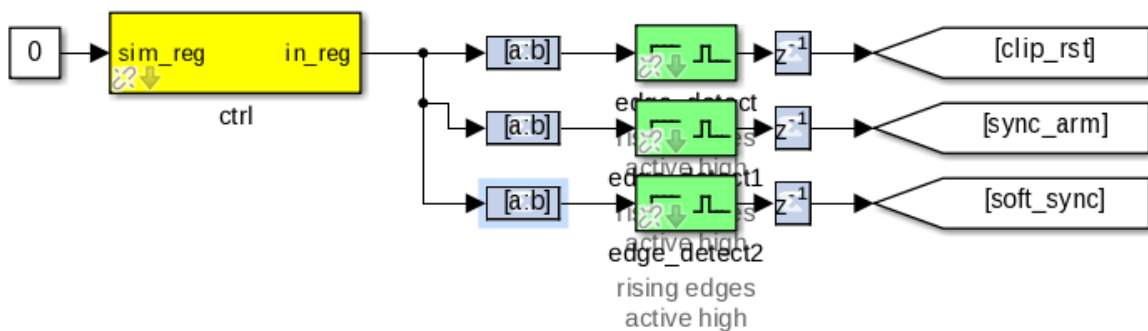
ADCs



Connection of the ADCs is as in tutorial 3 except now we are using all three available inputs.

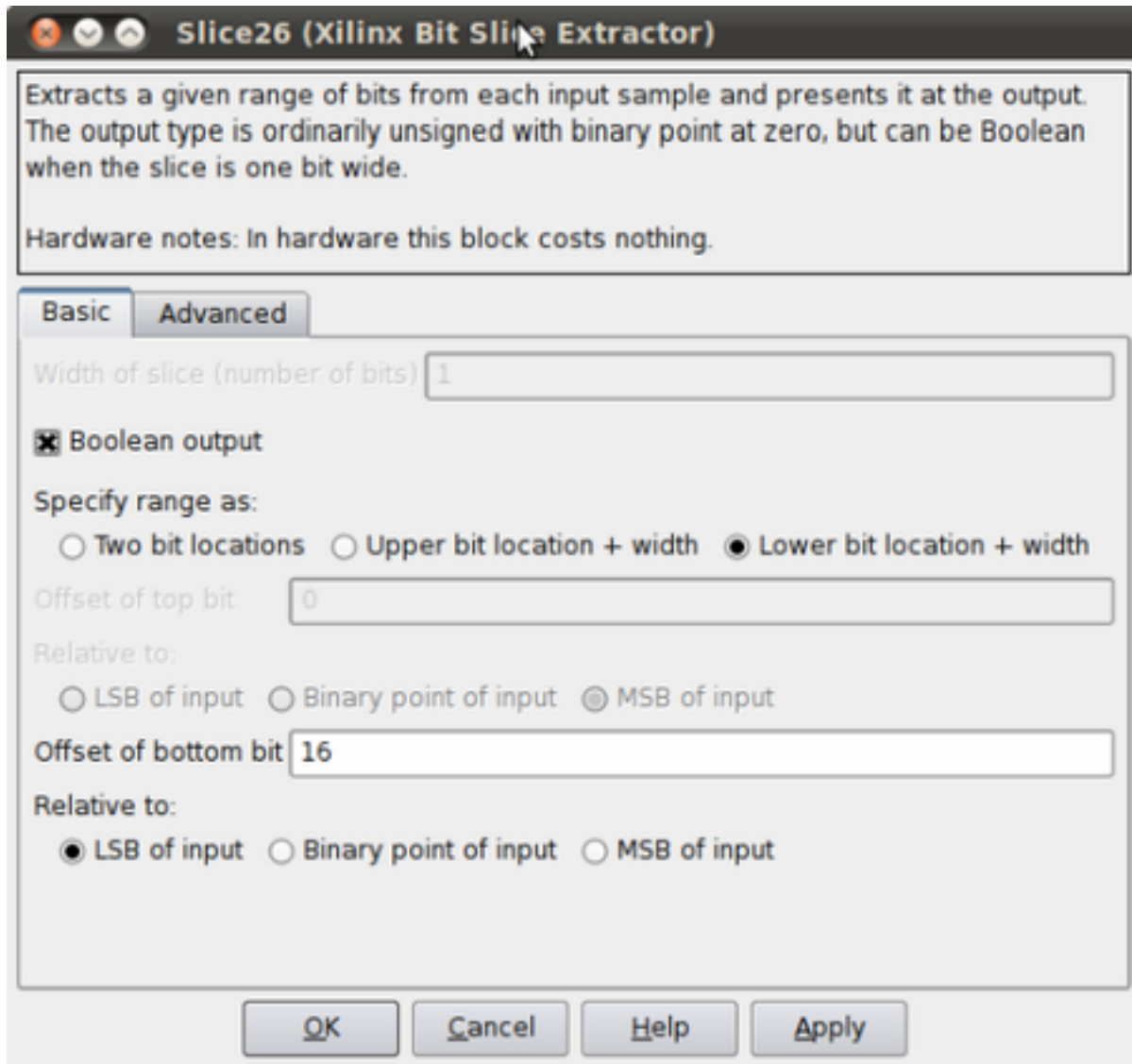
Throughout this design, we use CASPER's bus_create and bus_expand blocks to simplify routing and make the design easier to follow.

Control Register



This part of the Simulink design sets up a software register which can be configured in software to control the correlator. Set the yellow software register's IO direction as from processor. You can find it in the CASPER_XPS System blockset. The constant block input to this register is used only for simulation.

The output of the software register goes to three slice blocks, which will pull out the individual parameters for use with configuration. The first slice block (top) is setup as follows:



The slice block can be found under the Xilinx Blockset → Control Logic. The only change with the subsequent slice blocks is the Offset of the bottom bit. They are, from top to bottom, respectively, 16, 17 & 18.

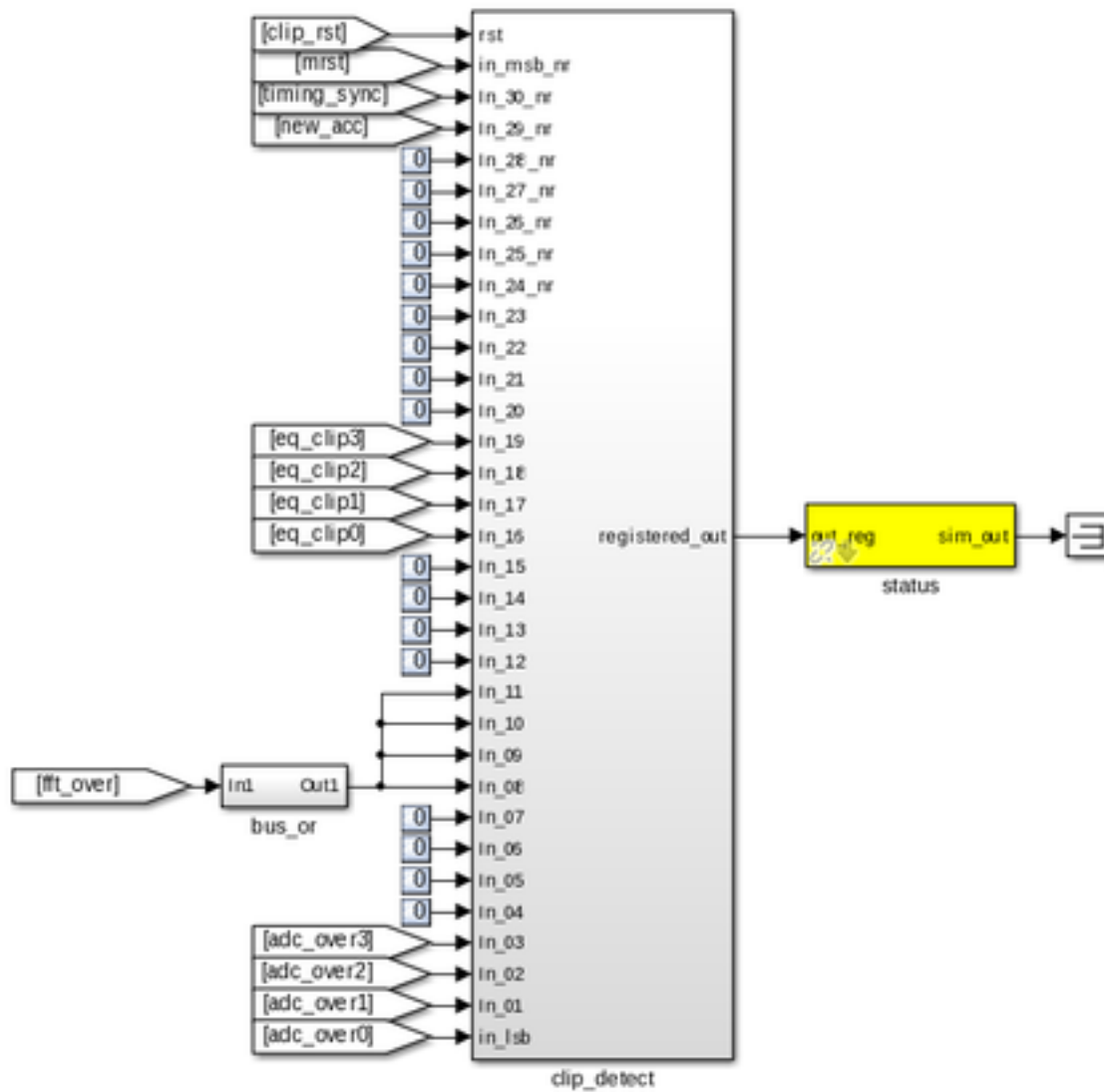
After each slice block we put an edge_detect block, this outputs true if a boolean input signal is true this clock and was false last clock. Found under CASPER DSP Blockset → Misc.

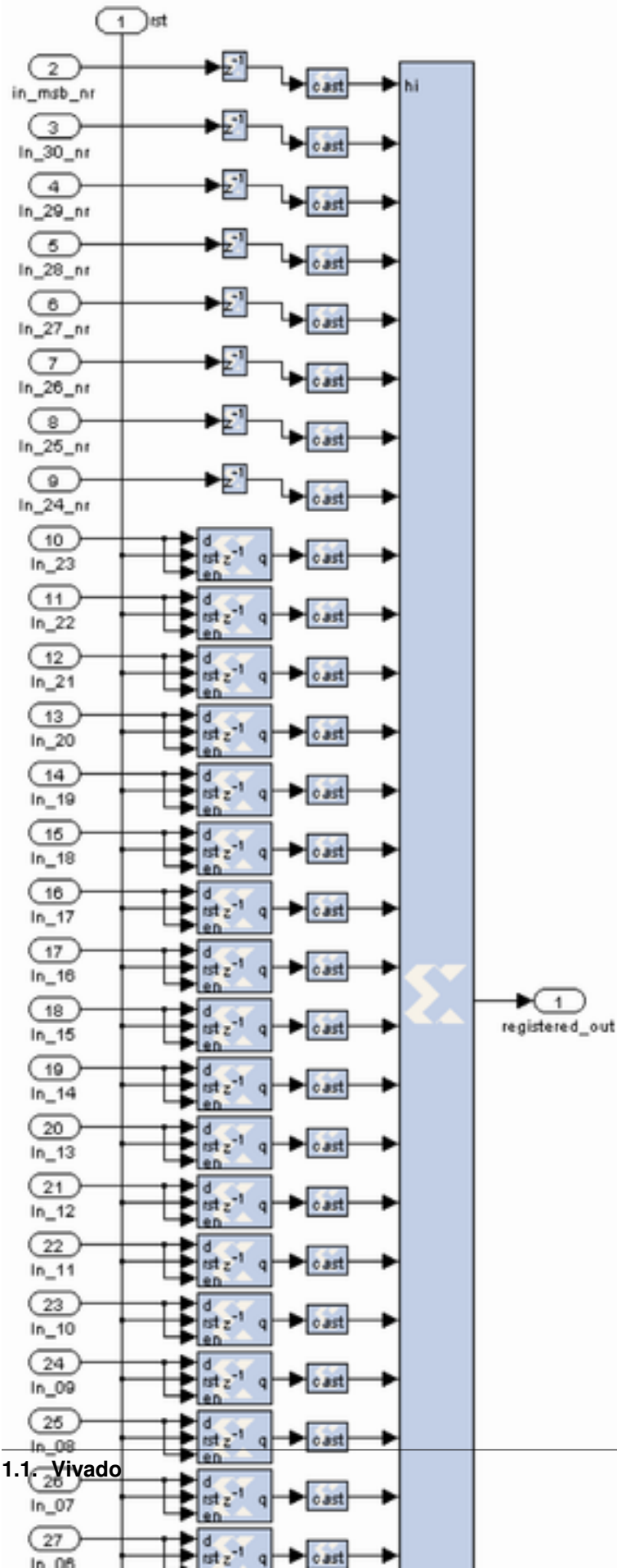
Next are the delay blocks. They can be left with their default settings and can be found under Xilinx Blockset → Common. The delays used here aren't necessary for the function of the design, but can help meet timing by giving the compiler an extra cycle of latency to use when routing control signals.

The Goto and From blocks can be found under Simulink → Signal Routing. Label them as in the block diagram above.

Clip Detect and status reporting

To detect and report signal saturation (clipping) to software, we will create a subsystem with latching inputs.





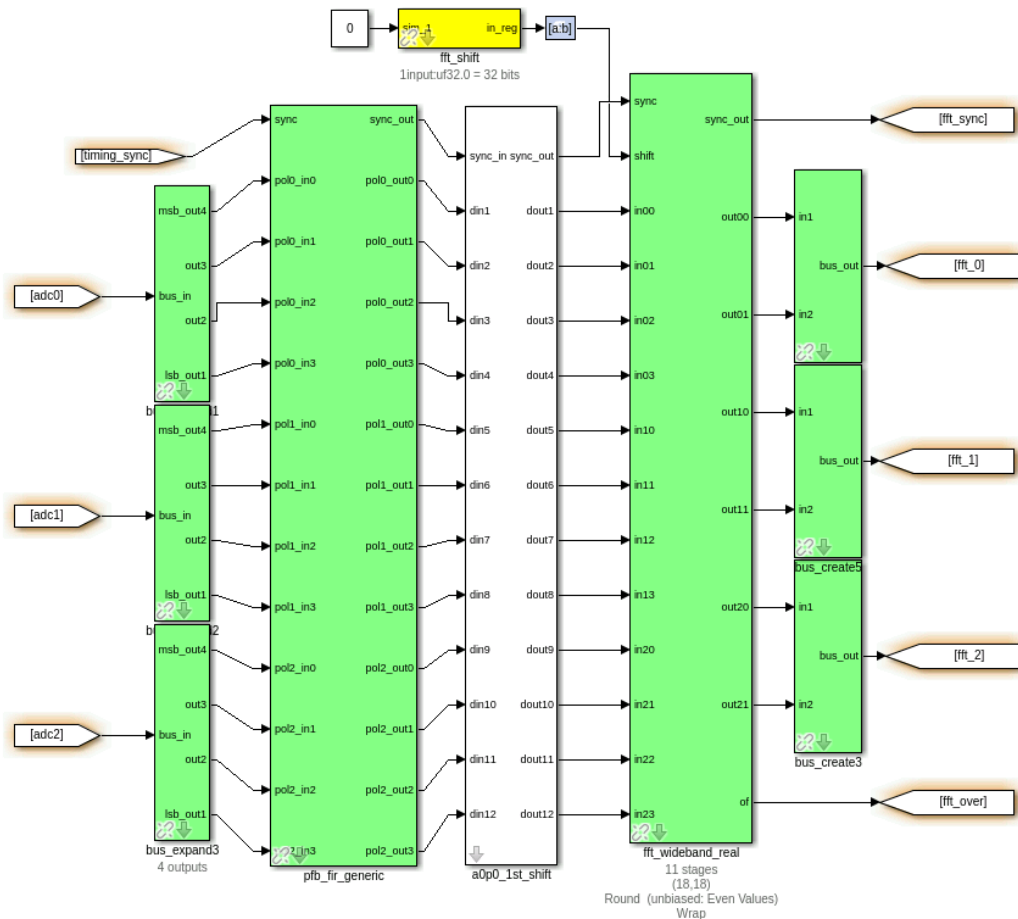
The internals of this subsystem (right) consist of delay blocks, registers and cast blocks.

The delays (inputs 2 - 9) can be kept as default. Cast blocks are required as only unsigned integers can be concatenated. Set their parameters to Unsigned, 1 bit, 0 binary points Truncated Quantization, Wrapped Overflow and 0 Latency.

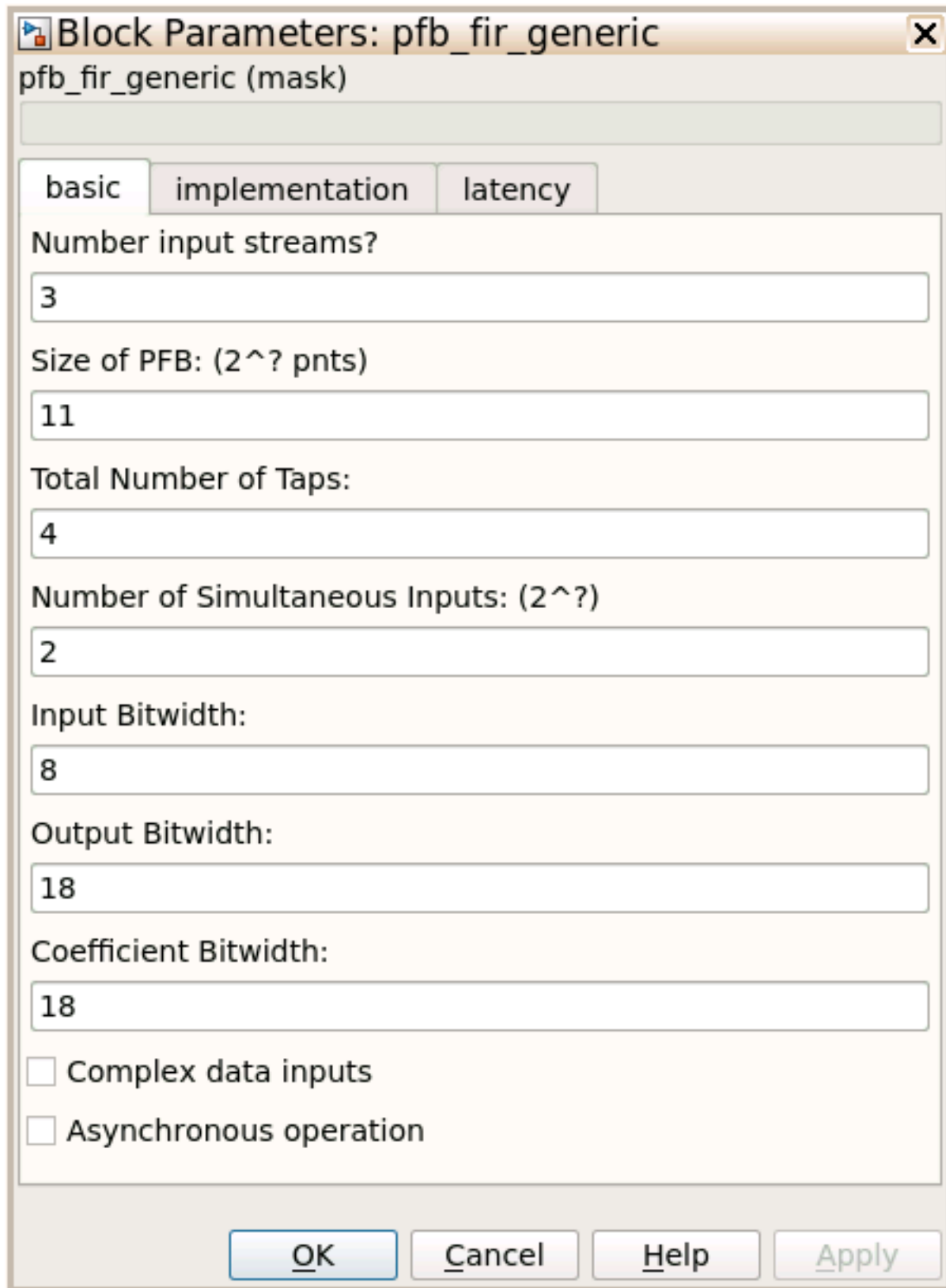
The Registers (inputs 10 - 33) must be set up with an initial value of 0 and with enable and reset ports enabled. The status register on the output of the clip detect is set to processor in with unsigned data type and 0 binary point with a sample period of 1.

PFBs, FFTs and Quantisers

The PFB FIR, FFT and the Quantizer are the heart of this design, there is one set of each for the 3 ADC channels. However, in order to save resources associated with control logic and PFB and FFT coefficient storage, the independent filters are combined into a single simulink block. This is configured to process three independent data streams by setting the “number of inputs” parameter on the PFB_FIR and FFT blocks to 3.



Configure the PFB_FIR_generic blocks as shown below:



A screenshot of a software dialog box titled "Block Parameters: pfb_fir_generic". The dialog has a close button (X) in the top right corner. Below the title bar, the text "pfb_fir_generic (mask)" is displayed. There are three tabs: "basic" (selected), "implementation", and "latency". The "basic" tab contains several input fields and checkboxes. The inputs are: "Number input streams?" with value 3, "Size of PFB: (2^? pnts)" with value 11, "Total Number of Taps:" with value 4, "Number of Simultaneous Inputs: (2^?)" with value 2, "Input Bitwidth:" with value 8, "Output Bitwidth:" with value 18, and "Coefficient Bitwidth:" with value 18. At the bottom of the tab are two checkboxes: "Complex data inputs" and "Asynchronous operation", both of which are unchecked. At the very bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

Block Parameters: pfb_fir_generic

pfb_fir_generic (mask)

basic implementation latency

Number input streams?

3

Size of PFB: (2^? pnts)

11

Total Number of Taps:

4

Number of Simultaneous Inputs: (2^?)

2

Input Bitwidth:

8

Output Bitwidth:

18

Coefficient Bitwidth:

18

☐ Complex data inputs

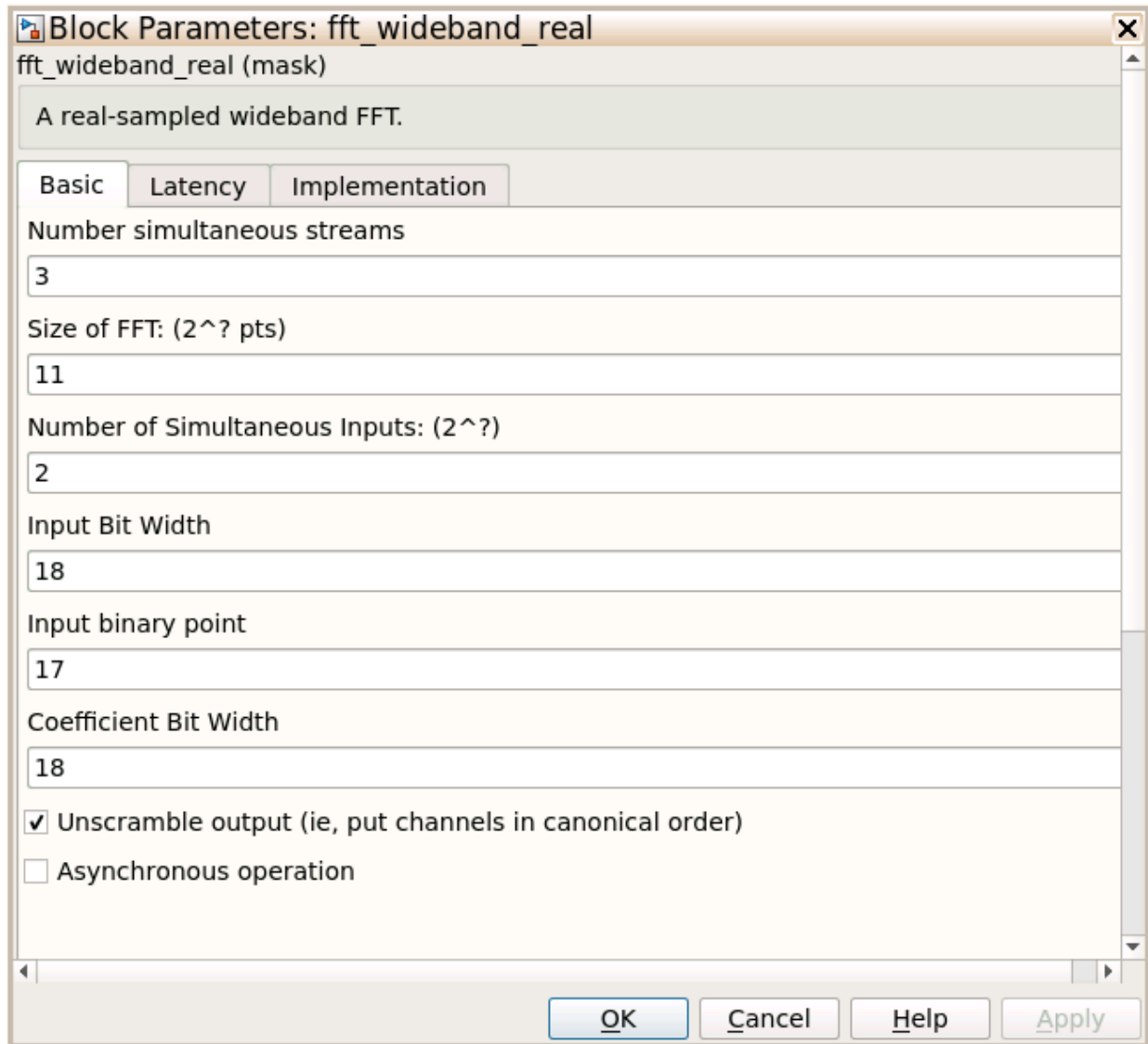
☐ Asynchronous operation

OK Cancel Help Apply

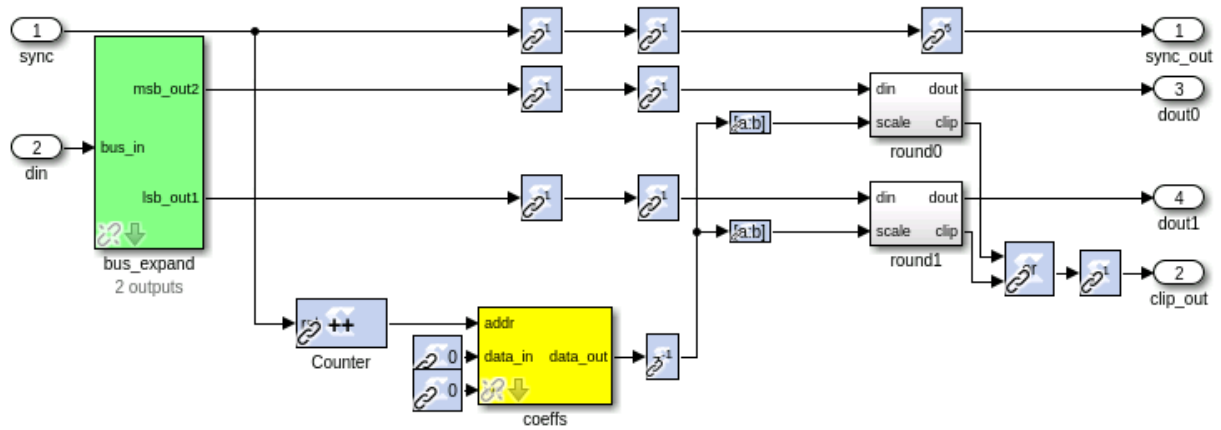
There is potential to overflow the first FFT stage if the input is periodic or signal levels are high as shifting inside the

FFT is only performed after each butterfly stage calculation. For this reason, we recommend casting any inputs up to 18 bits with the binary point at position 17 (thus keeping the range of values -1 to 1), and then downshifting by 1 bit to place the signal in one less than the most significant bits.

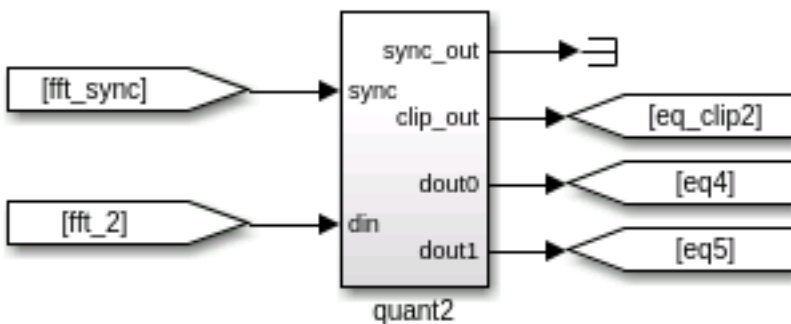
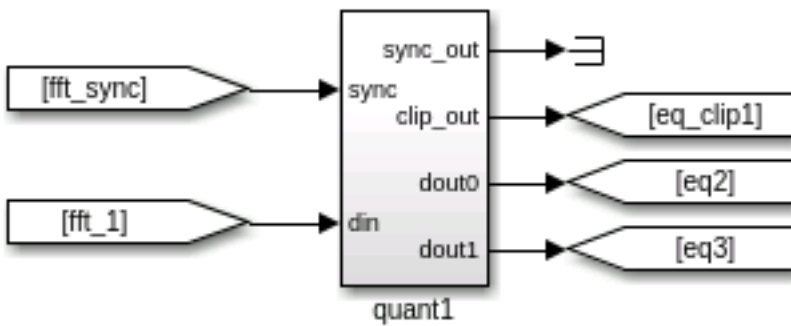
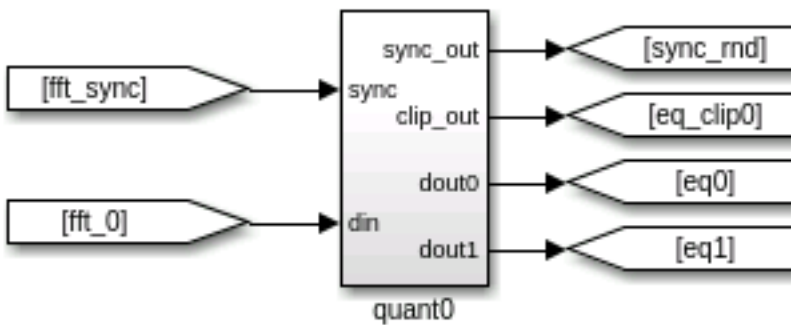
The `fft_wideband_real` block should be configured as follows:



The Quantizer Subsystem is designed as seen below. The quantizer cuts the data signals from the FFT output width (18 bits) down to 4 bits. This means that the downstream processing can be implemented with less resources. In particular, less RAM is needed to store the accumulated correlations. We have to be careful when quantizing signals to make sure that we're not either saturating the quantizer, or suffering from low signal levels. Prior to quantizing we multiply our signals by a runtime programmable set of coefficients, which can be set so as to ensure the quantizer power output levels are optimal.



The top level view of the Quantizer Subsystem is as seen below. We repeat this system once for each signal path.



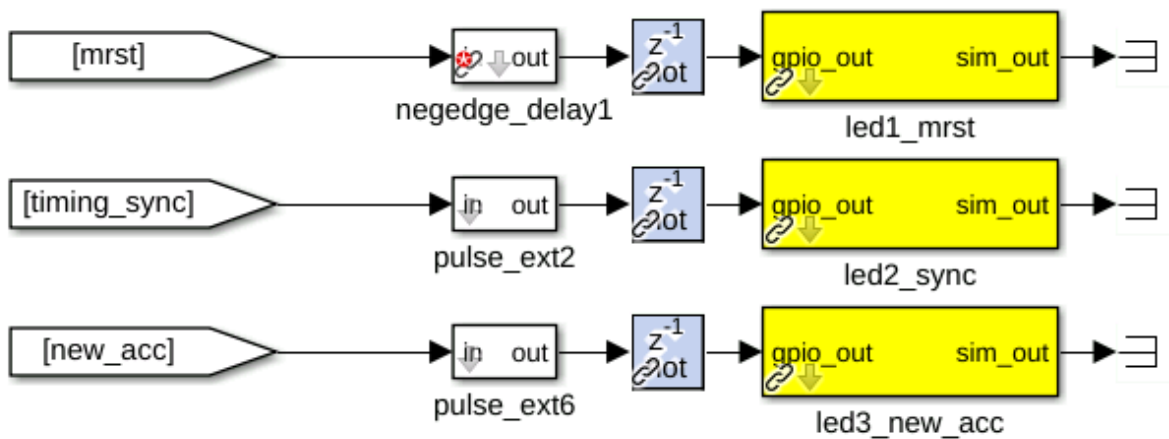
LEDs

The following sections are more periphery to the design and will only be touched on. By now you should be comfortable putting the blocks together and be able to figure out many of the values and parameters. The complete design is available in the tutorials repository for reference.

As a debug and monitoring output we can wire up the LEDs to certain signals. We light an LED with every sync pulse. This is a sort of heartbeat showing that the design is clocking and the FPGA is running.

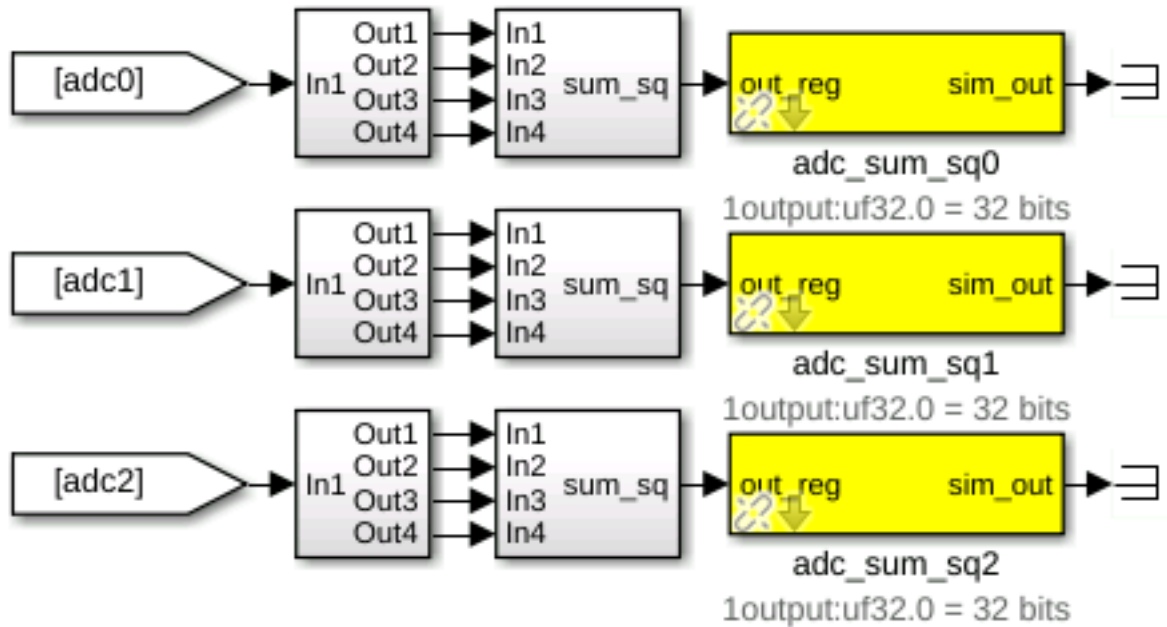
We also use an LED to give a visual indication of when an accumulation is complete.

Since the signals in our design are very low duty cycle, they won't naturally make LED flashes which are visible. We therefore use a pulse extend module to stretch pulses on these signals for 2^{24} FPGA clock cycles, which is about 10 ms.



ADC RMS

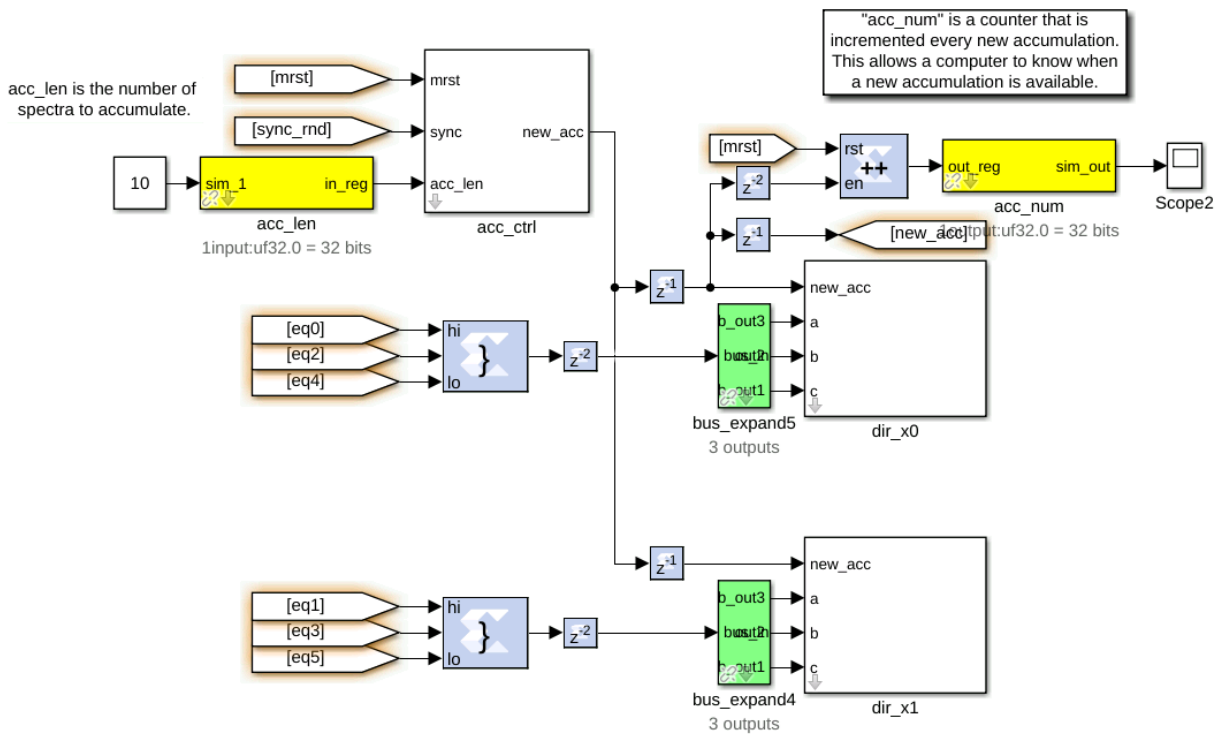
These blocks calculate the RMS values of the ADCs' input signals. We subsample the input stream by a factor of four and do a pseudo random selection of the parallel inputs to prevent false reporting of repetitive signals. This subsampled stream is squared and accumulated for 2^{16} samples.



The MAC operation

The multiply and accumulate is performed in the dir_x (direct-x) blocks, so named because different antenna signal pairs are multiplied directly, in parallel (as opposed to the packetized correlators' X engines which process serially).

Two sets are used, one for the even channels and another for the odd channels. Accumulation for each antenna pair takes place in BRAM using the same simple vector accumulator used in tut3.



CONTROL:

The design starts by itself when the FPGA is programmed. The only control register inputs are for resetting counters and optionally sync'ing to external signal.

Sync LED provides a "heartbeat" signal to instantly see if your design is clocked sensibly.

New accumulation LED gives a visual indication of data rates and dump times.

Accumulation counter provides simple mechanism for checking if a new spectrum output is available. (poll and compare to last value)

Software

The python scripts are located in the `tut_corr` tutorial directory. We first need to run `poco_init.py` to program the FPGA and configure the design. Then we can run either the auto or the cross correlations plotting scripts (`plot_poco_auto.py` and `plot_poco_cross.py`).

Try running these scripts with the `-h` option to get a description of optional and required arguments.

1.1.5 Yellow Block Tutorial: Bidirectional GPIO

This tutorial aims to provide a very entry level introduction to yellow block creation using the JASPER toolflow. A number of other tutorials and guides already exist. For example, the original [ROACH Yellow Block Tutorial](#) in which I based this tutorial from, the [Yellow block EDK wiki page](#), and [Dave George's guide to yellow blocking the KATADC](#). This tutorial attempts to be more of a guided tour around the inner workings of the toolflow, in which you will make an extremely simple new yellow block.

In this tutorial, you will create a yellow block for a bidirectional GPIO n-bit interface for the SNAP board.

Making a Bidirectional GPIO - HDL

So we want to design a bidirectional GPIO interface. That means we need to create a bidirectional GPIO module, and convince the toolflow to instantiate it.

(In most cases when we are porting something into the Toolflow, all verilog/vhdl code is completed, tested, and working in the form of a Xilinx Vivado project)

The simplest version of a bidirectional GPIO module that can be created is simply a wrapper around a Xilinx IOBUF instance. An IOBUF (see the 7 series user guide [page 39](#)) is a Xilinx module used to connect signals to a bi-directional external pin. It has the following ports, which are described (using slightly loose terminology) below:

I: the input (i.e., from the FPGA to the GPIO pin)

O: the output (i.e., from the GPIO pin to the FPGA)

IO: the GPIO pin (defined by the user in the Simulink mask later)

T: The control signal, which configures the interface as an input (i.e. IO \rightarrow O) when T=1, and an output (i.e. I \rightarrow IO) when T=0.

We construct a module “my_gpio_bidir” which wraps ‘n’ number such IOBUF instances (i.e., an n-bit wide buffer) and also registers the output signal. This simple module will form the entirety of the interface we will turn into a yellow block. Create a new folder in /mllib_devel/jasper_library/hdl_sources/ named ‘my_gpio_bidir’ and save your module description as my_gpio_bidir.v there.

NB: n-bit refers to the parameter WIDTH below.

```
module my_gpio_bidir #(parameter WIDTH=1) (
    input          clk,
    inout          [WIDTH-1:0] dio_buf, //inout, NOT input(!)
    input          [WIDTH-1:0] din_i,
    output reg     [WIDTH-1:0] dout_o,
    input          in_not_out_i
);

// A wire for the output data stream
wire [WIDTH-1:0] dout_wire;

// Buffer the in-out data line
IOBUF iob_data[WIDTH-1:0] (
    .O (dout_wire), //the data output
    .IO(dio_buf),   //the external in-out signal
    .I(din_i),      //the data input
    .T(in_not_out_i) //The control signal. 1 for input, 0 for output
);

//register the data output
always @(posedge clk) begin
    dout_o <= dout_wire;
end
endmodule
```

Making a Bidirectional GPIO - Simulink

Start by launching MATLAB via the ./startsg \<your startsg.local file>. Create a new Simulink model titled ‘tut_gpio_bidir.slx’ and save it.

NB: Now is another good time to remind you to save your Simulink model early and save it often! As it is prone to crash at complete random times.

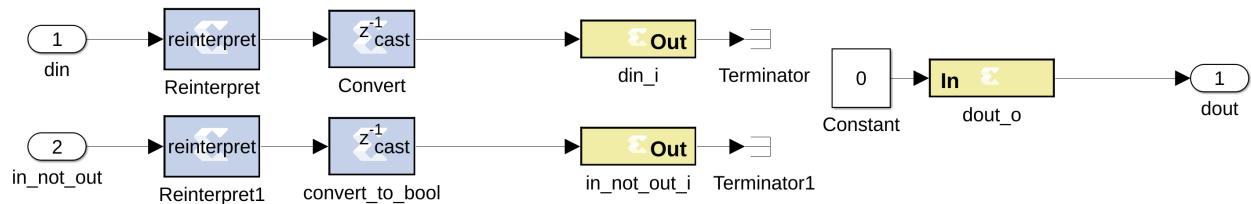
Grab a new yellow block from the XPS library conveniently name 'new_yellow_block'. Rename it 'gpio_bidir_a' as this will serve as our A bank GPIO bidir.

NB: This can be done via the Simulink Library Browser... Or by single clicking anywhere in your model and simply start typing 'new_yellow_block' then pressing enter on the highlighted block shown.

Next, right-click on the yellow block. Navigate to 'Library Link' -> click on 'Disable Link'. Once more, right-click on the yellow block, navigate to 'Library Lock' again, and click 'Break Link'.

Our module has five inputs/outputs. The only ports which need to appear in the yellow block are those that form connections between your module, and the Simulink model. In our case, these are I, O, and T. The clock signal does not come from the Simulink design, but rather some other module. Similarly, the IO signal does not connect to the Simulink model, but rather refers to an external GPIO pin.

Therefore, our yellow block should have 3 ports. An n-bit input (to be connected to I) named 'din', an n-bit output (to be connected to O) named 'dout' and a 1-bit input (to be connected to T) named 'in_not_out'. Be careful – an input to the yellow block requires a "Gateway out" of simulink, since the signal needs to go out from the Simulink module, in to the new GPIO module. Similarly, an output from the yellow block requires a gateway in, since the signal will go out of the GPIO module and in to the Simulink design.



text

The names of the ports should match the names used in your module, and gateways out should be preceded by reinterpret and cast blocks to force the correct data types.

Reinterpret block params:

- Force unsigned arithmetic type
- Force binary point of 0

'din' Convert block params:

- Fixed point, unsigned arithmetic type
- We will take care of the number of bits through the mask, but set binary point to 0
- Truncate and Wrap

'in_not_out' Convert block params:

- Output Type: boolean

Set both Gateway Out blocks to Translate into output port.

Gateway In block params:

- Fixed point, unsigned arithmetic type
- We will take care of the number of bits through the mask, but set binary point to 0
- Truncate and Wrap

Next, you need to tell the toolflow that this is a yellow block, by tagging it as an xps block. Open the block properties(right-click, then select properties), and tag the block by entering xps:<module_name> in the ‘tag’ field. In our case the block is tagged ‘xps:my_gpio_bidir’.

Block Properties: gpio_bidir_a

General Block Annotation Callbacks

Usage

Open Block: Click on the link to open the block.
 Description: Text saved with the block in the model file.
 Priority: Specifies the block's order of execution relative to other blocks in the same model.
 Tag: Text that appears in the block label that Simulink generates.

Open Block: [gpio_bidir_a](#)

Description:

Priority:

Tag:

xps:my_gpio_bidir

OK Cancel Help Apply alt

text

We are now almost finished with Simulink, except for one last modification to the block. As we have seen, the inputs/outputs of the “Simulink” module take the names of the gateway blocks that define them. In order that these names always be unique, the toolflow mandates that they follow a hierarchical naming scheme. This is defined by: <model_name><parent_block_name><user_specified_port_name>. Since Simulink ensures that no two blocks have the same name, this naming scheme always results in a unique port name, no matter how many times you instantiate your yellow block. Each yellow block has an initialization script which (amongst other possible functions) must rename gateways in a block according to this convention.

The ‘my_gpio_bidir’ block’s initialization script is my_gpio_bidir_mask.m, in the xps_library sub-directory of

mllib_devel. It does nothing except find all the gateways in the block (by looking for blocks whose name ends in “<user_specified_port_name>”) and renaming them appropriately. As in all things yellow blocky, If in doubt, copy from another block which works and tweak to your needs. :)

```
% find all the gateway in/out blocks
gateway_outs = find_system(gcb, ...
    'searchdepth', 1, ...
    'FollowLinks', 'on', ...
    'lookundermasks', 'all', ...
    'masktype', 'Xilinx Gateway Out Block');

gateway_ins = find_system(gcb, ...
    'searchdepth', 1, ...
    'FollowLinks', 'on', ...
    'lookundermasks', 'all', ...
    'masktype', 'Xilinx Gateway In Block');

% set number of bits for the Convert block
convert_blkHandle = getSimulinkBlockHandle([gcb, '/Convert']);
set_param(convert_blkHandle, 'n_bits', num2str(bitwidth));
if length(bit_index) ~= bitwidth
    error('Bit index does not have the same number of elements as the I/O bitwidth.
    ↳ When using bitwidths greater than one, you should specify a vector of bit indices_
    ↳ to use.');
```

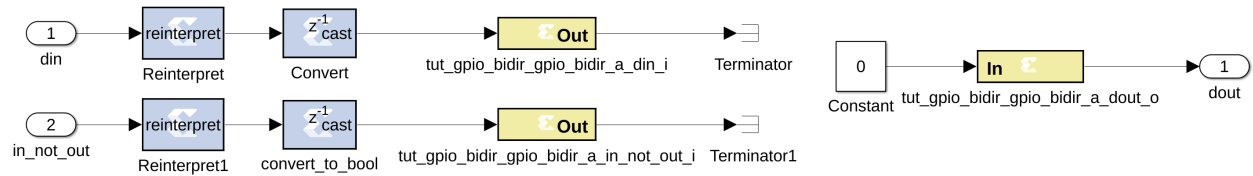
```
end

%rename the gateway outs
for i =1:length(gateway_outs)
    gw = gateway_outs{i};
    gw_name = get_param(gw, 'Name');
    if regexp(gw_name, 'in_not_out_i$')
        set_param(gw, 'Name', clear_name([gcb, '_in_not_out_i']));
    elseif regexp(gw_name, 'din_i$')
        set_param(gw, 'Name', clear_name([gcb, '_din_i']));
    else
        parent_name = get_param(gw, 'Parent');
        error('Unknown gateway: ', parent_name, '/', gw_name);
    end
end

%rename the gateway ins
for i =1:length(gateway_ins)
    gw = gateway_ins{i};
    gw_name = get_param(gw, 'Name');
    % Set number of bits for gateway in block
    set_param(gw, 'n_bits', num2str(bitwidth));
    if regexp(gw_name, 'dout_o$')
        set_param(gw, 'Name', clear_name([gcb, '_dout_o']));
    else
        parent_name = get_param(gw, 'Parent');
        error('Unknown gateway: ', parent_name, '/', gw_name);
    end
end
```

We call the ‘my_gpio_bidir_mask’ script by specifying it in the block’s initialization commands in the block mask. I.e., right click the block, create a mask, and add to the initialization section my_gpio_bidir_mask;. This is exactly the same as the procedure used to call drawing functions for any other (non-yellow) library blocks.

The result from the mask script will be:

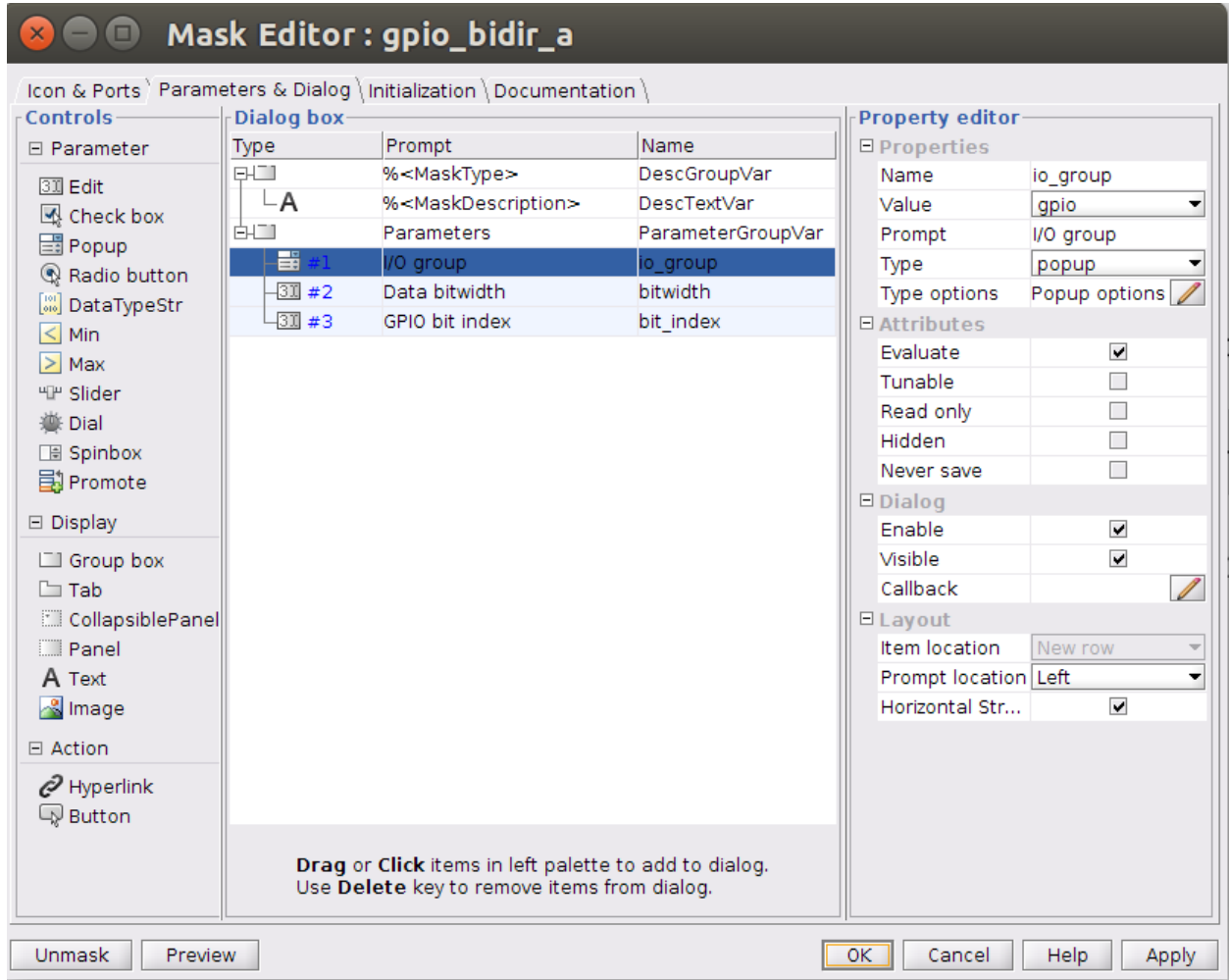


alt

text

Next, we need to add some parameters. Click on the 'Parameters & Dialog' tab, and click-and-drag a popup box over from the left hand side under the 'Parameters' folder. Repeat by adding two edit boxes to the parameters folder.

The configurations are as follows:

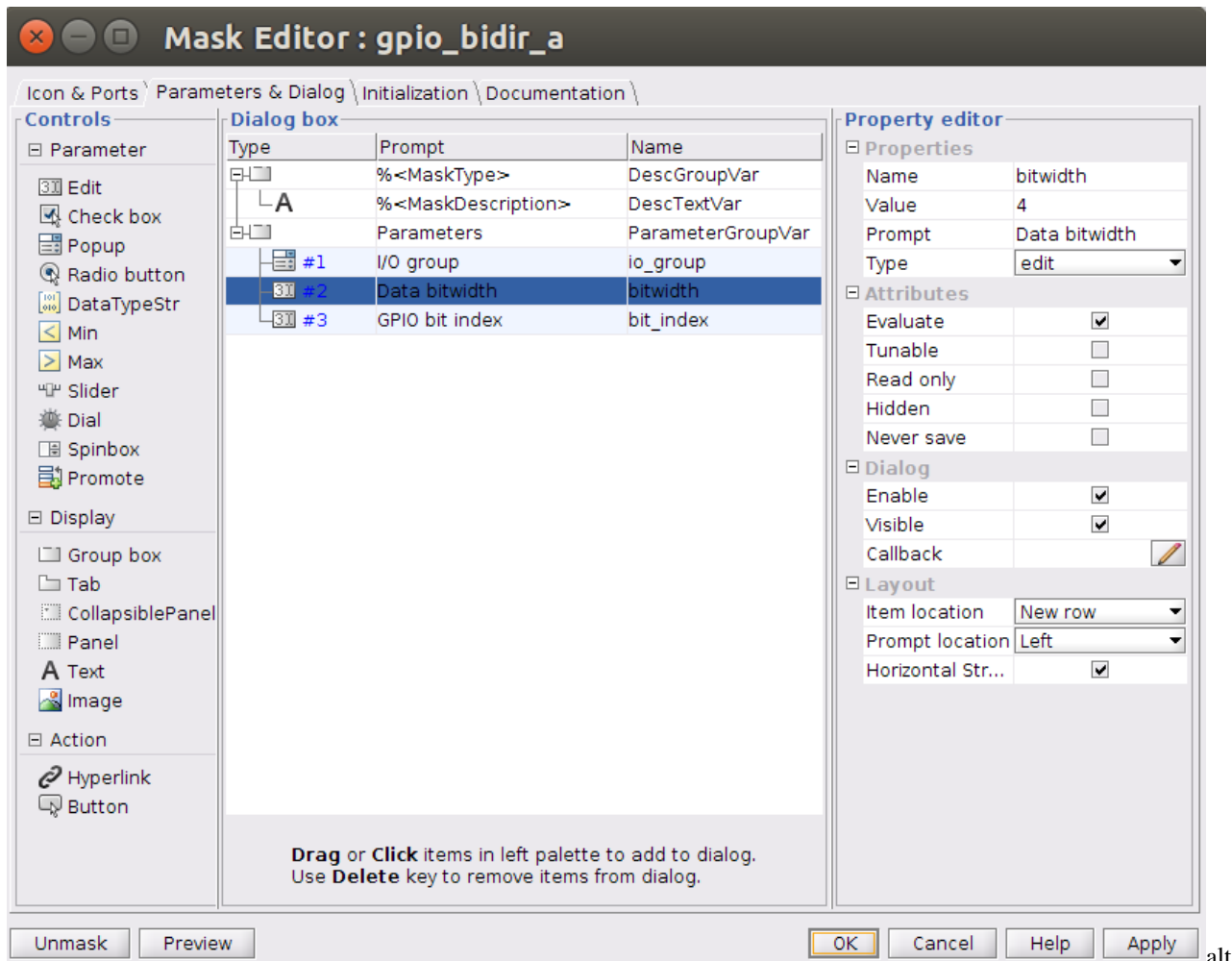


alt

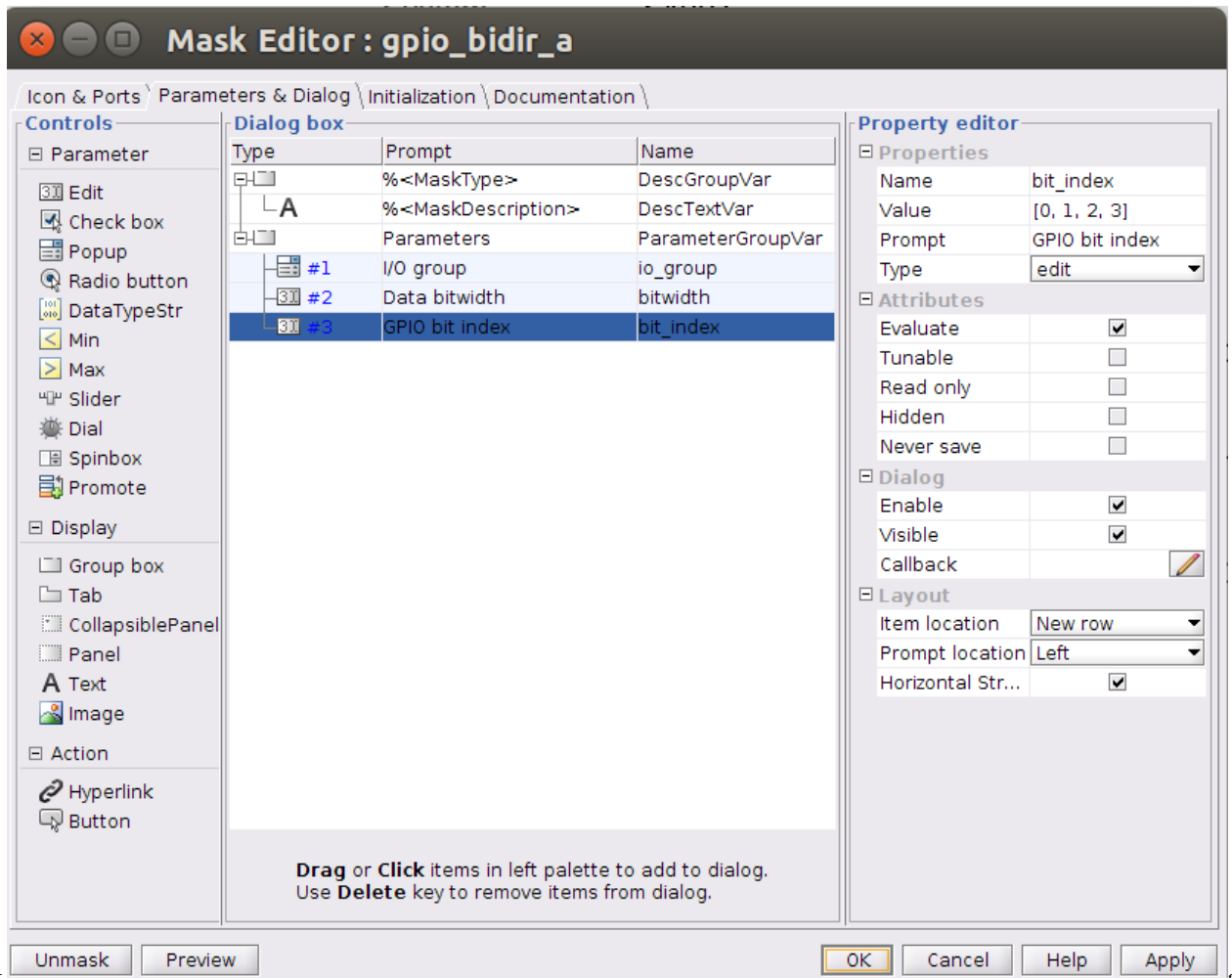
text

Set the 'Type options' parameter for 'iogroup' by clicking on the pencil next to 'Popup options' to (should match the options in the same popup for the GPIO block):

```
led
gpio
sync_in
sync_out
zdok0
zdok1
```



alt



text
text

Next, we continue by referring to the picture below. You should copy and paste the 'gpio_bidir_a' block and rename it as 'gpio_bidir_b', this second block will be used for B bank GPIO's. Be sure to set the I/O group and Data bitwidth for both my_gpio_bidir blocks to 'gpio' and 4. Set the GPIO bit index for gpio_bidir_a to the Matlab vector '[0, 1, 2, 3]' and for gpio_bidir_b to the Matlab vector '[4, 5, 6, 7]'. The length of these vectors must match the value entered for Data bitwidth.

Following, drop in 4 software register blocks, named 'to_gpio_a', 'a_is_input', 'to_gpio_b', and 'b_is_input'. All with the same parameters shown below:

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction From Processor

I/O delay

Initial Value

Sample period

Bitfield names [msb...lsb]

Bitfield widths

Bitfield binary pts

text

Additionally, drop in 2 more software register blocks named 'from_gpio_a' and 'from_gpio_b'. Both with the following parameters:

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction To Processor ▾

I/O delay

Initial Value

Sample period

Bitfield names [msb...lsb]

Bitfield widths

Bitfield binary pts

text

Finally, insert 2 gpio blocks that will be configured as leds, name them 'led_a' and 'led_b'. The first with a 'GPIO bit index' of '0' and the second as '1':

gpio (mask) (link)

GPIO interfaces for CASPER hardware.

For usage of LEDs on SKARAB boards, please note that the board will boot into 'BSP mode' by default. To change the control of the Front Panel LEDs to your design, execute the command 'control_front_panel_leds_write()' on your SKARAB CASPERFPGA object via the transport layer.

For more information please click help below.

Parameters

I/O group

Custom I/O group

I/O direction

Data Type

Data bitwidth

Data binary point

GPIO bit index

Sample period

block). If you don't then something went wrong, and you should re-read this tutorial to see where you differed. If you see the yellow block, please continue on.

Python auto-gen scripts (JASPER Toolflow)

Now we have the module (HDL you wrote first) and Simulink model finished. It is time to write some Python code so that the toolflow will see our yellow block and instantiate the module. When the toolflow runs, it will look for xps-tagged blocks in your design. For each one it will construct an instantiation, connecting your yellow block ports/parameters to the HDL code you wrote. This will all later show up in a top-level auto-generated entity, cleverly called 'top.v'.

The toolflow is as follows, starting with the jasper command in the matlab terminal (all scripts can be found in the mlib_devel directory or yellow_block sub-dir):

```
jasper.m -> jasper_frontend.m -> exec_flow.py -> toolflow.py -> yellow_block.py -> my_gpio_bidir.py
```

The last script will be the name of your module as in this case `my_gpio_bidir.py`. Create this script in the yellow block sub-directory of `mlib_devel`. I recommend carefully reading `yellow_block.py` as the function header comments are well written and explain what you need to do. :)

NB: I figured out how to create this script by comparing `my_gpio_bidir` to the gpio yellow block. First I found the `hdl_source` file for it. Next I compared the `top.v` from a different project that contained the gpio block (as `top.v` contains the instantiation for the gpio module) Then, I compared the script that generated that instantiation (`gpio.py`) to the `top.v` and `hdl` source.

Start by first just tweaking the `modify_top` function to suit your needs, run 'jasper' command and fix python errors until the errors point to the `gen_constraints` function. Next, repeat the process for the `gen_constraints` function. Debug and repeat until you compile w/out errors, look in `top.v` file in the build directory and you should see your yellow block instantiation. Carefully add the rest of your functionality from here until your `top.v` instantiation matches your HDL code (module).

(Move on to the next section, once the 'jasper' command finishes and your yellow block instantiation matches the module)

NB: The system generated verilog/VHDL code is all lowercase, be sure that your ports and signals match accordingly.

The code for the `my_gpio_bidir` yellow block is below (pay particular attention to the comments):

```
from yellow_block import YellowBlock
from constraints import PortConstraint
from helpers import to_int_list

class my_gpio_bidir(YellowBlock):
    def initialize(self):
        # Set bitwidth of block (this is determined by the 'Data bitwidth' parameter,
        ↪in the Simulink mask)
        self.bitwidth = int(self.bitwidth)
        # add the source files, which have the same name as the module (this is the
        ↪verilog module created above)
        self.module = 'my_gpio_bidir'
        self.add_source(self.module)

    def modify_top(self, top):
        # port name to be used for 'dio_buf'
        external_port_name = self.fullname + '_ext'
        # get this instance from 'top.v' or create if not instantiated yet
        inst = top.get_instance(entity=self.module, name=self.fullname, comment=self.
        ↪fullname)
```

(continues on next page)

(continued from previous page)

```

# add ports necessary for instantiation of module
inst.add_port('clk', signal='user_clk', parent_sig=False)
# parent_port=True, and dir='input', so add an input to 'top.v'
inst.add_port('dio_buf', signal=external_port_name, dir='inout', width=self.
->bitwidth, parent_port=True)
inst.add_port('din_i', signal='%s_din_i'%self.fullname, width=self.bitwidth)
inst.add_port('dout_o', signal='%s_dout_o'%self.fullname, width=self.bitwidth)
inst.add_port('in_not_out_i', signal='%s_in_not_out_i'%self.fullname)
# add width parameter from 'Data bitwidth' parameter in Simulink mask
inst.add_parameter('WIDTH', str(self.bitwidth))

def gen_constraints(self):
    # add port constraint to user_const.xdc for 'inout' ()
    return [PortConstraint(self.fullname+'_ext', self.io_group, port_
->index=range(self.bitwidth), iogroup_index=to_int_list(self.bit_index))]

```

Testing

Now we need to test. The python script for this tutorial is an automated testing of the Bidirectional GPIO block we just made. It sets one GPIO bank (a or b) as output, the other as an input. It then writes to one output bank and reads the others input. After which it swaps the modes of each bank in order to demonstrate that each bank can be either an input or output (bidirectional) and repeats the same manner of write/reading.

Run the script included below in the terminal using the command:

```
./tut_gpio_bidir.py -f <Generated fpg file here> <SNAP hostname or ip addr>
```

NB: You may need to run `chmod +x ./tut_gpio_bidir.py` first.

```

#!/usr/bin/env python
'''
Script for testing the Bi-Directional GPIO Yellow Block created for CASPER Tutorial 7.
Author: Tyrone van Balla, January 2016
Reworked for SNAP and tested: Brian Bradford, May 2018
'''
import casperfpga
import time
import sys
import numpy as np

fpgfile = 'tut_gpio_bidir.fpg'
fpgas = []

def exit_clean():
    try:
        for f in fpgas: f.stop()
    except:
        pass
    exit()

def exit_fail():
    print 'FAILURE DETECTED. Exiting . . .'
    exit()

if __name__ == '__main__':

```

(continues on next page)

(continued from previous page)

```

import argparse

parser = argparse.ArgumentParser()
parser.add_argument("snap", help="<SNAP_HOSTNAME or IP>")
parser.add_argument("-f", "--fpgfile", type=str, default=fpgfile, help="Specify_
↳the fpg file to load")
parser.add_argument("-i", "--ipython", action='store_true', help="Enable iPython_
↳control")

args = parser.parse_args()

if args.snap == "":
    print 'Please specify a SNAP board. \nExiting'
    exit()
else:
    snap = args.snap

if args.fpgfile != '':
    fpgfile = args.fpgfile

# try:

print "Connecting to server %s . . ."%(snap),
fpga = casperfpga.CasperFpga(snap)
time.sleep(1)

if fpga.is_connected():
    print 'ok\n'
else:
    print 'ERROR connecting to server %s . . .'%(snap)
    exit_fail()

# program fpga with bitstream

print '-----'
print 'Programming FPGA...',
sys.stdout.flush()
fpga.upload_to_ram_and_program(fpgfile)
time.sleep(1)
print 'ok'

# initialize gpio bank control registers
fpga.write_int('a_is_input', 1)
fpga.write_int('b_is_input', 1)

if args.ipython:
    # open ipython session for manual testing of yellow block

    # list all registers first
    print '\nAvailable Registers:'
    registers = fpga.listdev()
    for reg in registers:
        if not('sys' in reg):
            print '\t',
            print reg
        else:
            pass

```

(continues on next page)

(continued from previous page)

```

print '\n'

# how to use
print 'Use "fpga" as the fpga object\n'

import IPython; IPython.embed()

print 'Exiting . . .'
exit_clean()

'''
Automated testing of Bidirectional GPIO Block.
Sets one GPIO bank as output, other as input.
Writes to output bank, reads input.

Swaps mode of banks to demonstrate either bank can be either input or output.
'''

print '#####'
# Send from GPIO_LED (B) to GPIO_GPIO (A)
print '\nConfiguring to send from GPIO_LED (B) to GPIO_GPIO (A)\n'
fpga.write_int('a_is_input', 1) # GPIO_GPIO as input
fpga.write_int('b_is_input', 0) # GPIO_LED as output

print 'Initial Values: A: %s, B: %s\n' % (np.binary_repr(fpga.read_int('from_gpio_a'),
↳ width=4), np.binary_repr(fpga.read_int('from_gpio_b'), width=4))
print 'Writing 0xF to B . . . \n'

fpga.write_int('to_gpio_a', 0) # dummy data written to GPIO_GPIO
fpga.write_int('to_gpio_b', 0xFFFF) # data written to GPIO_LED
time.sleep(0.01)

print 'A: 0 <----- B: 0xF\n'

from_a = fpga.read_int('from_gpio_a') # read GPIO_GPIO
from_b = fpga.read_int('from_gpio_b') # read GPIO_LED

print 'Readback values: A: %s, B: %s\n' % (np.binary_repr(from_a, width=4), np.binary_
↳ repr(from_b, width=4))

print 'Writing 0x0 to B . . . \n'
print 'A: 0xF <----- B: 0x0\n'

fpga.write_int('to_gpio_a', 0xFFFF) # dummy data written to GPIO_GPIO
fpga.write_int('to_gpio_b', 0x0) # data written to GPIO_LED
time.sleep(0.01)

from_a = fpga.read_int('from_gpio_a') # read GPIO_GPIO
from_b = fpga.read_int('from_gpio_b') # read GPIO_LED

print 'Readback values: A: %s, B: %s\n' % (np.binary_repr(from_a, width=4), np.binary_
↳ repr(from_b, width=4))

print '#####'
# Send from GPIO_GPIO (A) to GPIO_LED (B)
print '\nConfiguring to send from GPIO_GPIO (A) to GPIO_LED (B)\n'
fpga.write_int('a_is_input', 0) # GPIO_GPIO as output

```

(continues on next page)

(continued from previous page)

```

fpga.write_int('b_is_input', 1) # GPIO_LED as input

print 'Initial Values: A: %s, B: %s\n' % (np.binary_repr(fpga.read_int('from_gpio_a'),
↳ width=4), np.binary_repr(fpga.read_int('from_gpio_b'), width=4))
print 'Writing 0x0 to A . . . \n'

fpga.write_int('to_gpio_a', 0) # data written to GPIO_GPIO
fpga.write_int('to_gpio_b', 0xFFFF) # dummy data written to GPIO_LED
time.sleep(0.01)

print 'A: 0 -----> B: 0xF\n'

from_a = fpga.read_int('from_gpio_a') # read GPIO_GPIO
from_b = fpga.read_int('from_gpio_b') # read GPIO_LED

print 'Readback values: A: %s, B: %s\n' % (np.binary_repr(from_a, width=4), np.binary_
↳ repr(from_b, width=4))

print 'Writing 0xF to A . . . \n'

print 'A: 0xF -----> B: 0x0\n'

fpga.write_int('to_gpio_a', 0xFFFF) # data written to GPIO_GPIO
fpga.write_int('to_gpio_b', 0x0) # dummy data written to GPIO_LED
time.sleep(0.01)

from_a = fpga.read_int('from_gpio_a') # read GPIO_GPIO
from_b = fpga.read_int('from_gpio_b') # read GPIO_LED

print 'Readback values: A: %s, B: %s\n' % (np.binary_repr(from_a, width=4), np.binary_
↳ repr(from_b, width=4))

# except KeyboardInterrupt:
#     exit_clean()
# except Exception as inst:
#     exit_fail()

exit_clean()

```

Your results without wiring pins on the SNAP board should look something close to:

```

Connecting to server rpi2-11 . . . ok

-----
Programming FPGA... ok
#####

Configuring to send from GPIO_LED (B) to GPIO_GPIO (A)

Initial Values: A: 0000, B: 0000

Writing 0xF to B . . .

A: 0 <----- B: 0xF

Readback values: A: 0000, B: 1111

```

(continues on next page)

(continued from previous page)

```

Writing 0x0 to B . . .
A: 0xF <----- B: 0x0

Readback values: A: 1111, B: 0000

#####

Configuring to send from GPIO_GPIO (A) to GPIO_LED (B)

Initial Values: A: 1111, B: 0000

Writing 0x0 to A . . .
A: 0 -----> B: 0xF

Readback values: A: 0000, B: 0000

Writing 0xF to A . . .
A: 0xF -----> B: 0x0

Readback values: A: 1111, B: 1111

```

Now we will wire up the pins on the SNAP board correctly. Put the following female jumpers between pins on the J9 GPIO (refer to page 14 of [SNAP Schematic](#) if necessary):

- TEST0 and TEST4
- TEST1 and TEST5
- TEST2 and TEST6
- TEST3 and TEST7

After this, run the same script again. The expected results are:

```

Connecting to server rpi2-11 . . . ok

-----
Programming FPGA... ok
#####

Configuring to send from GPIO_LED (B) to GPIO_GPIO (A)

Initial Values: A: 0000, B: 0000

Writing 0xF to B . . .
A: 0 <----- B: 0xF

Readback values: A: 1111, B: 1111

Writing 0x0 to B . . .
A: 0xF <----- B: 0x0

Readback values: A: 0000, B: 0000

```

(continues on next page)

(continued from previous page)

```
#####  
Configuring to send from GPIO_GPIO (A) to GPIO_LED (B)  
  
Initial Values: A: 1111, B: 1111  
  
Writing 0x0 to A . . .  
A: 0 -----> B: 0xF  
  
Readback values: A: 0000, B: 0000  
  
Writing 0xF to A . . .  
A: 0xF -----> B: 0x0  
  
Readback values: A: 1111, B: 1111
```

If you matched the result above, then congratulations you've successfully created and tested your first yellow block!

If not, start by ensuring your original HDL code was correct to begin with, then debug the yellow block Python script you wrote.

Add yellow block to XPS Library

1. Create a new Simulink model with the name identical to your yellow block name (rename your yellow block if it is an unacceptable model name)
2. Add your yellow block to the model. (This should be the only block in the model)
3. Add your yellow block mask script to 'xps_library' folder if needed.
4. Save your Simulink model in the 'xps_models' folder (please put it in the directory that makes sense, otherwise create a new directory)
5. Launch Matlab via the `./startsg` script in `mllib_devel` directory.
6. Double-click on 'xps_library' directory from the 'Current Folder' pane on the left-hand side of the Matlab window.
7. Run `xps_build_new_library`, click 'Yes' on overwrite dialog prompt and ignore any warnings.
8. For any models you wish to link with this new library, open the model and run `update_casper_blocks(bdroot)` in the Matlab command window. (Preferably all your models)

Now help out CASPER by adding more yellow blocks to our library :)

Author: Brian Bradford, June 1, 2018

Credit to Jack Hickish **for original ROACH yellow block tutorial, in which I based this from.**

SKARAB

1. Introduction Tutorial [Step-by-Step](#) or [Completed](#)
2. 40GbE Tutorial [Step-by-Step](#) or [Completed](#)
3. HMC Tutorial [Step-by-Step](#) or [Completed](#)

4. Spectrometer Tutorial *Step-by-Step* or *Completed*

1.1.6 Tutorial 1: Introduction to Simulink

In this tutorial, you will create a simple Simulink design using both standard Xilinx system generator blockset, as well as library blocks specific to CASPER boards (so-called “Yellow Blocks”). At the end of this tutorial, you will know how to generate an fpg file, program it to a CASPER FPGA board, and interact with your running hardware design using `casperfpga` via a Python Interface.

Creating Your Design

Create a New Model

Start Matlab via executing the `startsg` command, as described [here](#). This ensures that necessary Xilinx and CASPER libraries are loaded into your by Simulink. When MATLAB starts up, open Simulink by typing `simulink` on the MATLAB command line. Start a new model, and save it with an appropriate name. **With Simulink, it is very wise to save early and often.**

There are some Matlab limitations you should be aware-of right from the start:

- **Do not use spaces in your filenames** or anywhere in the file path as it will break the toolflow.
- **Do not use capital letters in your filenames** or anywhere in the file path as it will break the toolflow.
- **Beware block paths that exceed 64 characters.** This refers to not only the file path, but also the path to any block within your design.
 - For example, if you save a model file with a name `~/some_really_long_filename.slx`, and have a block called in a submodule the longest block path would be: `some_really_long_filename_submodule_block`.
 - If you use lots of subsystems, this can cause problems.

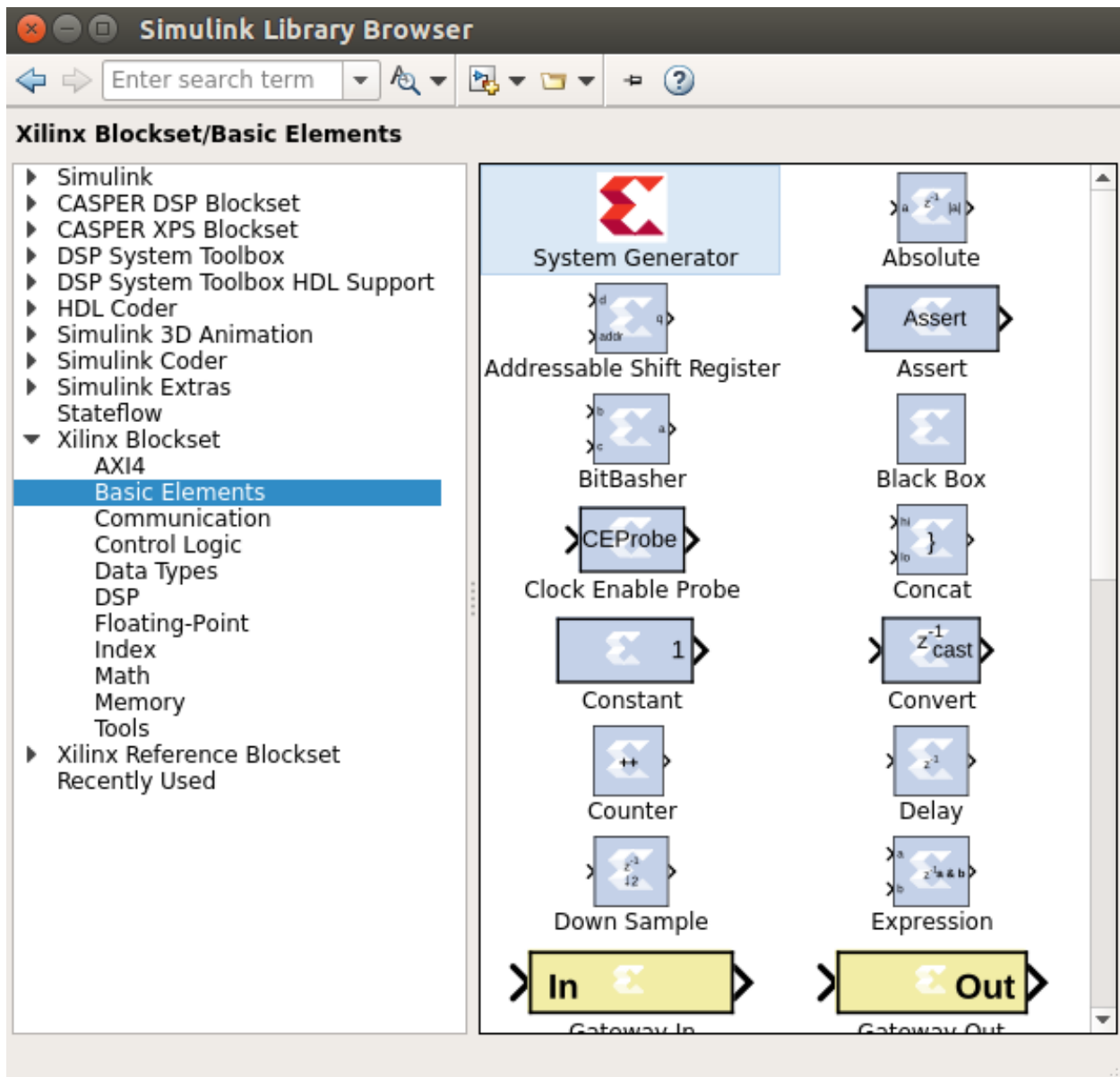
Library organization

There are three libraries which you will use when you design firmware in Simulink.

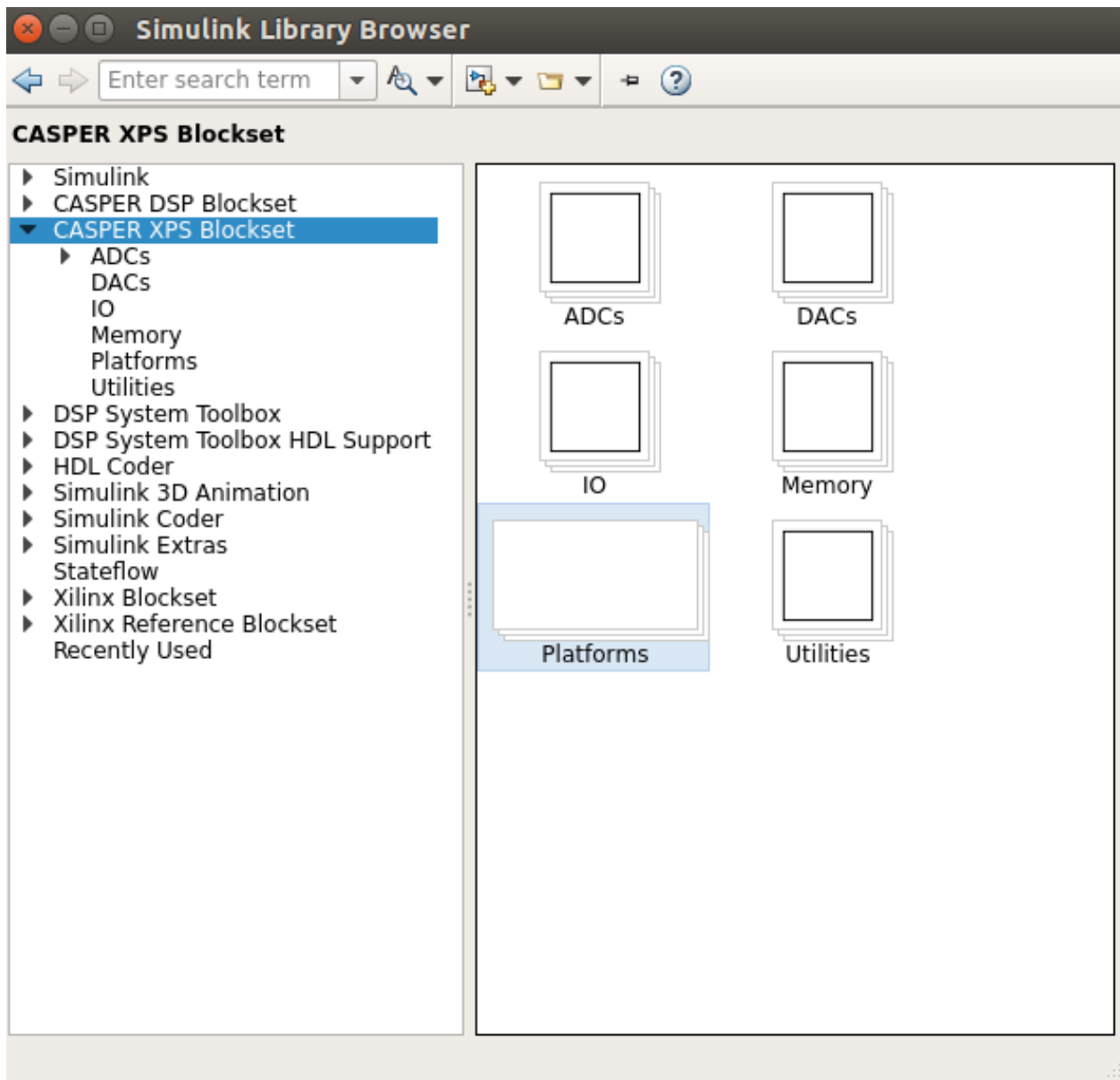
1. The **CASPER XPS Library** contains “Yellow Blocks” – these are blocks which encapsulate interfaces to hardware (ADCs, Memory chips, CPUs, Ethernet ports, etc.)
2. The **CASPER DSP Library** contains (mostly green) blocks which implement DSP functions such as filters, FFTs, etc.
3. The **Xilinx Library** contains blue blocks which provide low-level functionality such as multiplexing, delaying, adding, etc. The Xilinx library also contains the super-special System Generator block, which contains information about the type of FPGA you are targeting.

Add Xilinx System Generator and XSG core config blocks

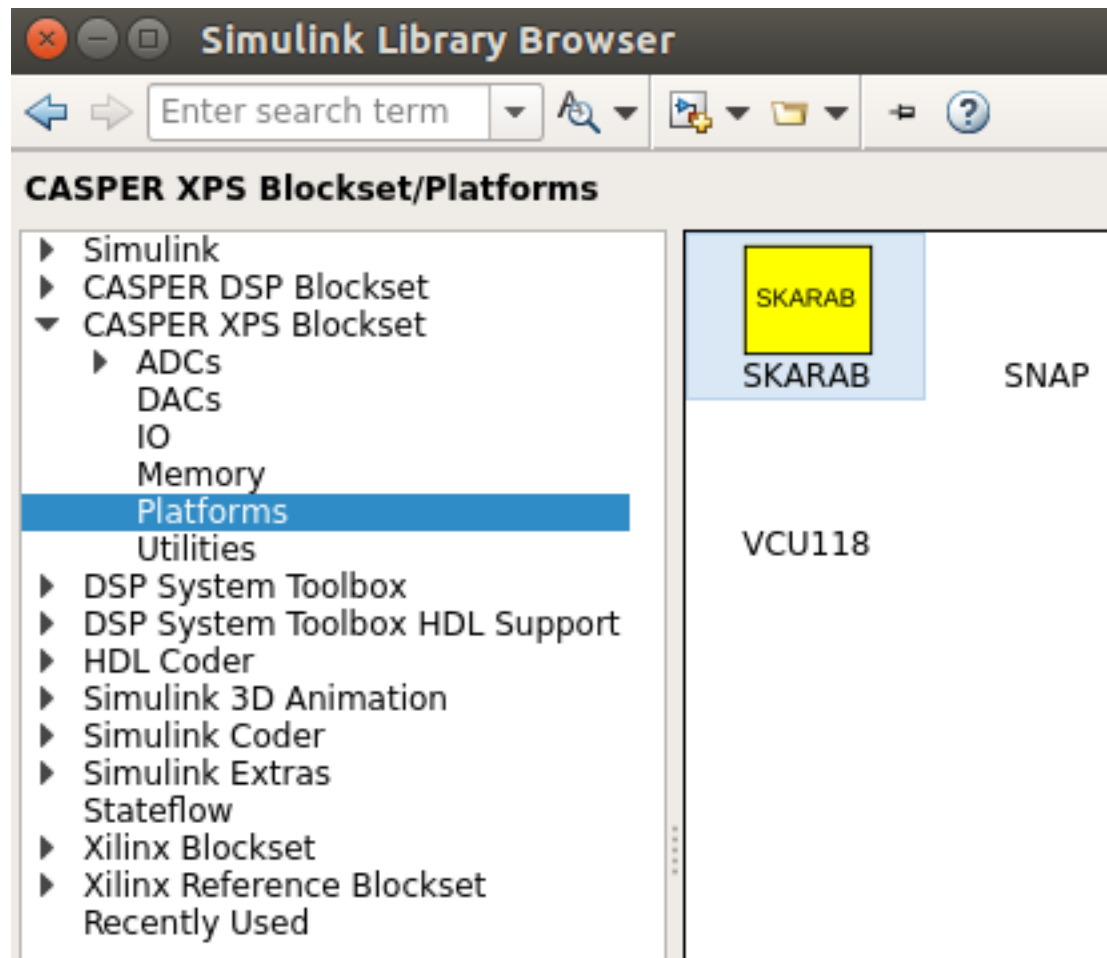
Add a System generator block from the Xilinx library by locating the Xilinx Blockset library’s Basic Elements subsection and dragging a System Generator token onto your new file.



Do not configure it directly, but rather add a platform block representing the system you are compiling for. These can be found in the CASPER XPS System Blockset library. For SKARAB (and later) platforms, you need a block which matches the platform name, which can be found in the library under “platforms”, as shown below.



casper_xps_select



casper_xps_select_platform_sk

Double click on the platform block that you just added. The Hardware Platform parameter should match the platform you are compiling for. Once you have selected a board, you need to choose where it will get its clock. In designs including ADCs you probably want the FPGA clock to be derived from the sampling clock, but for this simple design (which doesn't include an ADC) you should use the platform's on-board clock. To do this, set the User IP Clock Source to sys_clk. The on-board clock frequency should be greater than 156.25 MHz. 230 MHz should work well.

The configuration yellow block knows what FPGA corresponds to which platform, and so it will automatically configure the System Generator block which you previously added.

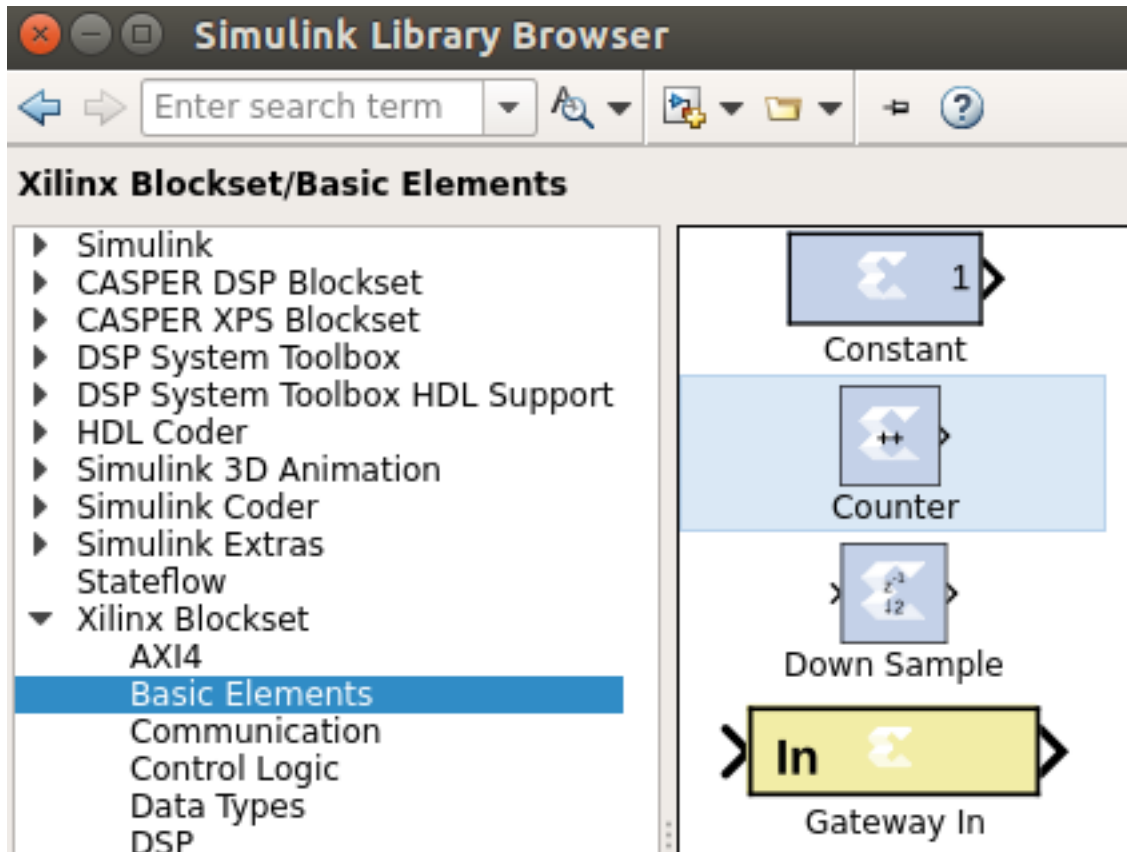
The System Generator and XPS Config blocks are required by all CASPER designs

Flashing LED

To demonstrate the basic use of hardware interfaces, we will make an LED flash. With the FPGA running at ~100MHz (or greater), the most significant bit (msb) of a 27 bit counter will toggle approximately every 0.67 seconds. We can output this bit to an LED on your board. Most (all?) CASPER platforms have at least four LEDs, with the exact configuration depending on the board (for example, SKARAB has four green LEDs and four red LEDs). We will make a small circuit connecting the top bit of a 27 bit counter to one of these LEDs. When compiled this will make the LED flash with a 50% duty cycle approximately once a second.

Add a counter

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



xilinx_select_counter.png

Double-click it and set it for free running, 27 bits, unsigned. This means it will count from 0 to $2^{27} - 1$, and will then wrap back to zero and continue.

counter_led (Xilinx Counter)

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☐ Provide synchronous reset port

☐ Provide enable port

Explicit Sample Period

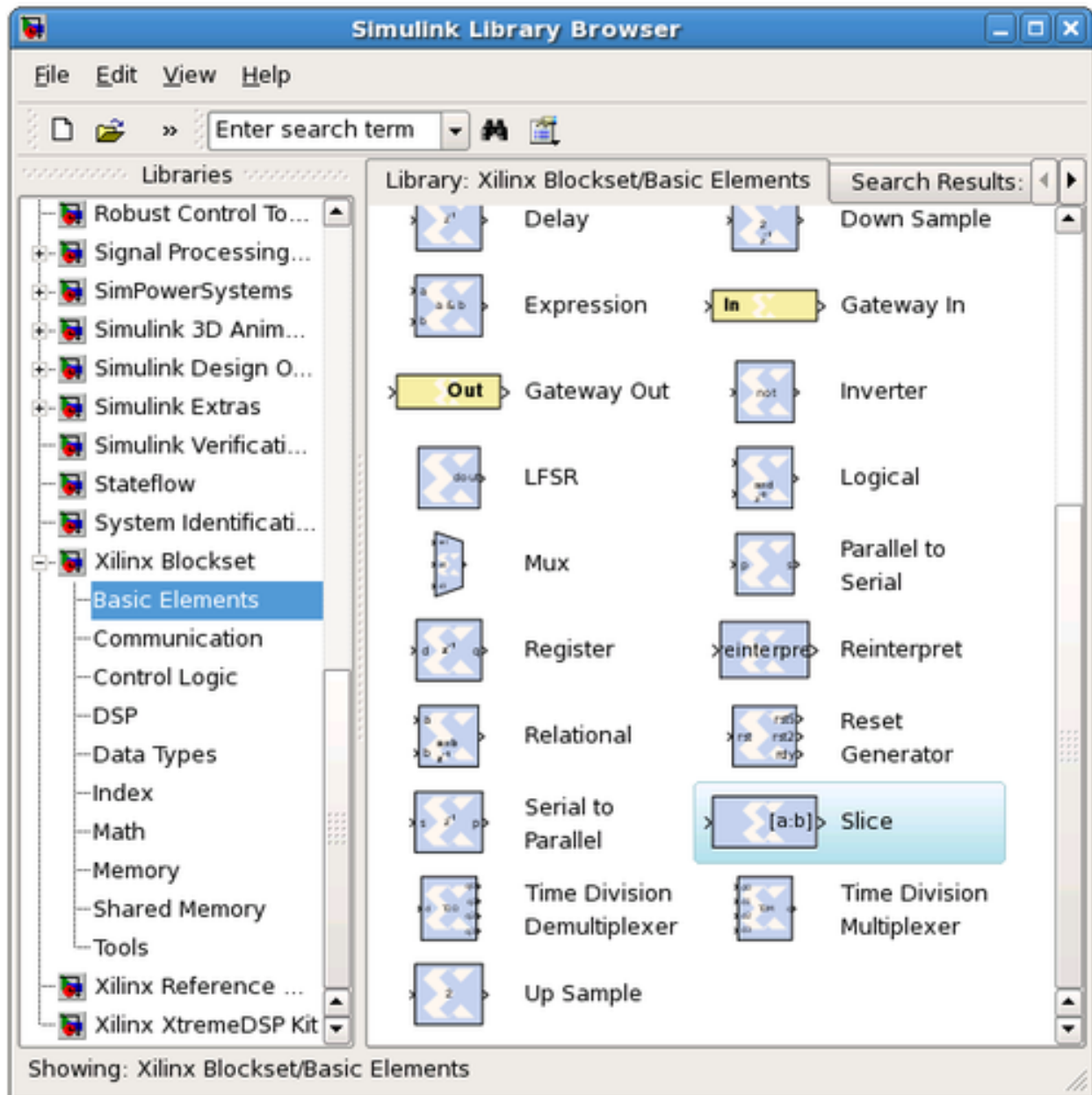
Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

OK **Cancel** **Help** **Apply**

Add a slice block to select out the msb

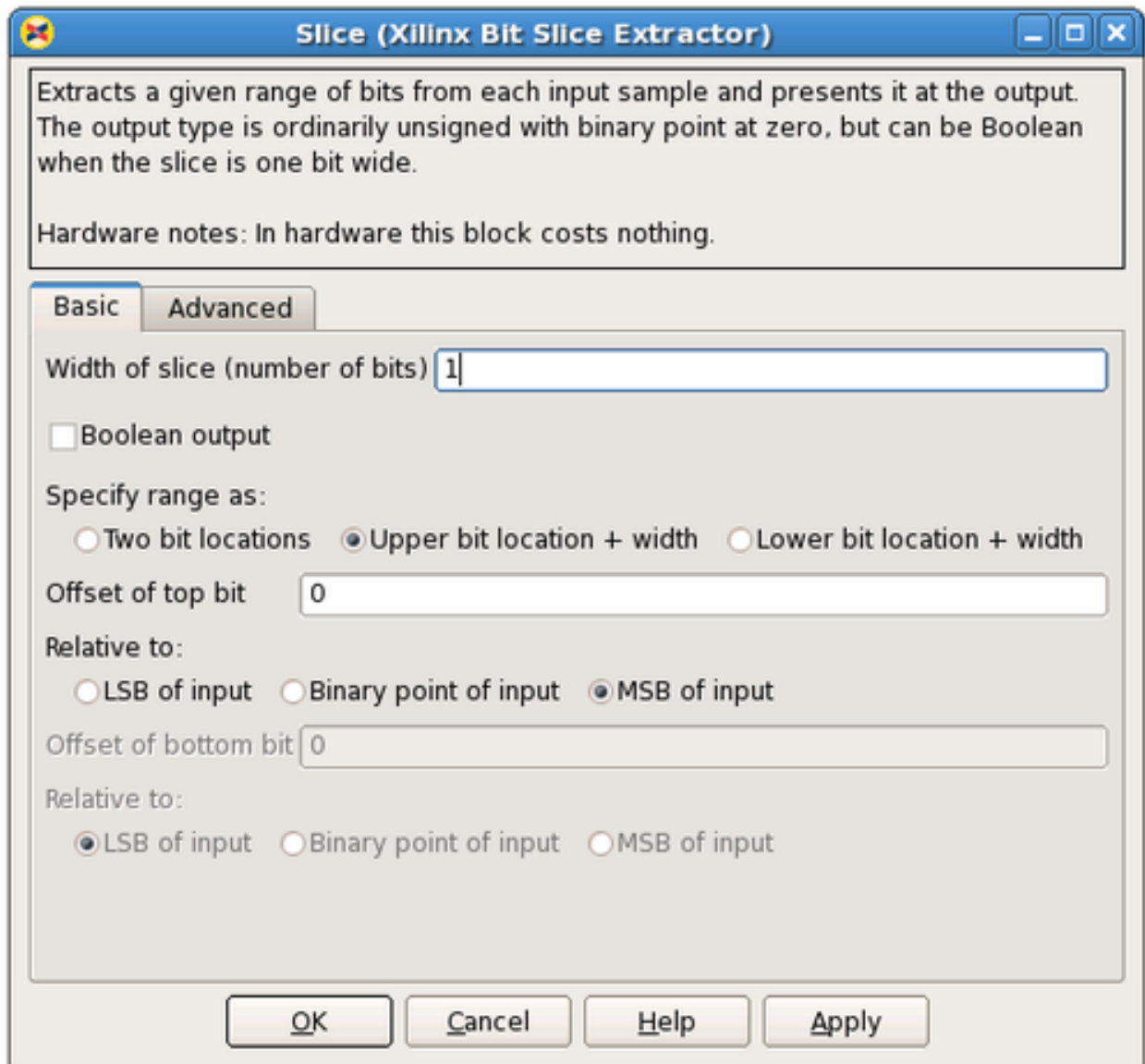
We now need to select the **most significant bit** (msb) of the counter. We do this using a slice block, which Xilinx provides. Xilinx Blockset -> Basic Elements -> Slice.



Slice_select.png

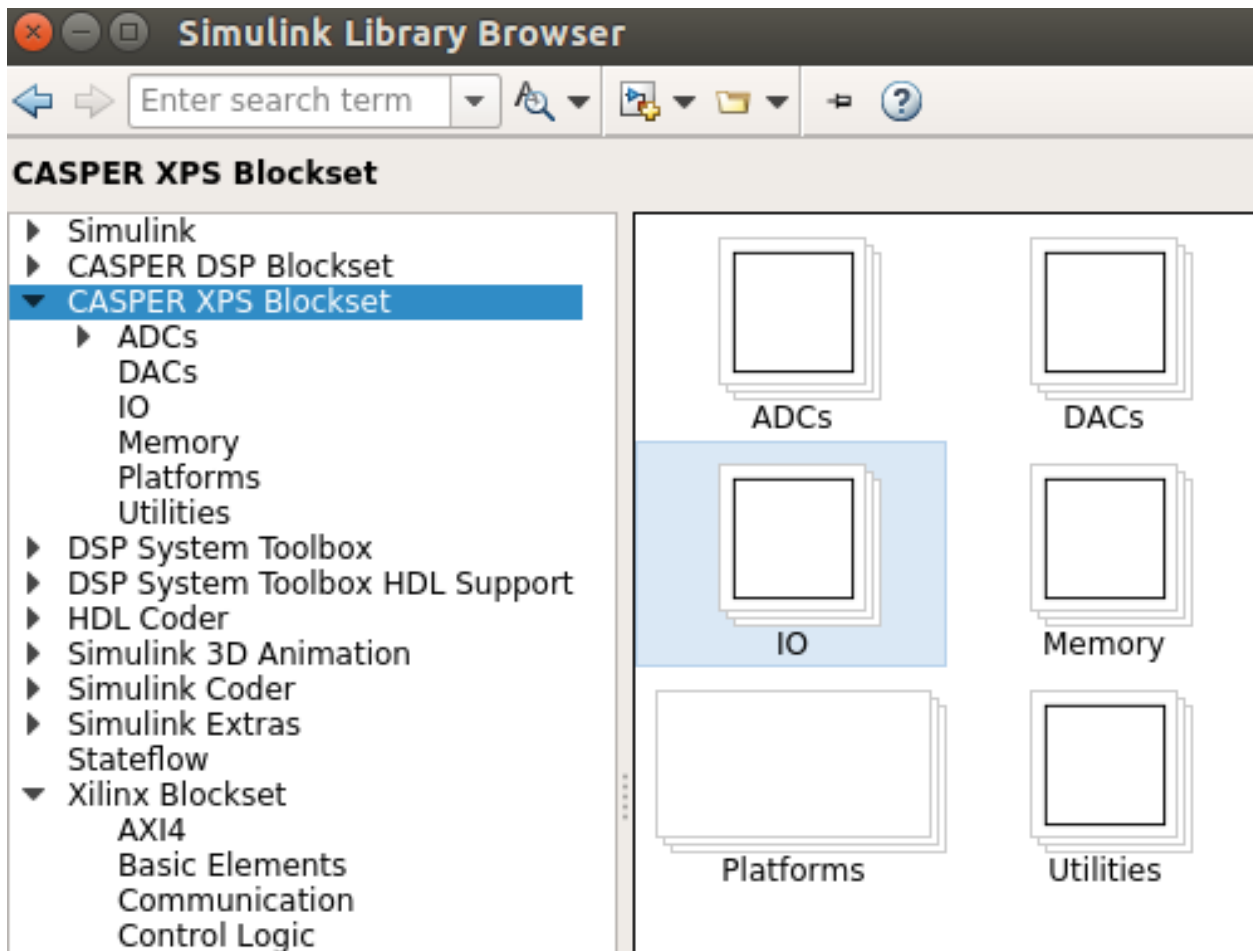
Double-click on the newly added slice block. There are multiple ways to select which bit(s) you want. In this case, it is simplest to index from the upper end and select the first bit. If you wanted the **least significant bit** (lsb), you can also index from that position. You can either select the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

Set it for 1 bit wide with offset from top bit at zero. As you might guess, this will take the 27-bit input signal, and output just the top bit.

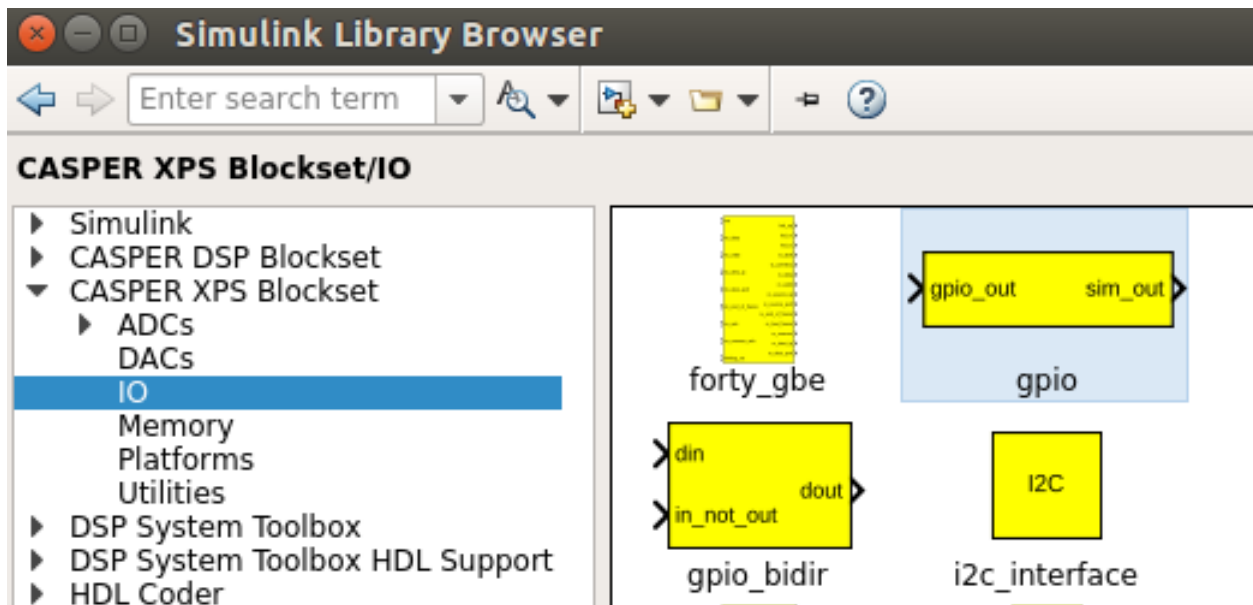


Add a GPIO block

From: CASPER XPS library -> IO -> gpio.



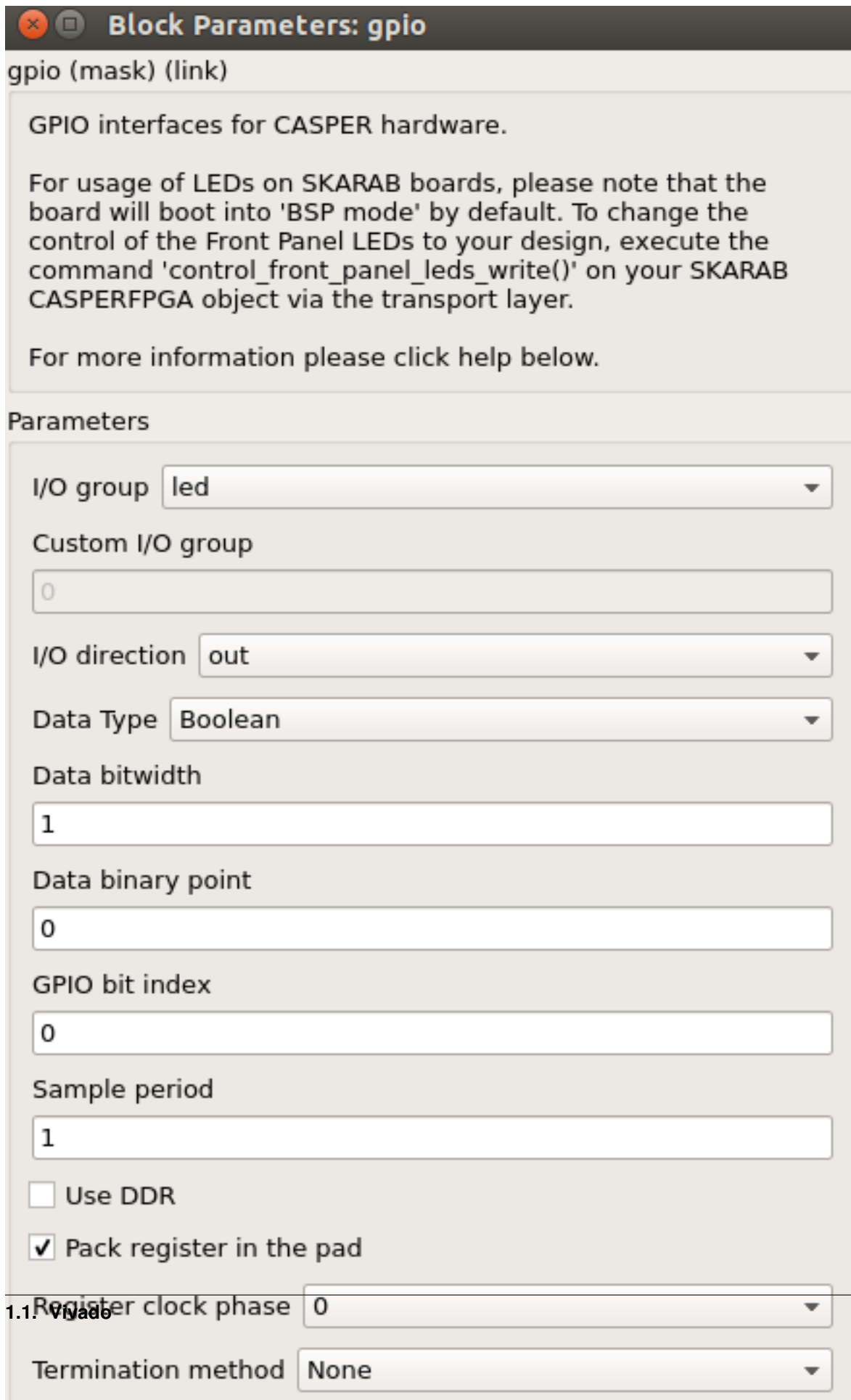
casper_xps_select



casper_xps_select

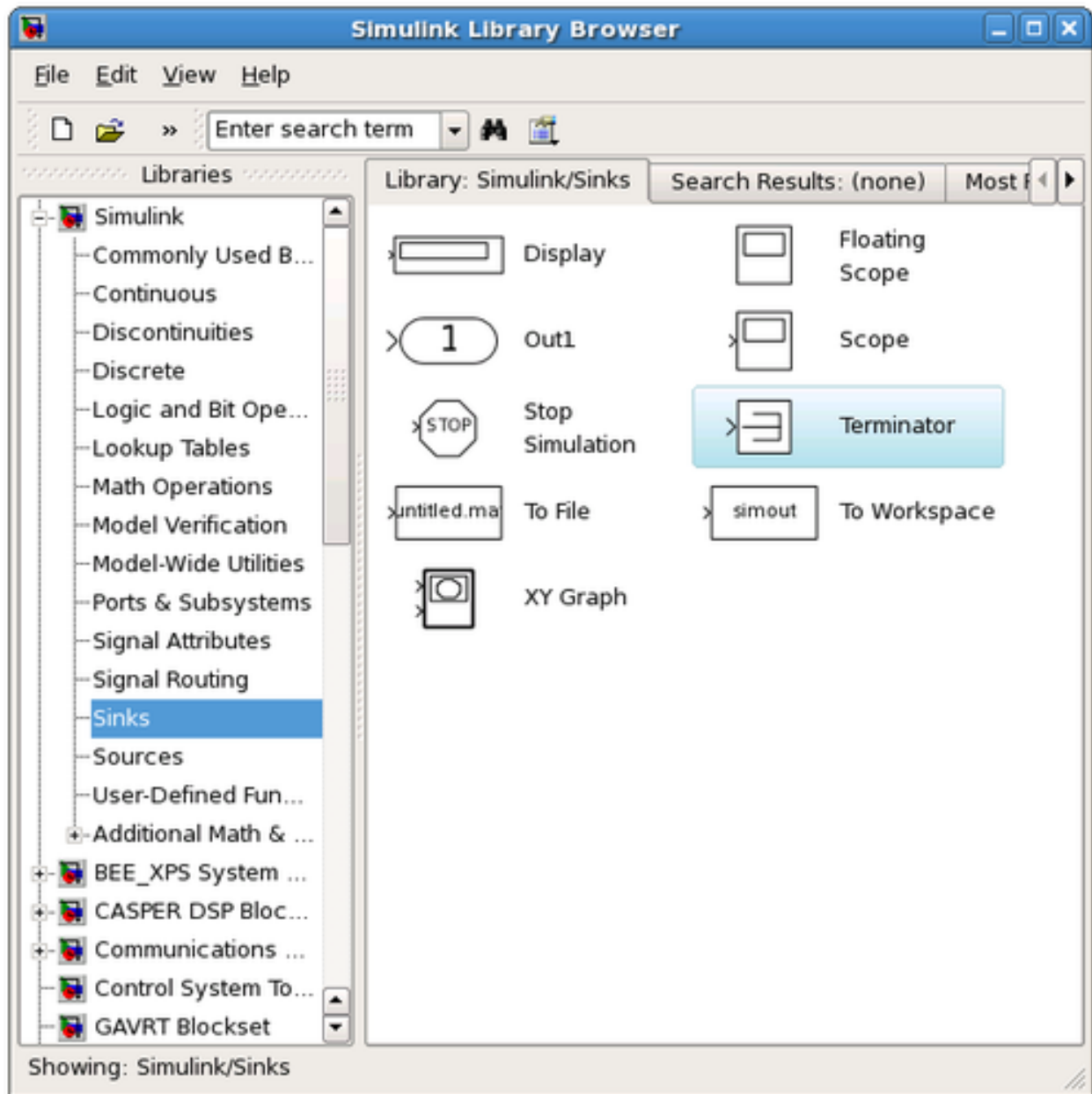
In order to send the 1 bit signal you have sliced off to an LED, you need to connect it to the right FPGA output pin. To do this you can use a GPIO (general-purpose input/output) block from the XPS library, this allows you to route a signal from Simulink to a selection of FPGA pins, which are addressed with user-friendly names. Set it to use SKARAB's LED bank as output. Once you've chosen the LED bank, you need to pick *which* LED you want to output to. Set the

GPIO bit index to 0 (the first LED) and the data type to Boolean with bitwidth 1. This means your simulink input is a 1 bit Boolean, and the output is LED0.



Add a terminator

To prevent warnings (from MATLAB & Simulink) about unconnected outputs, terminate all unused outputs using a *Terminator*:



From: Simulink -> Sinks -> Terminator

You can also use the Matlab function `XIAddTerms`, run in the MATLAB prompt, to automatically terminate your unused outputs.

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.

To connect the blocks simply click and drag from the ‘output arrow’ on one block and drag it to the ‘input arrow’ of another block. Connect the blocks together: Counter -> Slice -> gpio as showing in digram below.

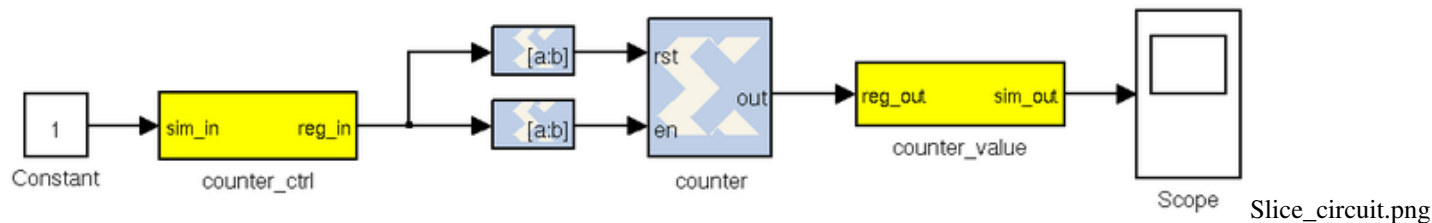


Remember to save your design often.

Software control

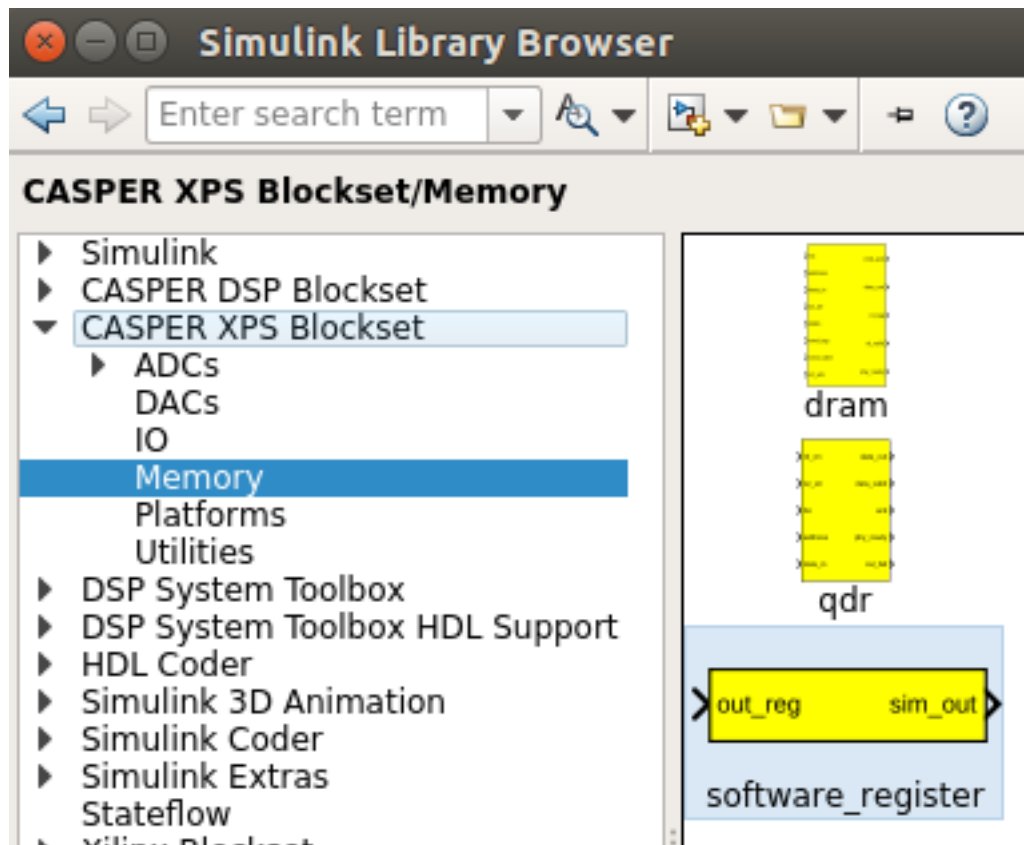
To demonstrate the use of software registers to control the FPGA from a computer, we will add a registers so that the counter in our design can be started, stopped, and reset from software. We will also add a register so that we can monitor the counter’s current value too.

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





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 CASPER XPS System Blockset library, drag two Software Registers onto your design.



casper_xps_select_memory_swreg.png

Set the I/O direction to *From Processor* on the first one (counter control) to enable a value to be set from software and sent *to* your FPGA design. Set it to *To Processor* on the second one (counter value) to enable a value to be sent *from* the FPGA to software. Set both registers to a bitwidth of 32 bits.

 **Block Parameters: counter_ctrl**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction From Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts

0



Bitfield types, ufix=0, fix=1, bool=2

0

☒ Provide sim input/output?

1.1. Vivado

☐ Print format string?

 **Block Parameters: counter_value**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction To Processor

I/O delay

Initial Value

Sample period

Bitfield names [msb...lsb]

Bitfield widths

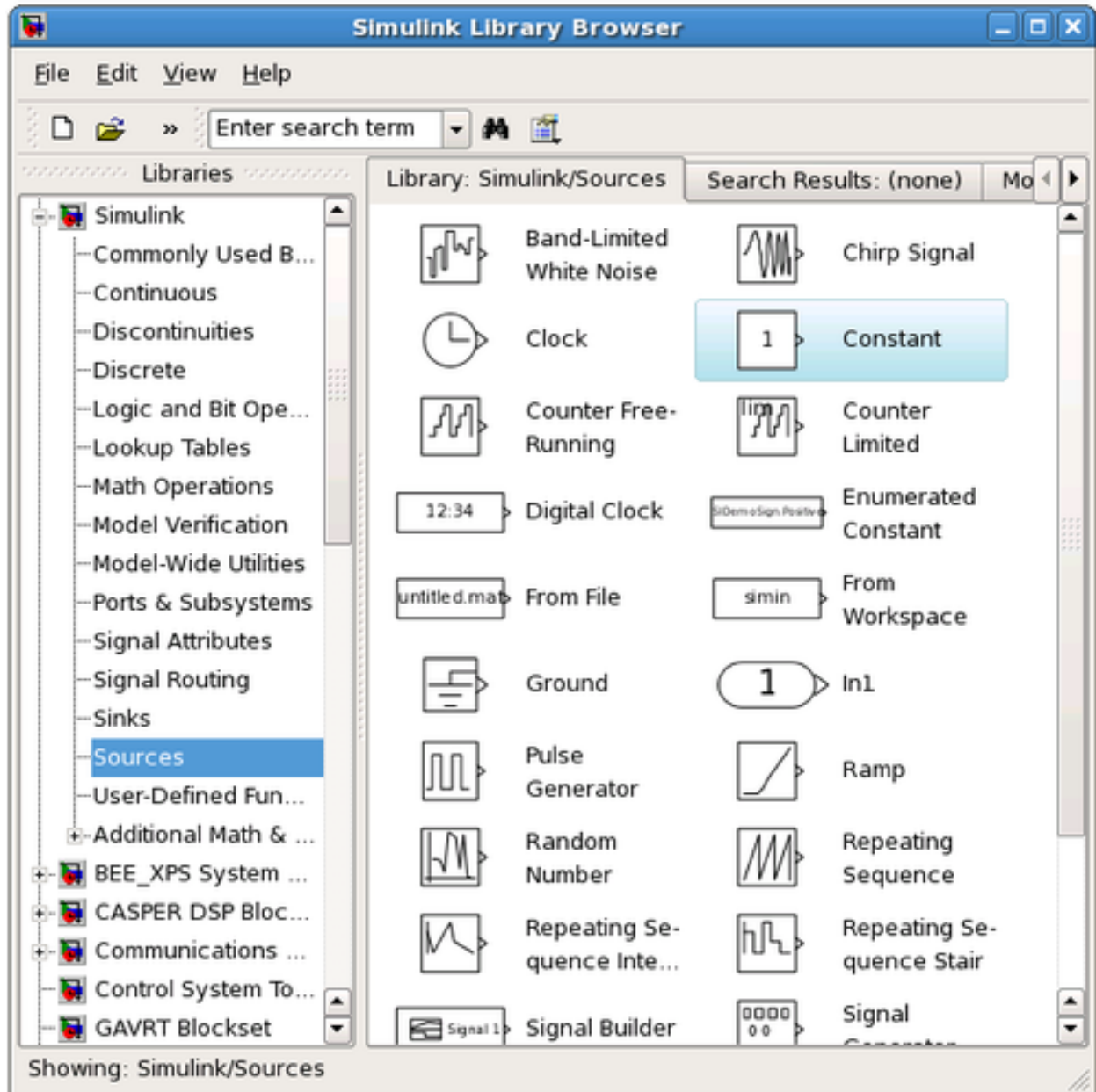
Bitfield binary pts

Bitfield types, ufix=0, fix=1, bool=2

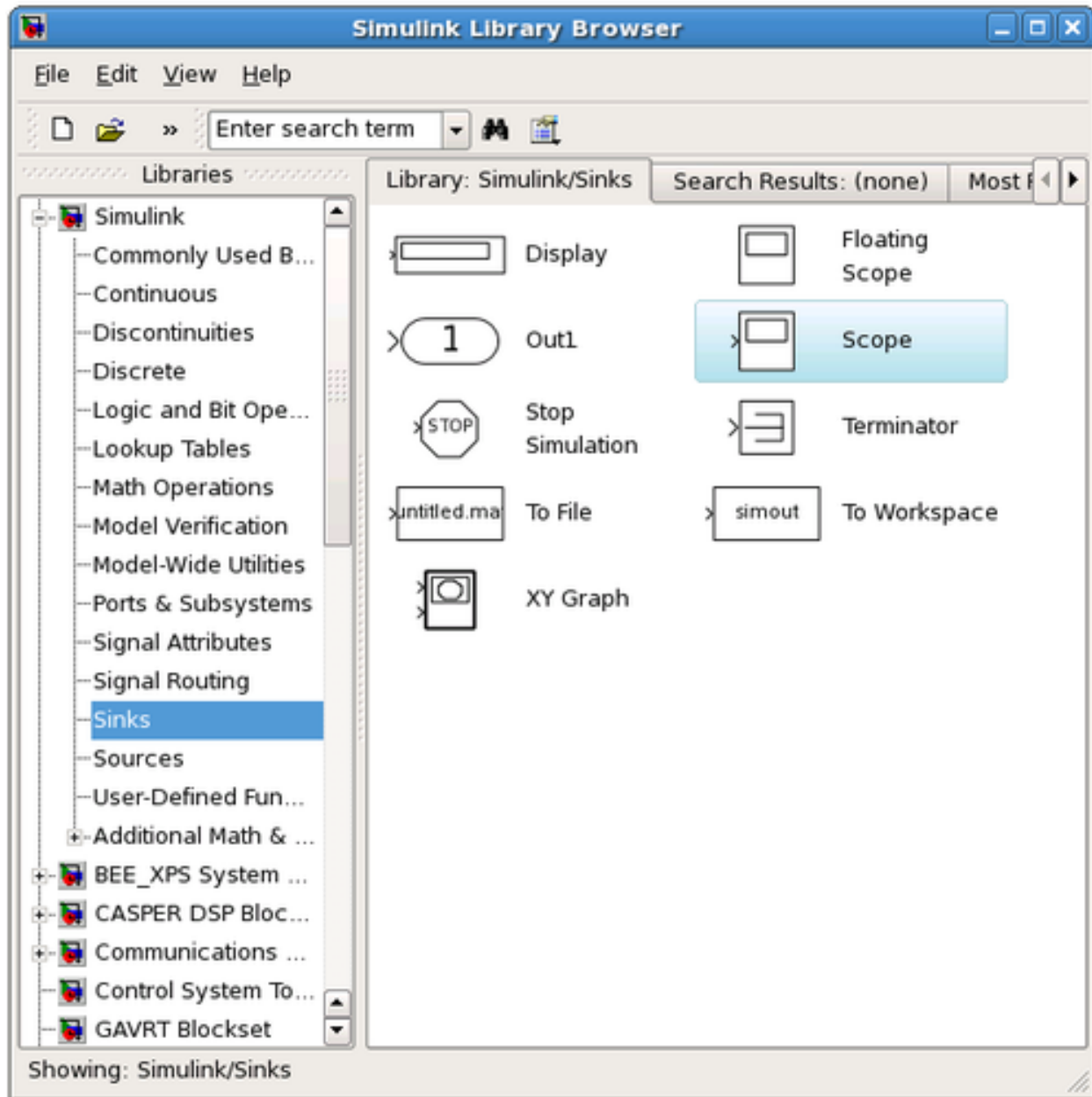
Rename the registers to something sensible. The names you give them here are the names you will use to access them from software. Do not use spaces, slashes and other funny characters in these. Perhaps *counter_ctrl* and *counter_value*, to represent the control and output registers respectively.

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

For now, set the *sim_reg* port to constant one using a Simulink-type constant. Found in *Simulink* -> *Sources*. This will enable the counter during simulations.



During simulation, we can monitor the counter's value using a scope (*Simulink* -> *Sinks*):



Here is a good point to note that all blocks from the *Simulink* library (usually white), will not be compiled into hardware. They are present for simulation only.


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

You need to use *gateway* blocks whenever connecting a Simulink-provided block (like a scope or sine-wave generator) to a from a Xilinx block, this will sample and quantize the simulink signals so that they are compatible with the Xilinx world. Some blocks (like the software register) provide a gateway internally, so you can feed the input of a software register with a xilinx signal, and monitor its output with a Simulink scope. However, in general, you must manually insert these gateways where appropriate. Simulink will issue warnings for any direct connections between the Simulink and Xilinx worlds.

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:

 **counter (Xilinx Counter)** [-] [] [X]

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Advanced** **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☒ Provide synchronous reset port

☒ Provide enable port

Explicit Sample Period

Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

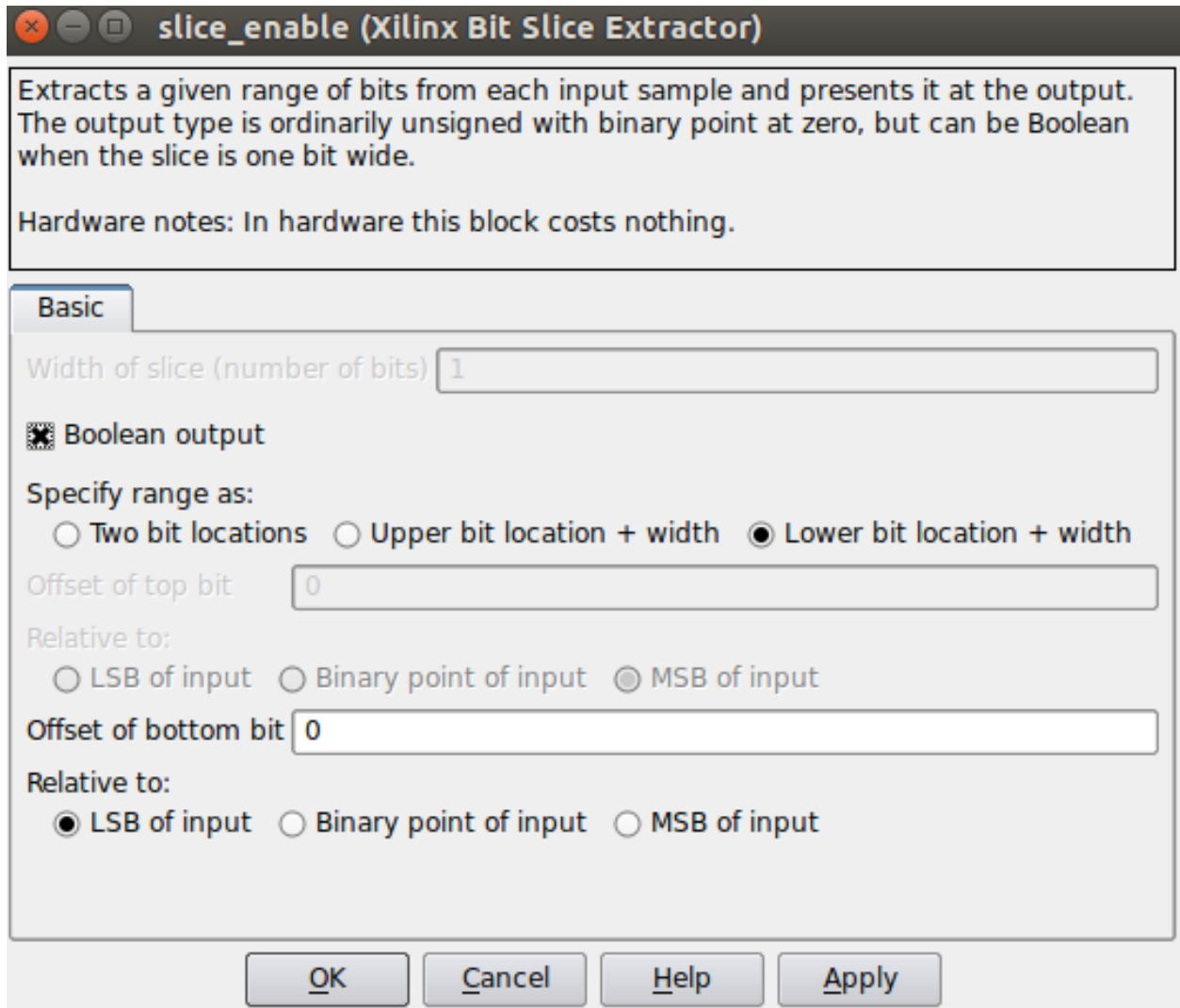
Add the slice blocks

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.

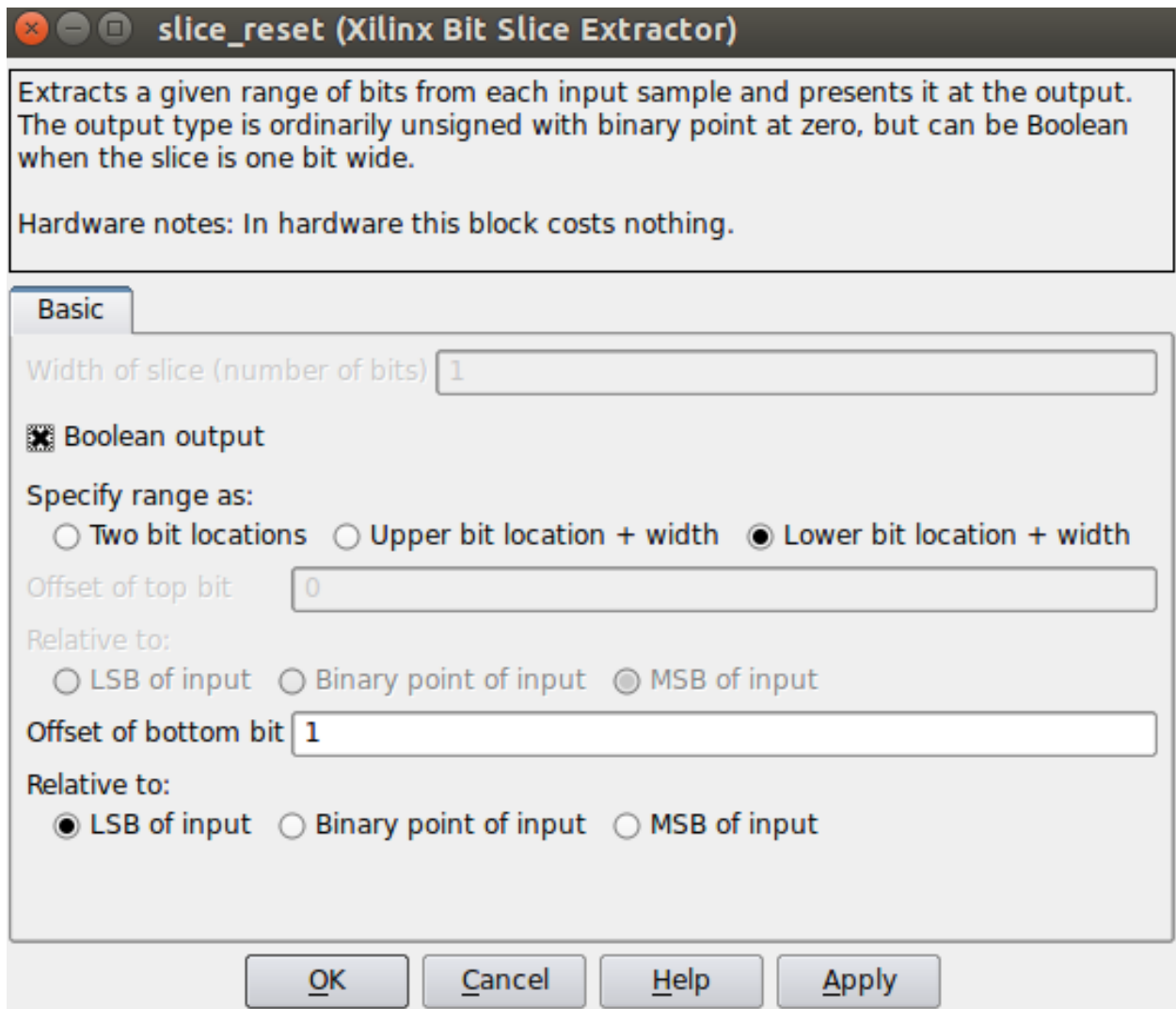
The enable and reset ports of the counter require boolean values (which Simulink interprets differently from ordinary 1-bit unsigned numbers). Configure the slices as follows:

Slice for enable:



casper_xps_param

Slice for reset:

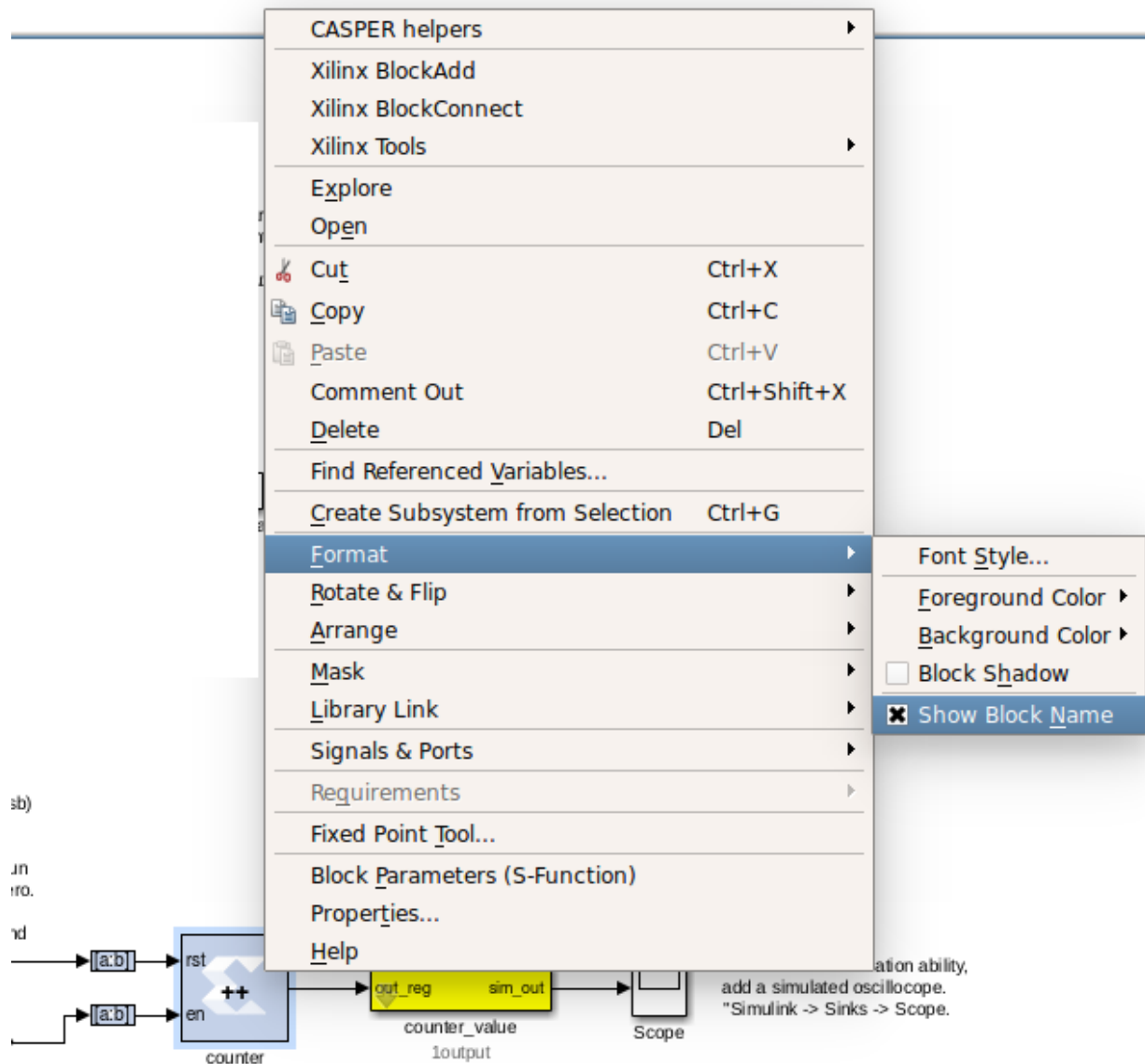


casper_xps_param

Connect 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.

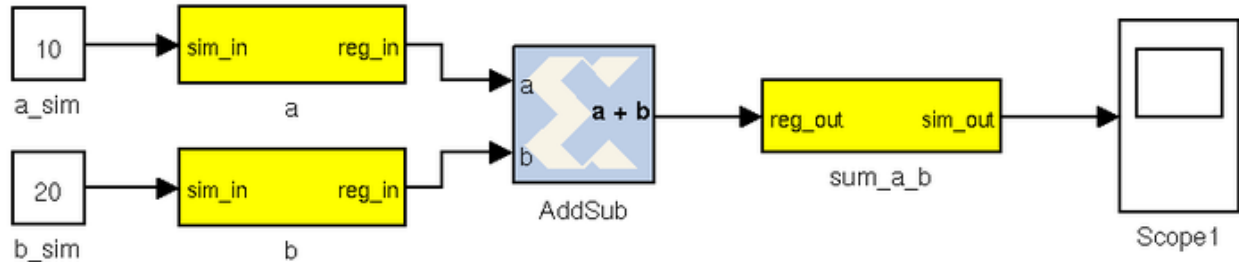
Do so by right-clicking and unchecking Format → Show Block Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



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 $a+b = \text{sum_a_b}$.



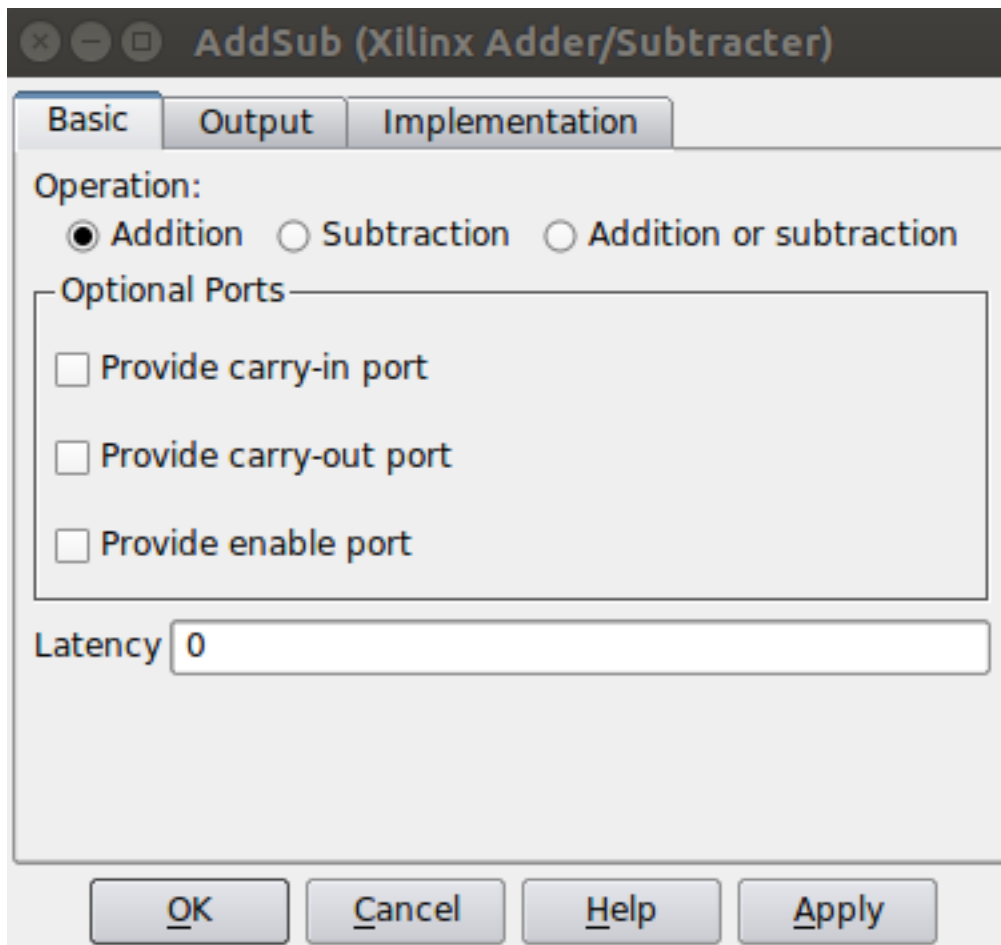
Add the software registers

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.

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.



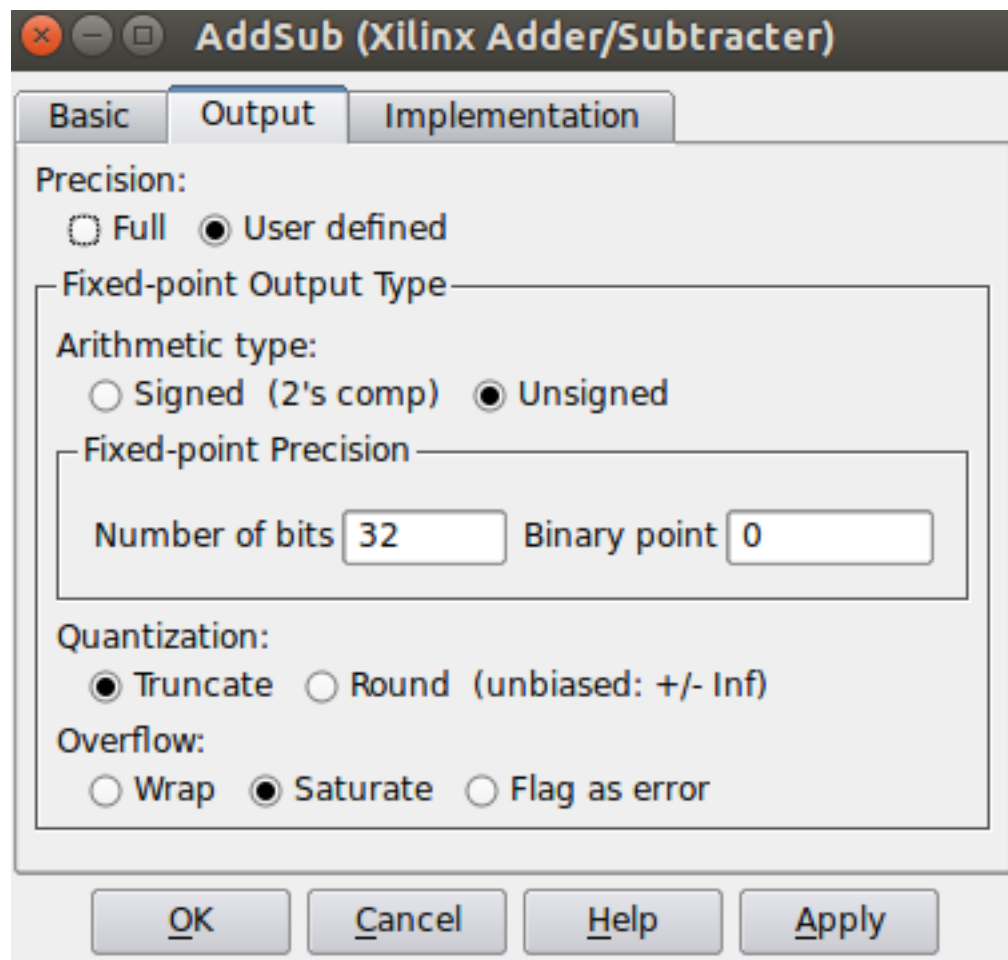
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

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 $2^{32} - 1$.

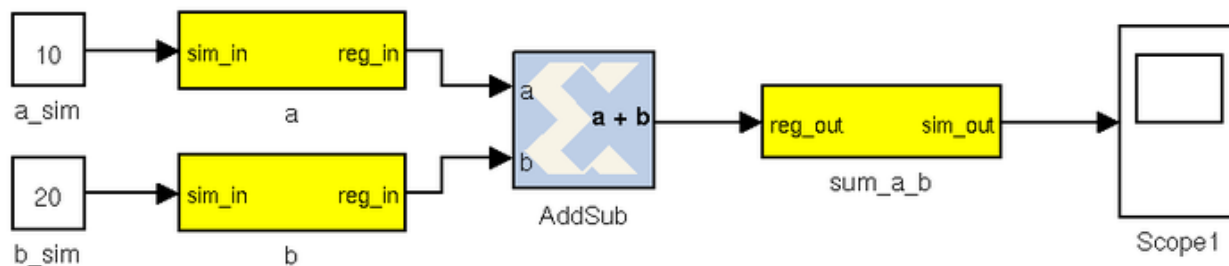


Add the scope and simulation inputs

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.

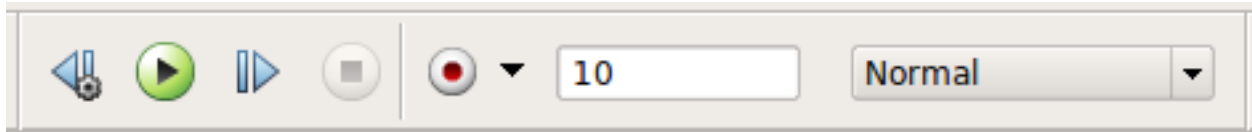
Connect it all together

Like this:



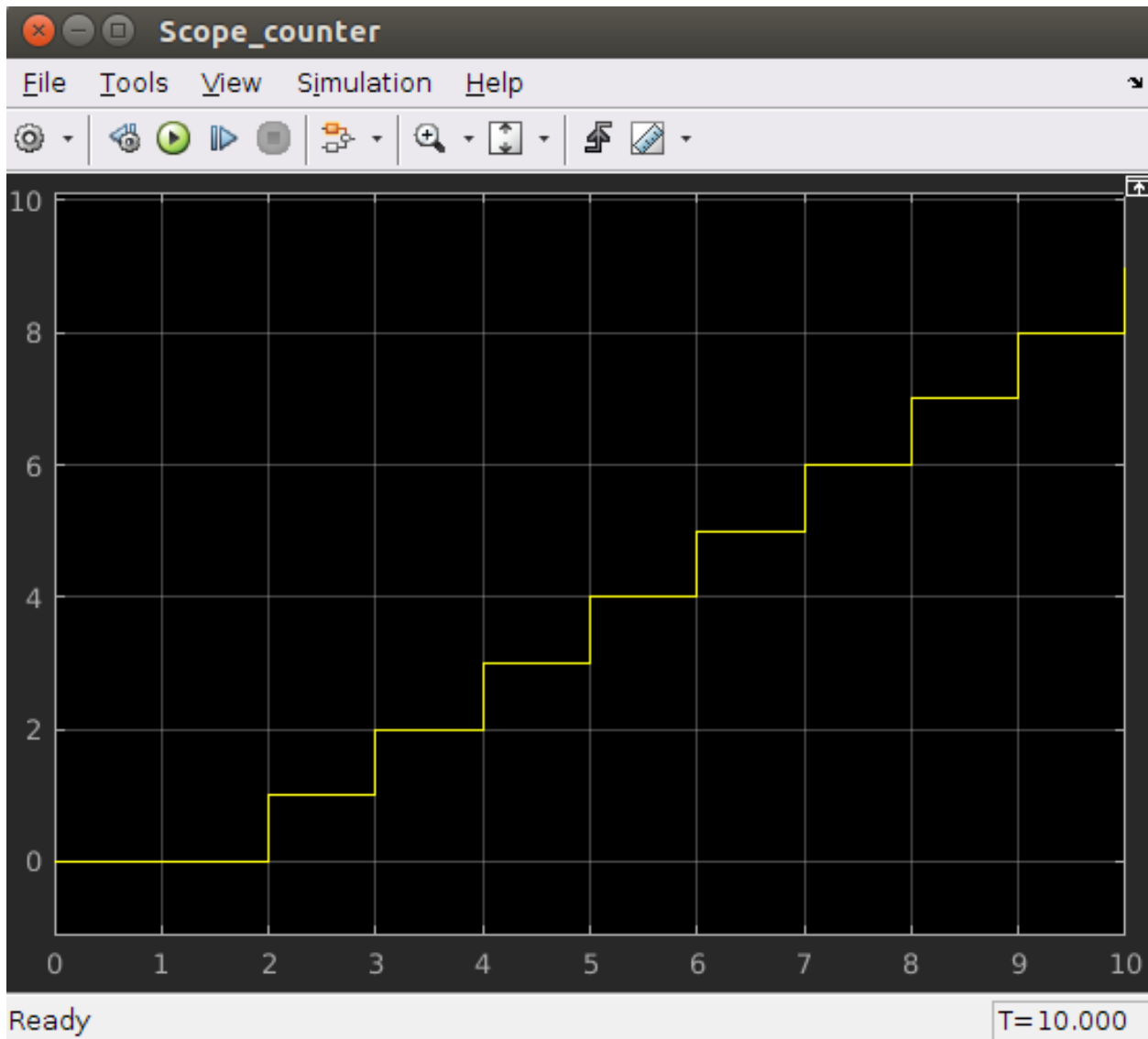
Simulating

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.

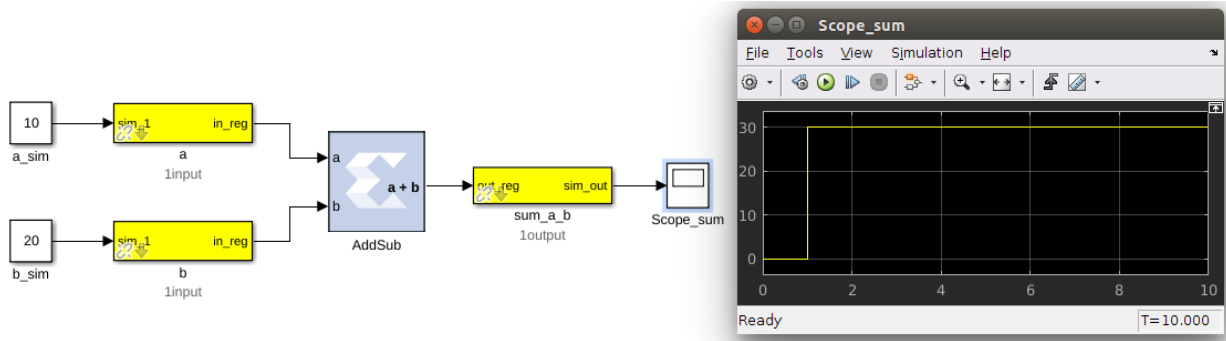


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

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...

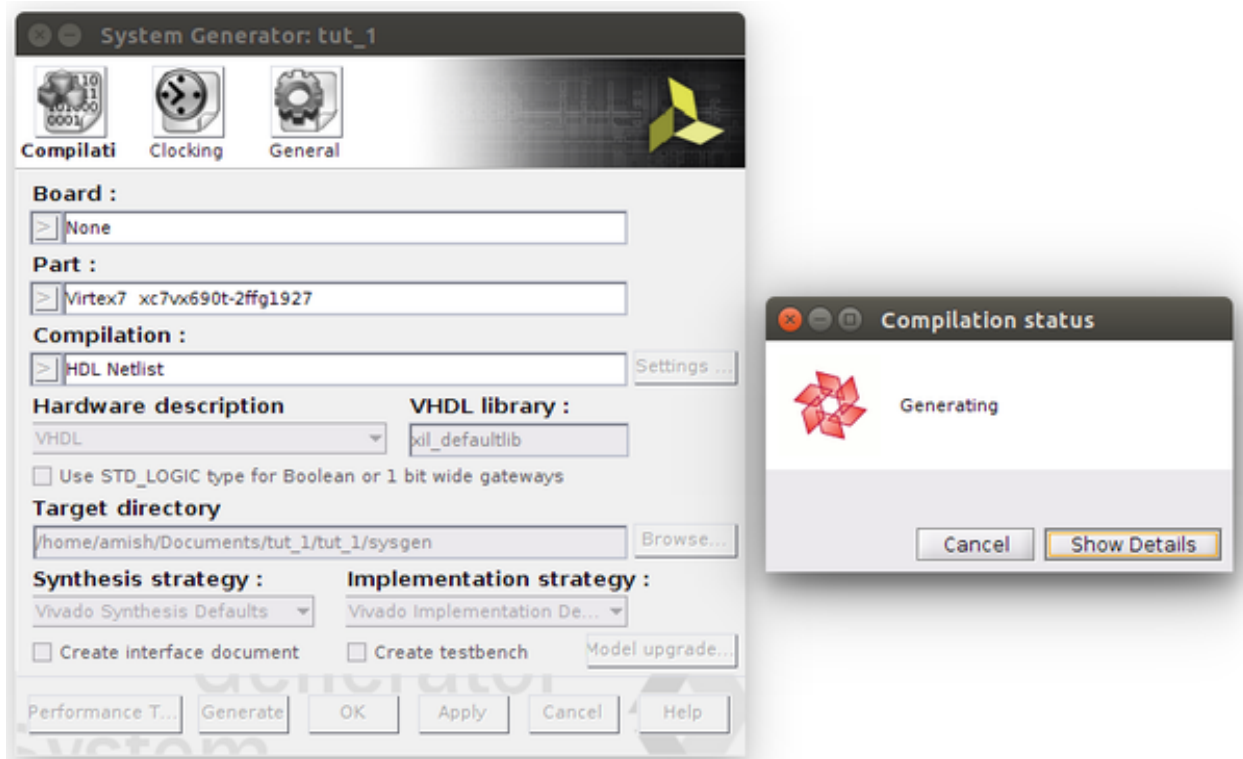
Compiling

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 156.25MHz system clock crystal and to your specified User IP Clock Rate, but they will operate independently.

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window. **THIS COMMAND DEPENDS WHICH PLATFORM YOU ARE TARGETING:**

```
>> jasper
```

When a GUI pops up, click "Compile!". This will run the complete build process, which consists of two stages. The first involving Xilinx's System Generator, which compiles any Xilinx blocks in your Simulink design to a circuit which can be implemented on your FPGA. While System Generator is running, you should see the following window pop up:



After this, the second stage involves synthesis of your design through Vivado, which goes about turning your design into a physical implementation and figuring out where to put the resulting components and signals on your FPGA. Finally the toolflow will create the final output fpg file that you will use to program your FPGA. This file contains the bitstream (the FPGA configuration information) as well as meta-data describing what registers and other yellow blocks are in your design. This file will be created in the 'outputs' folder in the working directory of your Simulink model. **Note: Compile time is approximately 15-20 minutes.**

```
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/
myproj/ outputs/ sysgen/
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/outputs/
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$ ls -lt
total 8336
-rw-rw-r-- 1 amish amish 1249412 Jul 25 14:18 tut_1_2017-7-25_1355.fpg
-rw-rw-r-- 1 amish amish 7280615 Jul 25 14:18 tut_1_2017-7-25_1355.bof
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$
```

Advanced Compiling

Once you are familiar with the CASPER toolflow, you might find you want to run the two stages of the compile separately. This means that MATLAB will become usable sooner, since it won't be locked up by the second stage of the compile. If you want to do this, you can run the first stage of the compile from the MATLAB prompt with

```
>> jasper_frontend
```

After this is completed, the last message printed will tell you how to finish the compile. It will look something like:

```
$ python /path_to/mlib_devel/jasper_library/exec_flow.py -m /home/user/path_to/skarab/
↳tut_intro/skarab_tut_intro.slx --middleware --backend --software
```

You can run this command in a separate terminal, after sourcing appropriate environment variables. Not recommended for beginners.

Programming the FPGA

Reconfiguration of CASPER FPGA boards is achieved using the casperfpga python library, created by the SA-SKA group.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration. However, should you want to run these tutorials on your own machines, you should download the latest casperfpga libraries from [here](#).

Copy your .fpg file to your Server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server. Instructions to do this are available [here](#)

Connecting to the board

SSH into the server that the SKARAB board is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
$ ipython
```

Now import the fpga control library. This will automatically pull-in the KATCP library and any other required communications libraries.

```
import casperfpga
```

To connect to the board we create a CasperFpga instance; let's call it fpga. The CasperFpga constructor requires just one argument: the IP hostname or address of your FPGA board.

```
fpga = casperfpga.CasperFpga('skarab hostname or ip_address')
```

The first thing we do is program the FPGA with the .fpg file which your compile generated.

```
fpga.upload_to_ram_and_program('your_fpgfile.fpg')
```

Should the execution of this command return true, you can safely assume the FPGA is now configured with your design. The SKARAB's Front Panel LEDs are, by default, displaying inherent Board Support Package diagnostics. More details on this can be found [here](#). The control of the Front Panel LEDs can be switched between BSP functionality and your Simulink design via a control signal. This control is available to you in casperfpga through `control_front_panel_leds_read()` and `control_front_panel_leds_write(dsp_override=True)`. You should see the LED on your board flashing. Go check! All the available/configured registers can be displayed using: `fpga.listdev()` The adder and counter can be controlled by [writing to](#) and [reading from](#) registers added in the design using:

```
fpga.write_int('a',10)
fpga.write_int('b',20)
fpga.read_int('sum_a_b')
```

With any luck, the sum returned by the FPGA should be correct.

You can also try writing to the counter control registers in your design. You should find that with appropriate manipulation of the control register, you can make the counter start, stop, and return to zero.

```
fpga.write_int('counter_ctrl',10')
fpga.read_uint('counter_value')
```

Conclusion

This concludes the first CASPER Tutorial. You have learned how to construct a simple Simulink design, program an FPGA board and interact with it with Python using `casperfpga`. Congratulations!

While the design you made might not be very complicated, you now have the basic skills required to build more complex designs which are covered in later tutorials.

1.1.7 Tutorial 2: 40GbE Interface

Introduction

In this tutorial, you will create a simple Simulink design which uses the SKARAB's 40GbE ports to send data at high speeds to another port. This could just as easily be another SKARAB board or a computer with a 40GbE network interface card. In addition, you will learn to control the design remotely using a supplied Python library for KATCP. The UDP packets sent by the SKARAB will be recorded to disk.

This tutorial essentially implements the transmission of a counter through one QSFP+ port and back into another. This allows for a test of the communications link in terms of performance and reliability. This test can be used to test the link between boards and the effect of different cable lengths on communications quality.

Background

For more info on the SKARAB please follow this link to the [SKARAB](#) hardware platform. Of particular interest for this tutorial is the section on the [QSFP+ Mezzanine Card](#).

The maximum payload length of the 40GbE core is 8192 bytes (implemented in BRAM) plus another 512 bytes (implemented in distributed RAM) - which is useful for an application header. These ports (and hence part of the 40 GbE cores) run at 156.25MHz, while the interface to your design runs at the FPGA clock rate (sys_clk, etc). The interface is asynchronous and buffers are required at the clock boundary. For this reason, even if you send data between two SKARAB boards which are running off the same hard-wired clock, there will be jitter in the data. A second consideration is how often you clock values into the core when you try to send data. If your FPGA is running faster than the core, and you attempt to clock data in on every clock cycle, the buffers will eventually overflow. Likewise for receiving, if you send too much data to a board and cannot clock it out of the receive buffer fast enough, the receive buffers will overflow and you will lose data. In our design we are clocking the FPGA at 200MHz with the cores running at 156.25MHz. We will therefore not be able to clock data into the Tx buffer continuously for very long before it overflows. If this doesn't make much sense to you now don't panic, it will become clear after you've tried it.

Tutorial Outline

This tutorial will be run in an explain-and-explore kind of way. There are too many blocks for us to run through each one and its respective configuration. Hence, each section will be generally explained and it is up to you to explore the design and understand the detail. Please don't hesitate to ask any questions during the tutorial session. If you are doing these tutorials outside of the CASPER workshop please email any questions to the [CASPER email list](#).

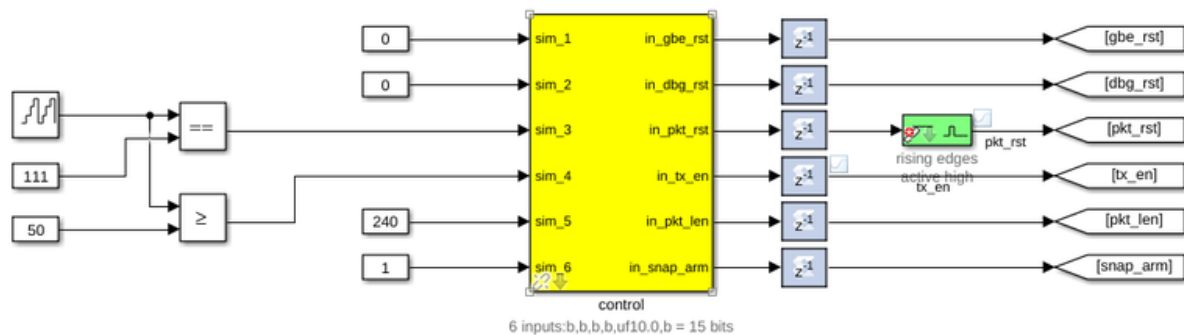
This tutorial consists of 2 designs: a transmitter and a receiver. We will look at the transmitting design first.

Tx Design

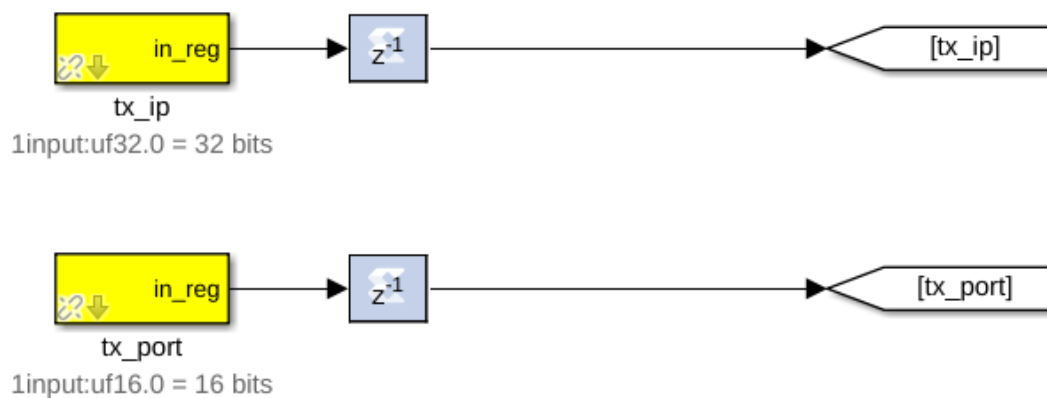
As with the previous tutorial, drop down a Xilinx XSG Block and then the SKARAB platform yellow block. Configure the clock frequency to 170Mhz. Firstly, we use a software register to control our system. In this design we are using a single 32-bit register with the lower 6 bits being used for the following logic signals:

- 40GbE core reset,
- Debug logic reset,
- Packet reset,
- Transmit enable,
- Packet enable, and
- Snap block arming.

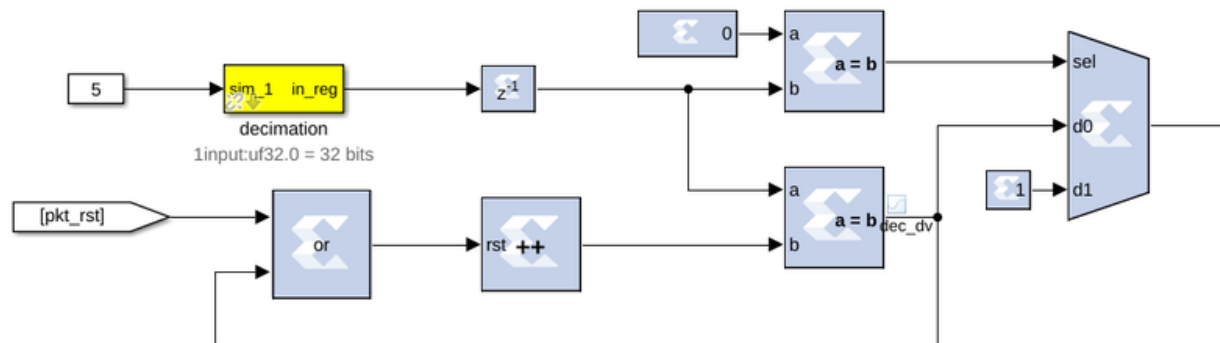
This software register also takes in some simulation stimuli. Try playing with these and stimulate the design using the play button on the top of the window. It is advisable to use a short simulation time as more complex designs can take ages to simulate.



Each packet that is sent from the FPGA fabric can be sent to a specified IP and port. These values are configurable and, in this case, are set via software registers. These could also be set dynamically from the fabric as required.

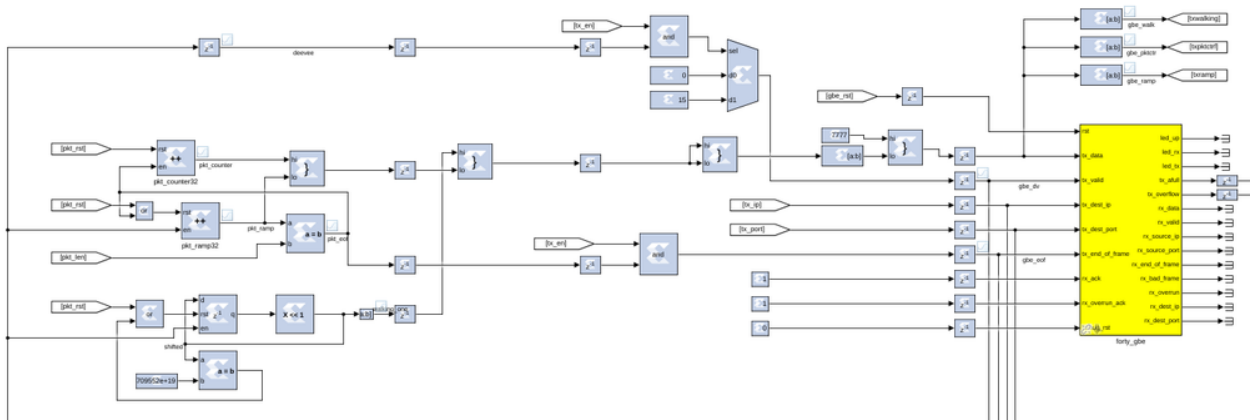


This is the start of the logic to build up our payload. The decimation register is used to control the rate at which packets are sent. Have a look at lines 307-312 of the `Tx python script` to see how this value is calculated and used.

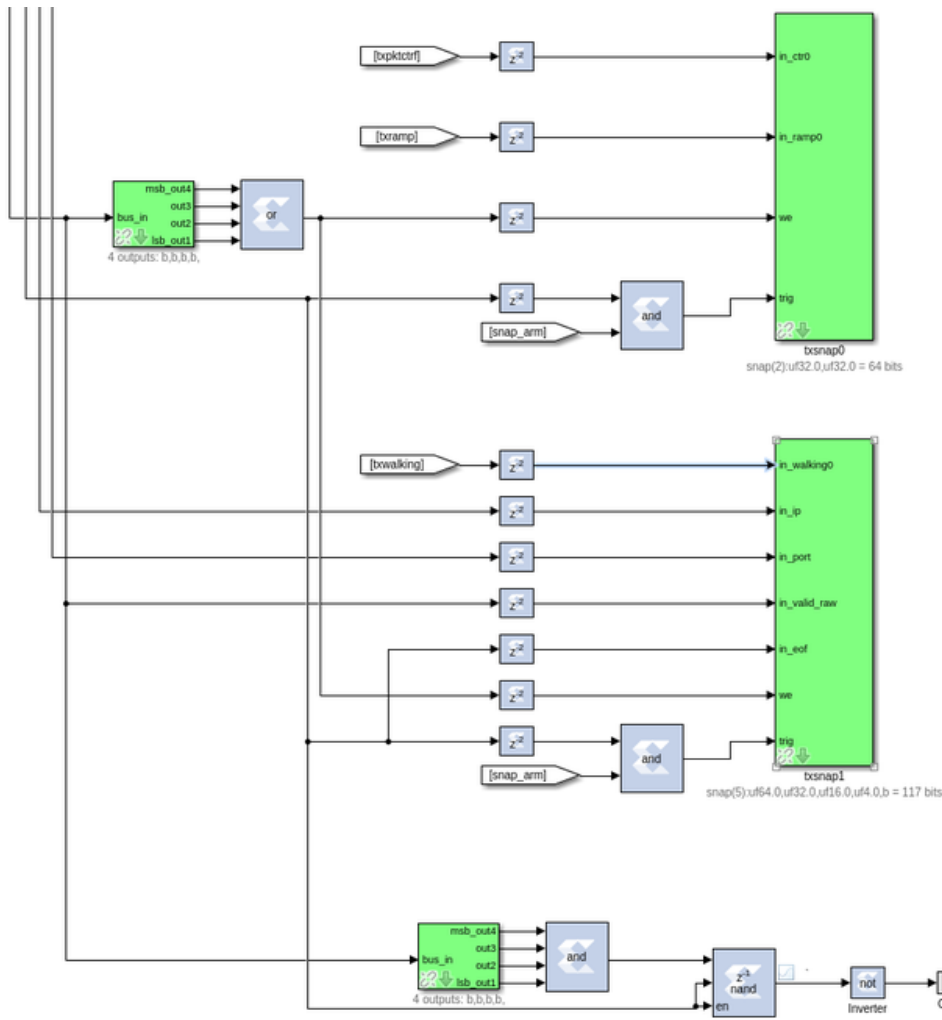


Here is the rest of the payload generation logic. We are creating 2 ramps and a walking-1 pattern. The payload is generated using a combination of counters, slice blocks, delays, adders and comparators. The ramps and walking-1 are concatenated together and put into the payload buffer by toggling the `tx_valid` signal on the 40GbE core. The

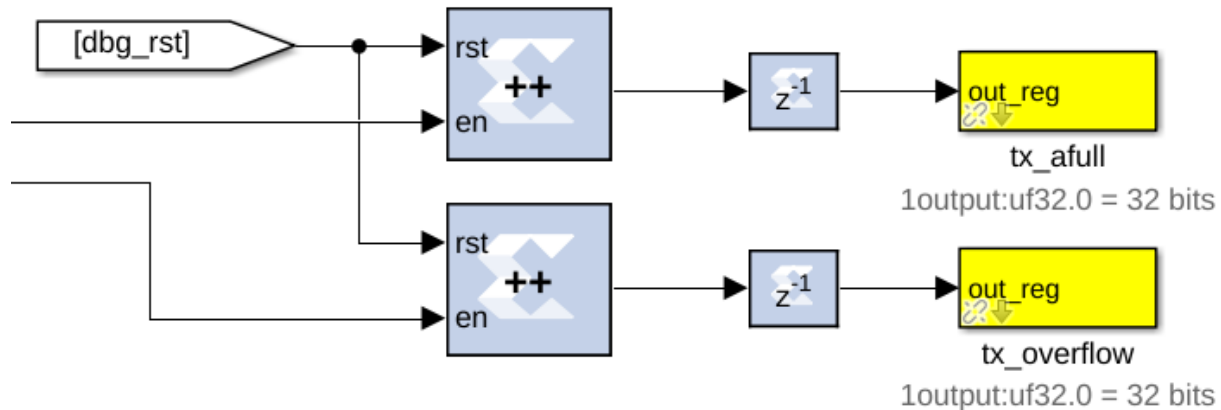
tx_data bus is 256 bits wide so only 256 bits can be clocked in on a clock cycle. The buffer can accept a payload of up to 8192 bytes. Once all the data we require is in the payload buffer we toggle the tx_end_of_frame signal to send the packet into the ether.



As a method of debugging, the transmit side also has some data snapshot (snap, not **SNAP**) blocks which can capture data as it is sent to the core. The snap block is a BRAM which can be triggered to capture data on a particular signal and then read out from software. They are very useful for debugging and checking the data at particular stages through your design.



The design also has a counter that keeps track of each time the overflow or almost-full lines are driven high by the core. This will tell us if we have any overflow or almost-overflowing buffers.

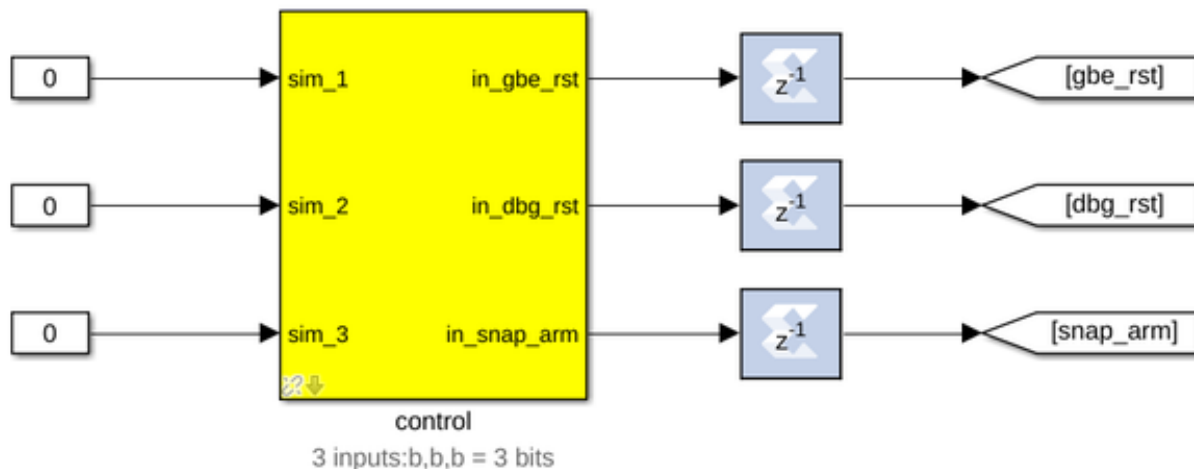


Now take a look through the Tx python script to see how the registers are being set and the debug snap blocks are used to validate the data being sent. This should be well commented, but please ask questions where things aren't clear.

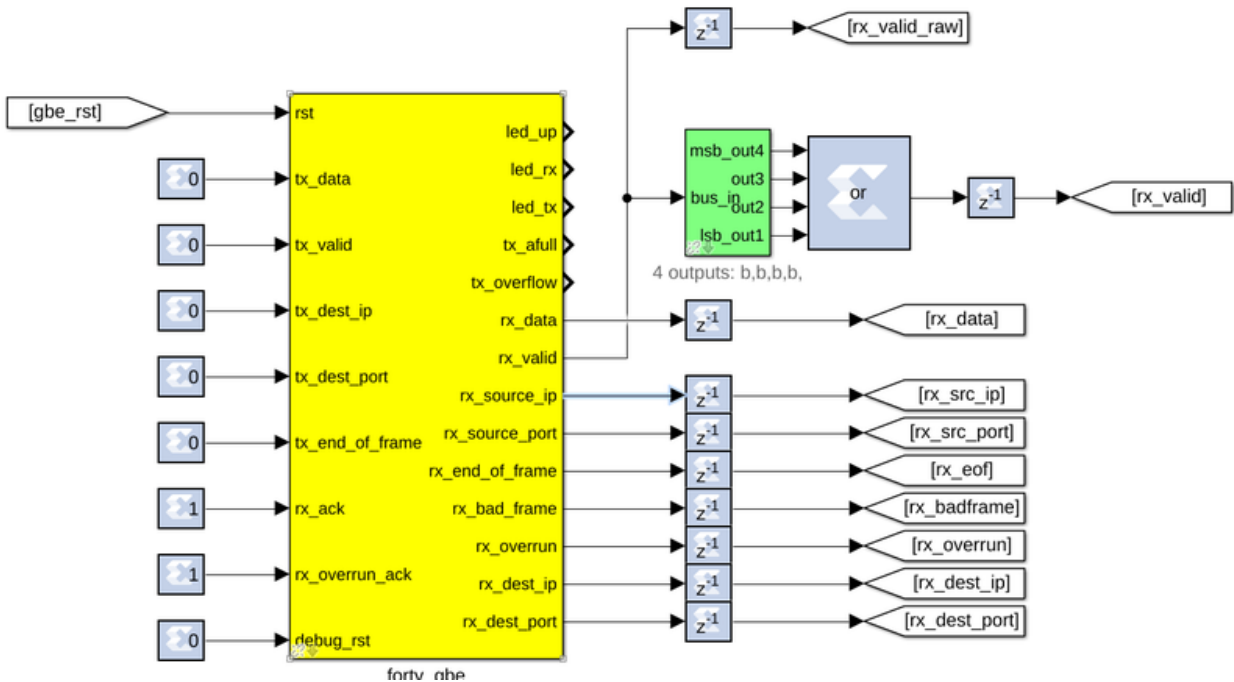
Rx Design

For the receiver design do the same as the previous design by dropping down an XSG block and the SKARAB platform block. Configure the clock rate to 230Mhz. We want this to be well above the clock rate of the transmit design so that we can handle the variable rate from the transmitter and not overflow our buffers.

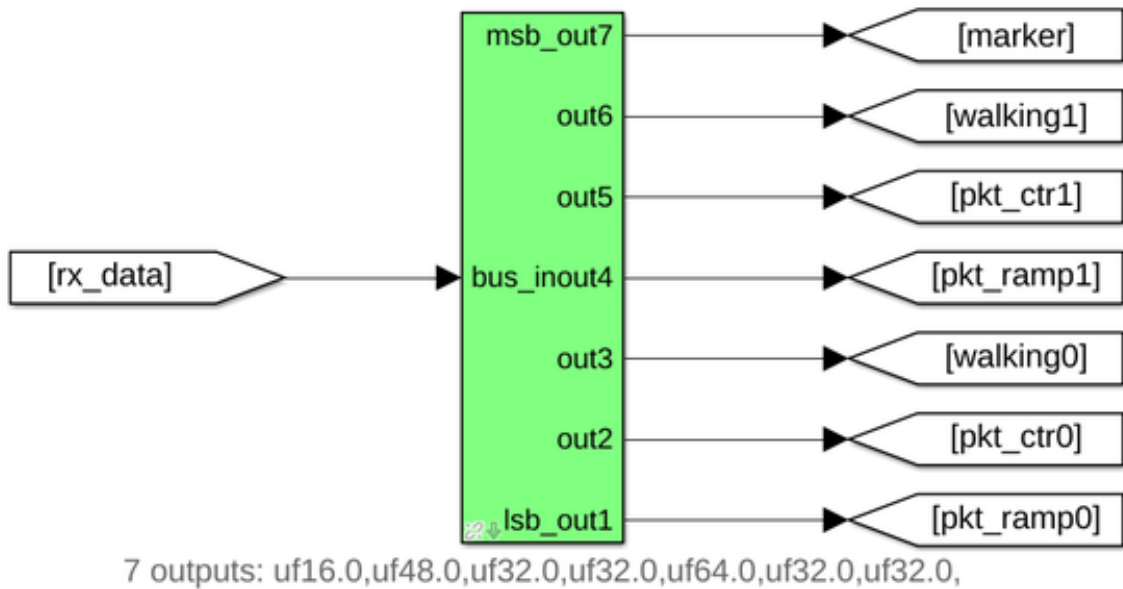
Again we have a control register which manages resets, enables and snap block triggering.



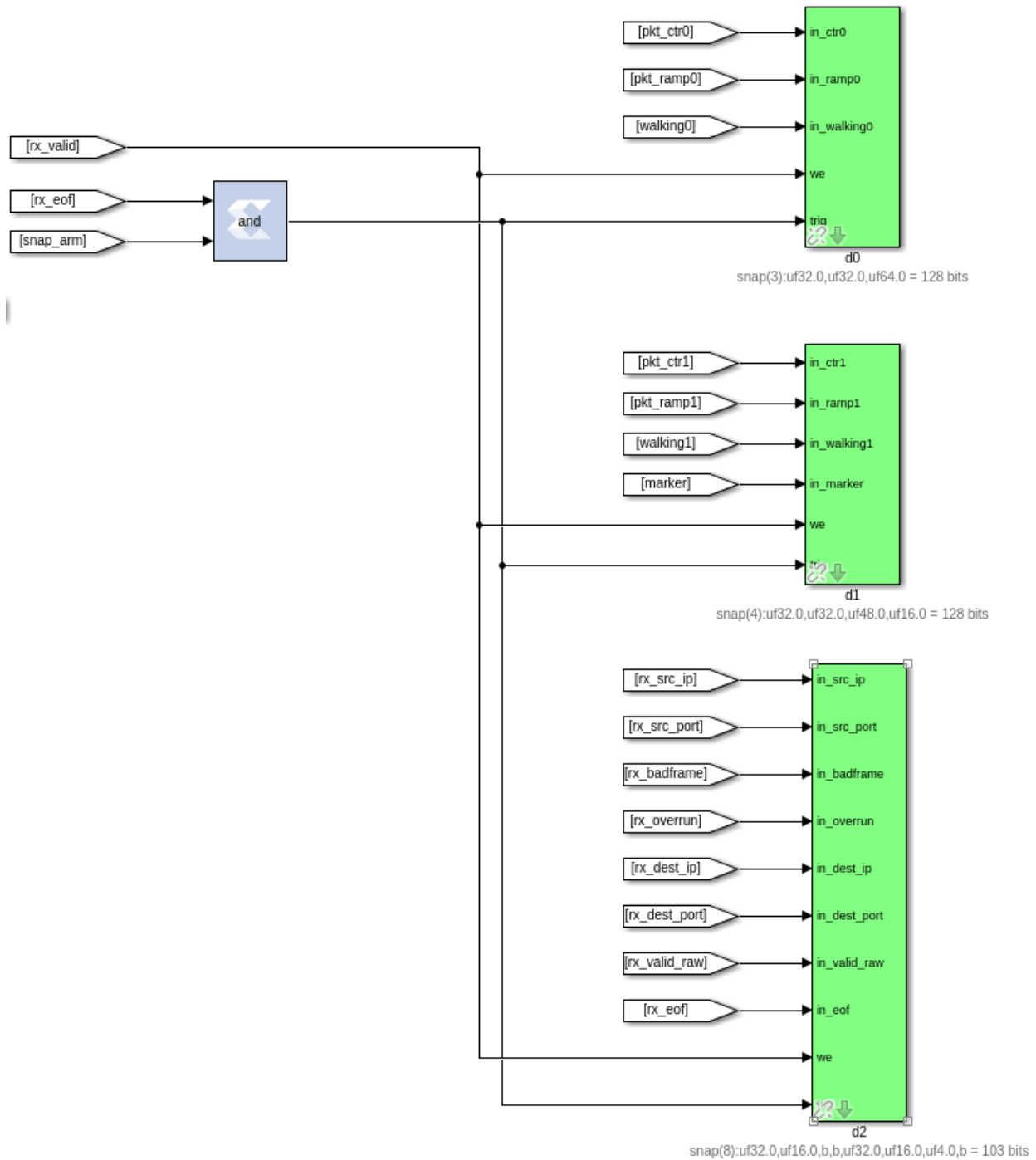
This is the receiving 40GbE. If the Tx-side is all tied to zero (0) this interface is not used. The Rx side is connected up to labels which are used to reduce the wires running around the design.



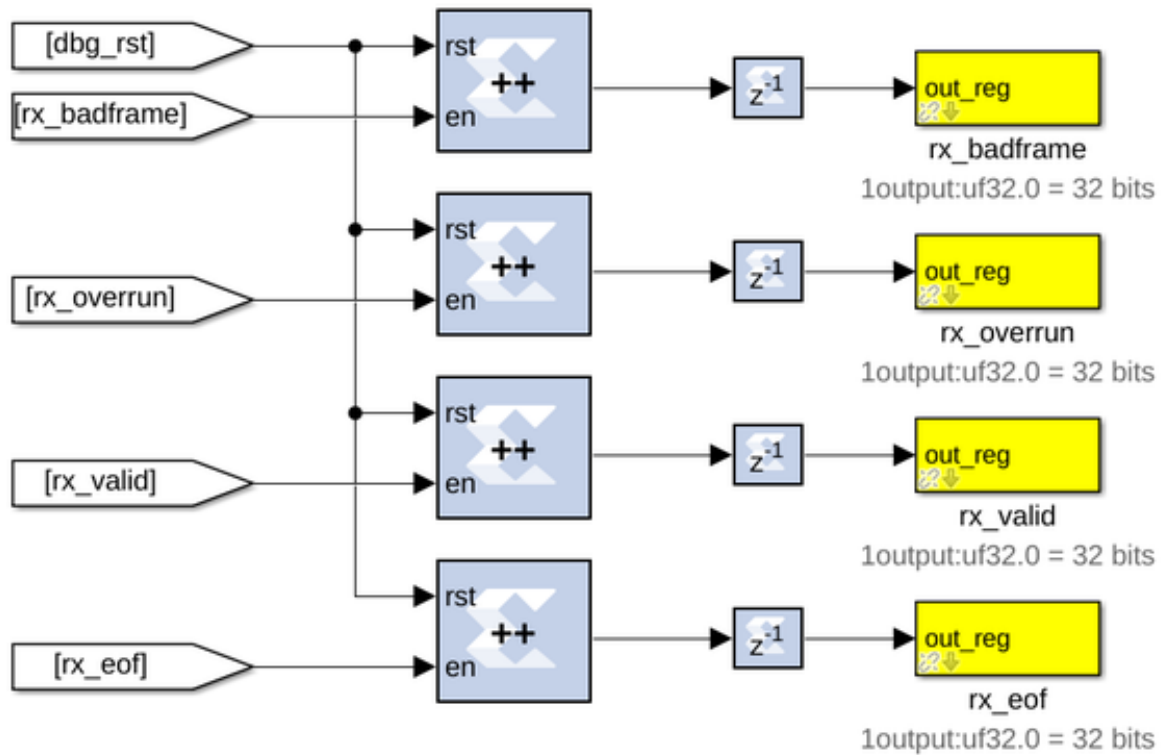
The following blocks split out the walking 1 and the ramps from the received data.



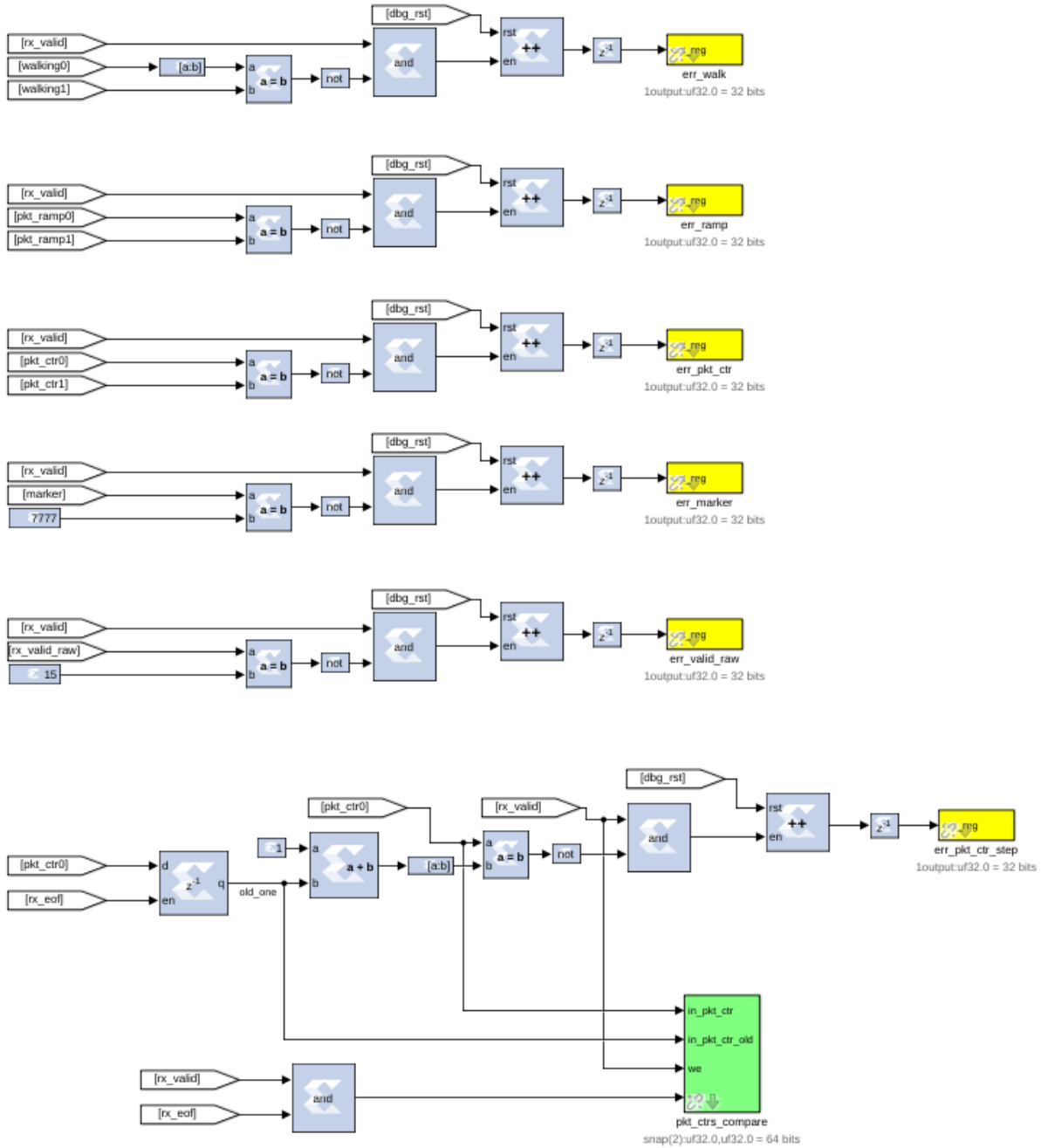
Here each of the split data are written into snap blocks. The snap blocks are triggered by the end-of-frame signal and the write enable is driven by the rx_valid signal.



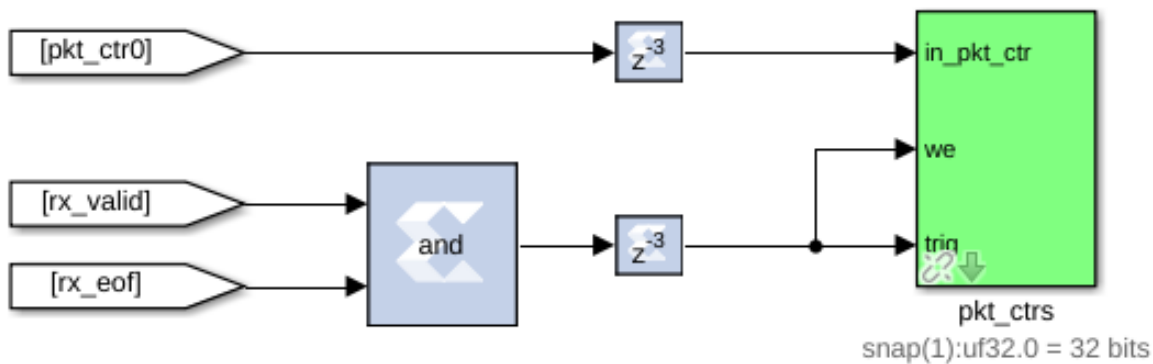
Here we have counters used to count any errors on the receiver's side. They are fed into software registers for access from software.



Here are more registers used for debugging; they count any errors in the expected data, the ramps and the walking 1.



This writes the packet header data into a snap block just to provide more debugging data.



Running the python script

Once you are finished examining the designs and feel that you have a good handle on them. Look through the python tx script. Try to figure out how to call the script with the correct parameters and files. You might have to [scp](#) the files to the control server, then run it and see what data you can get out. You can also start an ipython session to manually connect and run each of the commands. If you are running the tutorial during the workshop the facilitators should have the control server information up to view, or look [here](#). If you are running the tutorials elsewhere please familiarise yourself with your local setup/server(s) in order to run the tutorial.

The python script, `tut_40gbe.py`, does all the heavy lifting for the communication aspect of the tutorial. The script allows you to specify a few parameters and subsequently programs the boards, transmits and receives the test data.

Script arguments

Much like most python scripts, you can query the arguments via:

```
$ python tut_40gbe.py --help
usage: tut_40gbe_tx.py [-h] [--txhost TXHOST] [--rxhost RXHOST]
                      [--txfpg TXFPG] [--rxfpg RXFPG] [--pktsize PKTSIZE]
                      [--rate RATE] [--decimate DECIMATE] [-p] [-i]
                      [--loglevel LOG_LEVEL]
```

Script and classes **for** SKARAB Tutorial 2

optional arguments:

<code>-h, --help</code>	show this help message and exit
<code>--txhost TXHOST</code>	Hostname or IP for the TX SKARAB. (default:)
<code>--rxhost RXHOST</code>	Hostname or IP for the RX SKARAB. (default:)
<code>--txfpg TXFPG</code>	Programming file for the TX SKARAB. (default:)
<code>--rxfpg RXFPG</code>	Programming file for the RX SKARAB. (default:)
<code>--pktsize PKTSIZE</code>	Packet length to send (in words). (default: 160)
<code>--rate RATE</code>	TX bitrate, in Gbps. (default: 2.0)
<code>--decimate DECIMATE</code>	Decimate the datarate by this much. (default: -1)
<code>-p, --program</code>	Program the SKARABs (default: False)
<code>-i, --ipython</code>	Start IPython at script end. (default: False)
<code>--loglevel LOG_LEVEL</code>	log level to use, default None, options INFO, DEBUG, ERROR (default: INFO)

As per the info above, we can see that running the 40GbE tutorial script requires the following (compulsory) parameters:

- The IPs or Hostnames of SKARABs assigned to do the Transmitting and Receiving of data.
 - It is also worth mentioning that the SKARABs you intend on using to carry out this tutorial **must** have at least one QSFP+ cable plugged in to each board.
- Programming files for the Tx and Rx SKARABs. These fpg-files will be output after the build process is executed on Tx and Rx simulink models.
 - There are already-built versions of these images available in the working directory, namely `tut_40gbe_tx.fpg` and `tut_40gbe_rx.fpg`. Feel free to use these if you don't want to wait for the build process to complete.
- The **-p** or **-program** flag dicatates whether the programming files specified in the `-txfpg` and `-rx fpg` flags will be programmed to the board (using the method you learnt in the [previous tutorial](#))
 - I would suggest specifying this flag purely to minimise the 'setup' work you need to do to get the tutorial running.
 - You could run the tutorial script without specifying this flag, however that would require programming the two SKARABs with the associated fpg-files before running the script. Nothing major.

The other flags already have default values and don't need to be specified unless you want to, for example, test how different parameters change the behaviour of the system. An example of the script execution is show below:

```
$ python tut_40gbe.py --txhost skarab020201-01 --rxhost skarab020202-01 --txfpg tut_40gbe_tx.fpg --rx fpg tut_40gbe_rx.fpg -p
```

Of course, please do make sure you are in the correct directory holding the `tut_40gbe.py` script. Equally, substitute `tut_40gbe_tx.fpg` and `tut_40gbe_rx.fpg` for the paths to your generated fpg-files in the event you ran through the build process. These files should be in `tut_40gbe_tx/outputs/` and `tut_40gbe_rx/outputs/`. After executing the script as above you should see something resembling the following being printed to your terminal window:

```
INFO:root:Connecting to SKARABs
*
*
INFO:root:Programming SKARABs
*
*
INFO:root: Done programming TXer
*
*
INFO:root: Done programming RXer
skarab020202-01
INFO:root:Setting TX destination to 10.0.0.2.
INFO:root:Sending data at 1.970Gbps (0.177Ghz * 256 / 23)
INFO:root:Setting RX port.
INFO:root:Starting TX.
INFO:root:Some RX stats:
INFO:root:      valid: 7432640
INFO:root:      eof: 221779039
INFO:root:      badframe: 0
INFO:root:      overrun: 0
----- pkt_000 -----
ctr mark      walking_one  pkt_ctr      ramp
0 7777         32      47491         1
1 7777         64      47491         2
2 7777        128      47491         3
3 7777        256      47491         4
```

(continues on next page)

(continued from previous page)

4	7777	512	47491	5
*				
*				
*				
*				
*				

If you see any errors do make a note of them! Regardless, please ask if you have any questions, of which I am sure there will be many.

1.1.8 Tutorial 3: HMC Interface

AUTHORS: A. Isaacson and A. van der Byl

EXPECTED TIME: 2 hours

Introduction

In this tutorial, you will create a simple Simulink design which writes and reads to/from the HMC Mezzanine Card that is plugged into the SKARAB Mezzanine 0 MegaArray slot - refer to [SKARAB](#) for more information. In addition, we will learn to control the design remotely, using a supplied Python library for KATCP.

In this tutorial, a counter will be used to generate HMC test write data. Another counter will be used to read the test data from the HMC. The write and read data rate can be controlled by a software register to read and write every second clock cycle or read and write every clock cycle. This test can be used to demonstrate the throughput that the HMC can handle. This tutorial will also explain why the HMC read data needs to be reordered and shows a way of doing this using BRAM with BRAM read and write control.

Background

SKARAB boards have four MegaArray mezzanine slots. Mezzanine 3 is traditionally used for the QSFP+ Mezzanine Card, which makes provision for the 40GbE Interface functionality - refer to [SKARAB](#). The rest of the Mezzanine slots (0, 1 and 2) can be used for the HMC Mezzanine Card. This tutorial assumes that the HMC Mezzanine Card is fitted in mezzanine 0 slot. The SKARAB board can have up to three HMC Mezzanine Cards.

The HMC Mezzanine Card is fitted with a single Micron HMC MT43A4G80200 – 4GB 8H DRAM stack device. The HMC (Hybrid Memory Cube) Mezzanine Card, is a 4GB, serialised memory, which uses 16 x 10Gbps SERDES lanes to the FPGA per a mezzanine site. The HMC hardware works using the OpenHMC core designed by Juri Schmidt who works for the Computer Architecture Group at the University of Heidelberg [CAG](#) and [OpenHMC](#). The OpenHMC core is designed to be fully configurable, but this tutorial is using the configuration: 256 bit data transfer and 4 FLITS per a word.

The HMC yellow block makes provision for two links: Link 2 and Link 3. Each Link uses 8 x TX/RX GTH full duplex lines rated at 10Gbps each. This can be higher, but we are using it at this rate. This means that theoretically each link can handle 80Gbps throughput. The reality is that the HMC interface uses a protocol (FLIT = Floating Unit), which has a 128 bit (64 bit header and 64 bit tail), so at least half of that data is header and so the actual throughput will be in the region of 40Gbps. This has been tested successfully at 32 Gbps, with both links active. This tutorial will only write and read from link 3, so can handle a throughput between 32Gbps - 40Gbps. This tutorial tests the HMC with a write/read throughput of 29.44Gbps (pass) and a write/read throughput of 58.88Gbps (fail). The user will see the difference in the HMC data and status monitoring when the HMC read and write mechanism is successful and compare it to when it is unsuccessful.

The HMC memory has controller logic built in as well as stacked DRAM. This controller logic and stacked DRAM are divided into vaults. It is important to note that due to the DRAM refresh cycles when you request data (i.e. read)

from the HMC you will not always get the reads returned to you in the correct order. The HMC will only return data that has been requested only when the vault is available. If the vault is not available then it will handle other requests that are available. This out of order return allows the HMC to meet the higher throughput, but it does mean that the data read from the HMC needs to be reordered before using it. This tutorial will show you how to do this.

The HMC address scheme is user configurable, but also has set patterns and we are using one of the set patterns. The FLITs per a word (FPW) is also configurable, but the HMC yellow block is using 4 FLITs per a word. Each Flit is 128 bits, so 512 bits in total. The HMC is running at a 156.25MHz rate and the data width (after the header and tail have been removed) is 256bits. This gives a maximum data throughput of $256\text{bits} \times 156.25\text{MHz} = 40\text{Gbps}$.

It is important to note that each write uses 3 x FLITs and each read uses 1 x FLIT, so if you try to interleave read and writes then there will be a loss of throughput as the HMC devices are not being accessed nominally, as the FLITs per a word is set to 4. It is important to write and read simultaneously in order to achieve the correct throughput. It is not possible to read and write from the same address simultaneously and so there needs to be an offset between the read and write address. This tutorial uses an offset between the write and read addresses.

The address structure also influences the overall latency in the device. If you access a vault from a link that is not hardware linked to the vault then there will be additional latency to move from one vault to another. There are 16 vaults for this particular HMC device. All links can access all the vaults. The hardware linking looks as follows:

Link 0: vault 0, 1, 2, 3 (Link 0 is not available)

Link 1: vault 4, 5, 6, 7 (Link 1 is not available)

Link 2: vault 8, 9, 10, 11

Link 3: vault 12, 13, 14, 15

Therefore, using Link 2 and Link 3 try and use their corresponding vaults if the goal is to minimise latency, but in order to maximise throughput then circle the write and read address through all the vaults so that there will always be data available to process. This tutorial cycles through a limited address space to demonstrate this.

The HMC yellow block is using a 4GB device with 32 Byte Max Block size, which is shown in Table 13 below. The HMC firmware handles the request address bits 0-4 and 32-33. The HMC yellow block write and read address is 27 bits, which is mapped to request address bit 5-31 of the 32-Byte Max Block Size, which is shown in Table 13 below. Bit 5 is the LSB and bit 31 is the MSB.

Table 13: Default Address Map Mode Table

Request Address Bit	2GB			4GB		
	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size	32-Byte Max Block Size	64-Byte Max Block Size	128-Byte Max Block Size
33	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
32	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
31	Ignored	Ignored	Ignored	Ignored	Ignored	Ignored
30	DRAM[19]	DRAM[19]	DRAM[19]	DRAM[19]	DRAM[19]	DRAM[19]
29	DRAM[18]	DRAM[18]	DRAM[18]	DRAM[18]	DRAM[18]	DRAM[18]
28	DRAM[17]	DRAM[17]	DRAM[17]	DRAM[17]	DRAM[17]	DRAM[17]
27	DRAM[16]	DRAM[16]	DRAM[16]	DRAM[16]	DRAM[16]	DRAM[16]
26	DRAM[15]	DRAM[15]	DRAM[15]	DRAM[15]	DRAM[15]	DRAM[15]
25	DRAM[14]	DRAM[14]	DRAM[14]	DRAM[14]	DRAM[14]	DRAM[14]
24	DRAM[13]	DRAM[13]	DRAM[13]	DRAM[13]	DRAM[13]	DRAM[13]
23	DRAM[12]	DRAM[12]	DRAM[12]	DRAM[12]	DRAM[12]	DRAM[12]
22	DRAM[11]	DRAM[11]	DRAM[11]	DRAM[11]	DRAM[11]	DRAM[11]
21	DRAM[10]	DRAM[10]	DRAM[10]	DRAM[10]	DRAM[10]	DRAM[10]
20	DRAM[9]	DRAM[9]	DRAM[9]	DRAM[9]	DRAM[9]	DRAM[9]
19	DRAM[8]	DRAM[8]	DRAM[8]	DRAM[8]	DRAM[8]	DRAM[8]
18	DRAM[7]	DRAM[7]	DRAM[7]	DRAM[7]	DRAM[7]	DRAM[7]
17	DRAM[6]	DRAM[6]	DRAM[6]	DRAM[6]	DRAM[6]	DRAM[6]
16	DRAM[5]	DRAM[5]	DRAM[5]	DRAM[5]	DRAM[5]	DRAM[5]
15	DRAM[4]	DRAM[4]	DRAM[4]	DRAM[4]	DRAM[4]	DRAM[4]
14	DRAM[3]	DRAM[3]	DRAM[3]	DRAM[3]	DRAM[3]	Bank[3]
13	DRAM[2]	DRAM[2]	Bank[2]	DRAM[2]	Bank[3]	Bank[2]
12	DRAM[1]	Bank[2]	Bank[1]	Bank[3]	Bank[2]	Bank[1]
11	Bank[2]	Bank[1]	Bank[0]	Bank[2]	Bank[1]	Bank[0]
10	Bank[1]	Bank[0]	Vault[3]	Bank[1]	Bank[0]	Vault[3]
9	Bank[0]	Vault[3]	Vault[2]	Bank[0]	Vault[3]	Vault[2]
8	Vault[3]	Vault[2]	Vault[1]	Vault[3]	Vault[2]	Vault[1]
7	Vault[2]	Vault[1]	Vault[0]	Vault[2]	Vault[1]	Vault[0]
6	Vault[1]	Vault[0]	Byte[6], DRAM[2]	Vault[1]	Vault[0]	Byte[6], DRAM[2]
5	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]	Vault[0]	Byte[5], DRAM[1]	Byte[5], DRAM[1]
4	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]	Byte[4], DRAM[0]
3	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored	Byte[3] = Ignored
2	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored	Byte[2] = Ignored
1	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored	Byte[1] = Ignored
0	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored	Byte[0] = Ignored

More information on the HMC device (Rev D) and OpenHMC controller (Rev 1.5) can be found under the following repo (in the “hmc” folder):

[SKARAB Docs](#) (master branch)

Create a new model

Start Matlab and open Simulink (either by typing ‘simulink’ on the Matlab command line, or by clicking on the Simulink icon in the taskbar). A template is provided for this tutorial with a pre-created HMC reordering function, SKARAB XSG core config or platform block, Xilinx System Generator block and a 40GbE yellow block. Get a copy of this template and save it. Make sure the SKARAB XSG_core_config_block or platform block is configured for:

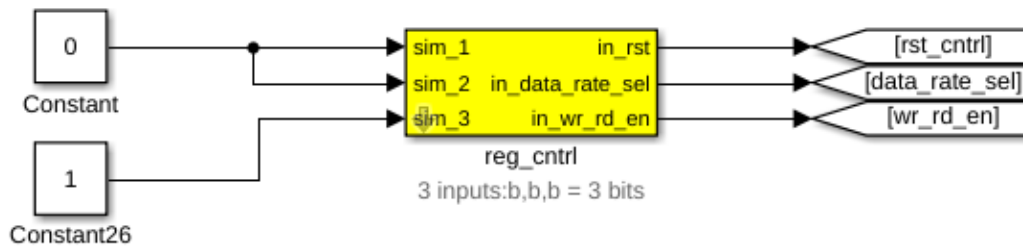
1. Hardware Platform: “SKARAB:xc7vx690t”
2. User IP Clock source: “sys_clk”
3. User IP Clock Rate (MHz): 230 (230MHz clock derived from 156.25MHz on-board clock). This clock domain is used for the Simulink design

The rest of the settings can be left as is. Click OK.

The 40GbE yellow block needs to be added as the SKARAB Board Support Package (BSP) is currently integrated in this block. If the 40GbE yellow block is not included then your Simulink will not compile as the SKARAB BSP will be missing.

Add control and reset logic

A very important piece of logic to consider when designing your system is how, when and what happens during reset. In this example we shall control our resets via a software register. We shall have one reset to reset the HMC design counters and trigger the data capture snap blocks. We shall have one data rate select, which will control/select the throughput through the HMC and we shall have one HMC write/read enable signal, which allows the user to disable/enable the process, so that the hmc read counter, write counter and hmc out counter all represent the same instant in time. Construct reset and control circuitry as shown below.



Add a software register

Use a software register yellow block from the CASPER XPS Blockset->Memory for the reg_cntrl block. Rename it to reg_cntrl. Configure the I/O direction to be From Processor. Attach two Constant blocks from the Simulink->Sources section of the Simulink Library Browser to the input of the software register and make the value 0 and 1 as shown above.

Add Goto Blocks

Add three Goto blocks from Simulink->Signal Routing. Configure them to have the tags as shown (rst_cntrl, data_rate_sel and wr_rd_en). These tags will be used by associated From (also found in Simulink->Signal Routing) blocks in other parts of the design. These help to reduce clutter in your design and are useful for control signals that are routed to many destinations. They should not be used a lot for data signals as it reduces the ease with which data flow can be seen through the system.

Add a write and read counter to generate test data for the HMC

Add Counter Blocks

Add four Counter blocks from Xilinx Blockset->Basic Elements and configure it to be unsigned, free-running, 9-bits, incrementing by 1 as shown below - the block parameters are the same for all counters. The first counter represents the write data, the second counter represents the write address, the third counter represents the read data and the fourth counter represents the read address as shown below.

Add Delay Blocks

Add the delay blocks from Xilinx Blockset->Basic Elements and configure it as shown below. The read enable is delayed by 256 clock cycles in order to prevent the read and write address from clashing. It also allows ample time for a write to occur before reading occurs. The HMC write enable and read enable signal are aligned in order to ensure that the HMC reading and writing happen concurrently.

Add Goto and From Blocks

Add Goto and From blocks from Simulink->Signal Routing as shown below. Configure them to have the tags as shown.

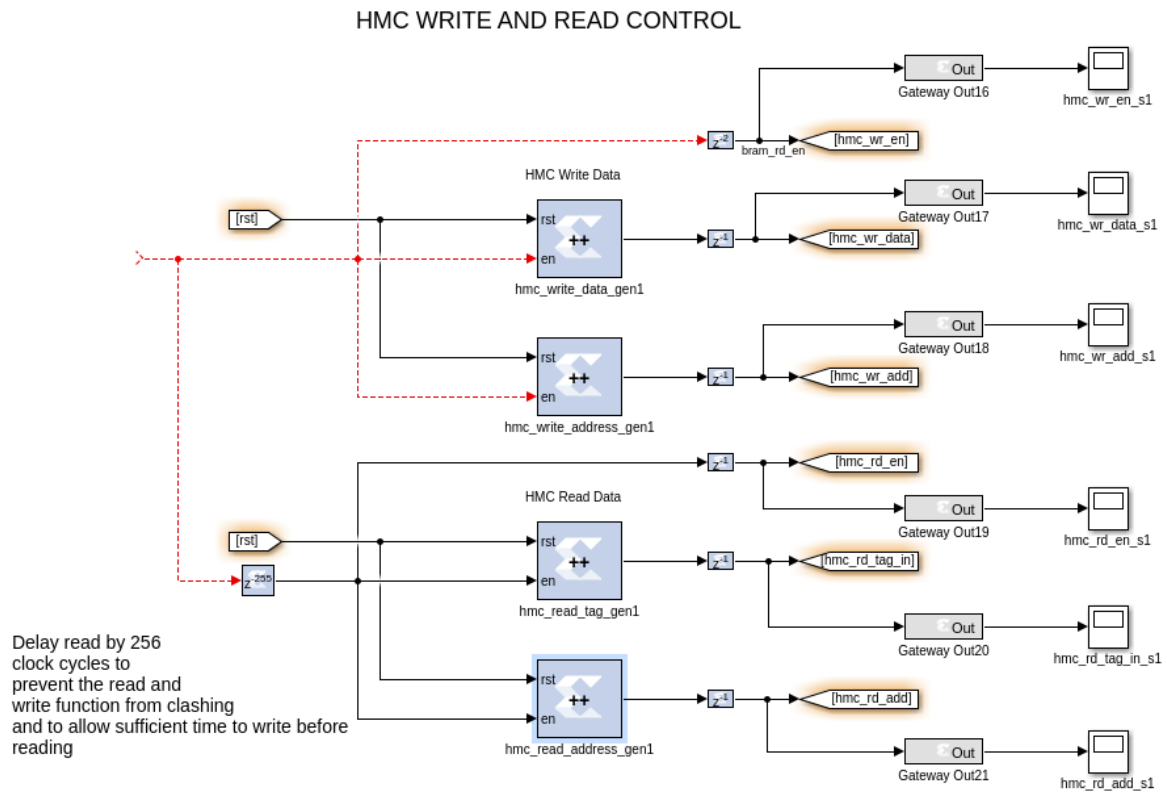
Add Gateway Out Blocks and Scopes

Add Gateway Out blocks from Xilinx Blockset->Data Types as shown below. Remember to disable the “translate into output port” check. The purpose of the gateway out block is because we are connecting Xilinx blocks to the Simulink scope. You don’t need a gateway, but a warning will be generated when you compile the simulink block (Ctrl+D).

Add Scopes from Simulink->Sinks, in order to visually display the signals after simulation as shown below.

In simulation this circuit will generate a write & read address and data counter from 0 to 511 and the counter will wrap around after 511 as it is only 9 bits. This will allow us to generate simple test data for the HMC in order to analyse the memory writing and reading process. If the data is the same as the address then it is easier to see what is going on. Also, if the counter overflows all the time then it is easier to compare the HMC write with the HMC read and the HMC reorder process.

The dotted red-lines represent the counter enable signal path and that is generated by the data rate control function in the section below.





Add functionality to control the write and read data rate

As mentioned earlier in this tutorial, it is impossible to perform HMC read and write at the full clock rate. This would mean writing/reading a 256 bit word at 230MHz (58.88Gbps), and the HMC firmware supports up to a maximum of 256 bit at 156.25MHz (40Gbps) data throughput. We thus want to limit the data rate, so that the HMC firmware FIFOs do not overflow. We thus add circuitry to limit the data rate as shown below. The logic that we have added below can either enable the HMC write and read counters every clock cycle or enable the HMC write and read counters every second cycle. There is a multiplexer, which is controlled via a software register that can select either rate for demonstration purposes.

Implement the function that performs the write and read data rate control as shown below:

Add a Counter Block

Add one Counter block from Xilinx Blockset->Basic Elements and configure it to be unsigned, free-running, 2-bits, incrementing by 1 as shown below. This counter is used to divide the data rate by 2 using the LSB of the counter.

Add Xilinx Constant Blocks

Add three Constant blocks from Xilinx Blockset->Basic Elements and configure it to be a 1 or a 0 as shown below. All constants must be boolean.

Add Slice Block

Add a Slice block from the Xilinx Blockset-> Basic Elements, as shown below. Configure it to select the least significant bit.

Add From Blocks

Add From blocks from Simulink->Signal Routing as shown below. Configure them to have the tags as shown below.

Add Xilinx Convert (cast) Block

Add a Xilinx Convert block from Xilinx Blockset-> Data Types. Configure it to be boolean.

Add Xilinx Bus Multiplexer (Mux) Block

Add a Xilinx Mux block from Xilinx Blockset-> Basic Elements. Configure it to have two inputs, 1 clock cycle latency and full output precision, as shown below.

Add Xilinx Logical Block

Add a Xilinx logical block from Xilinx Blockset-> Basic Elements. Configure it to have four inputs, no latency and full output precision, as shown below.

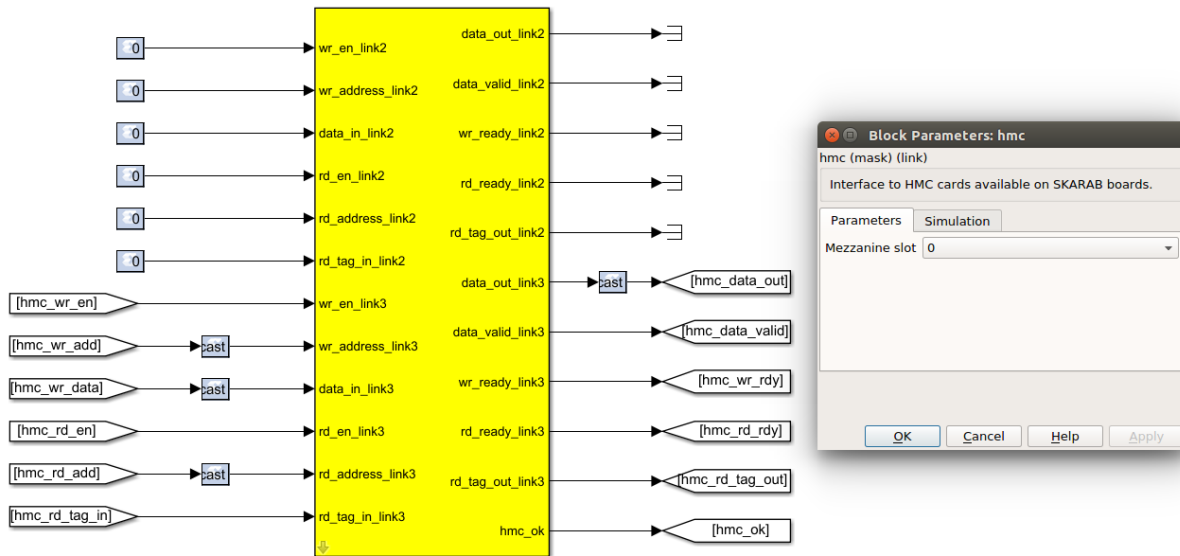
Add Gateway Out and To Workspace Block (Optional)

Add Gateway Out blocks from Xilinx Blockset->Data Types as shown below. Remember to disable the “translate into output port” check. The purpose of the gateway out block is because we are connecting Xilinx blocks to the Simulink workspace variable. You don’t need a gateway, but a warning will be generated when you compile the simulink block (Ctrl+D).

Add to To Workspace blocks from Simulink->Sinks as shown below. This captures all the simulation data to the Matlab workspace variable. This makes it easier to see if data is aligned than just looking at the scope display. This step is optional, but you are welcome to try it.

The dotted red lines indicates where it interfaces with the HMC write and read control functionality in the section above.

HMC YELLOW BLOCK

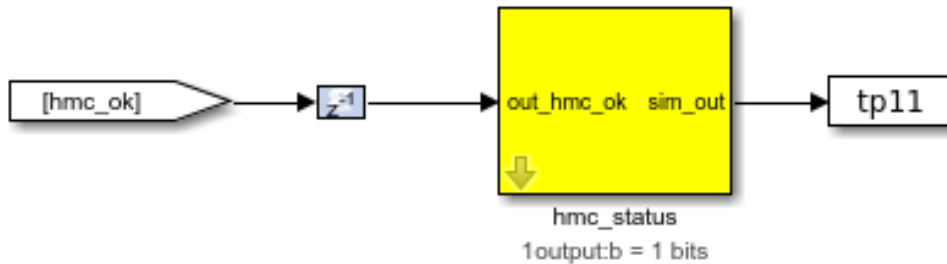


Add a register to provide HMC status monitoring

Add one yellow-block software register to provide the HMC status (1 bit). Name it as shown below. The register should be configured to send its value to the processor. Connect it to the HMC yellow block as shown below using GoTo blocks. A Convert (cast) block is required to interface with the 32 bit registers. Delay blocks are also required. To workspace blocks from Simulink->Sinks are attached to the simulation outputs of the software registers.

The HMC status is made up of the HMC OK flag. If there are any errors with the HMC initialisation, HMC POST (Power On Self Test), FLIT transactions or HMC ERRSTAT register then this flag will be set to "0". It takes a maximum of 1.2s for the HMC lanes to align and the initialisation process to complete. Once this is done then internally generated test data is written into the HMC. The data is then read out and compared with the data written in. If there is a match then POST passes and the internal POST OK flag is set to '1'. In this case, HMC initialisation done will be '1' when the initialisation is successful and the POST process has finished. The internal POST OK flag will only be set to '1' when the memory test is successful. Therefore, the user can only start writing and reading to/from the HMC when HMC Ok is set to '1'. If this flag is '0' then the HMC did not properly start up or data has been corrupted. Refer to the HMC Data Rate Control functionality above, which uses this flag to only start the write and read process when they are asserted.

The internal HMC receive FLIT protocol error status register (HMC ERRSTAT) is 7 bits. If any of these bits are '1' then this means an error has occurred. This should always be '0'. In order to decode what this error means there is a table in the HMC data sheet on page 48 Table 20.



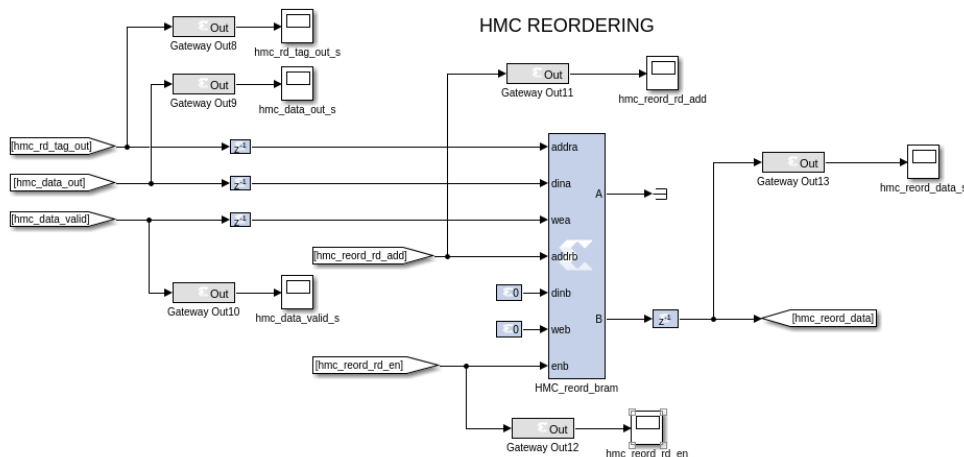
The HMC is connected to the wishbone interface and hence, it is possible to read back the internal HMC status registers using casperfpga - see below for how to do this.

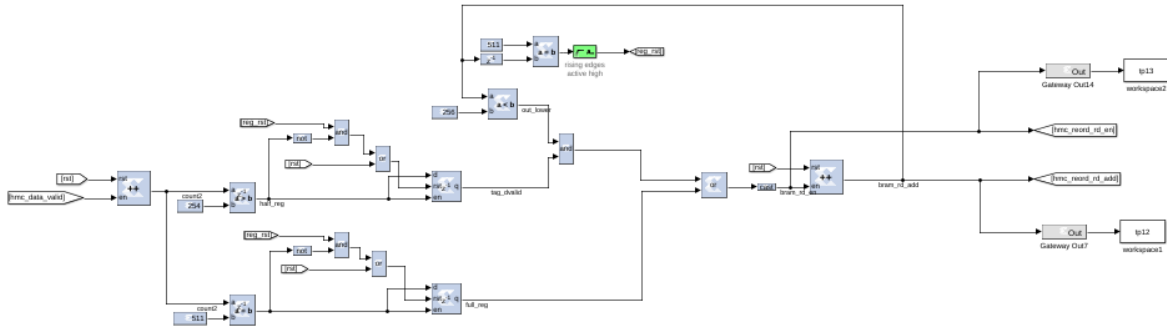
Implement the HMC reordering functionality

We will now implement logic to reorder the data that is read out of sequence from the HMC. This is critical, as the data is no use to us if it is out of sequence. This is already included in the template for this tutorial, so please use this functionality as is to save time. Some details are provided here for completeness.

The logic below looks complicated, but it is not. The HMC does not read back the data in the order it was requested due to how the HMC vaults operate and the DRAM refresh cycles. This makes the HMC readback undeterministic. The HMC reorder BRAM (512 Deep) reorders all the data read back from the HMC. This will synchronise the reorder readouts by using the read out tag as the write address of the reorder BRAM. It turns out through experience that the maximum delay can be in the order of 256 tags, when the data is requested. The function below does the following:

1. It ensures that the HMC has written at least 256 words into the reorder BRAM before reading out of the reorder BRAM.
2. It makes sure the read pointer does not exceed the write pointer i.e. do not read data that has not been written yet.
3. Once the read pointer reaches count 256 then it waits until the write pointer count is at 512 and then continues to read the rest of the reorder BRAM while the write pointer starts from 0 again. This prevents the write and read pointers from clashing. This is essentially a bank swopping control mechanism.



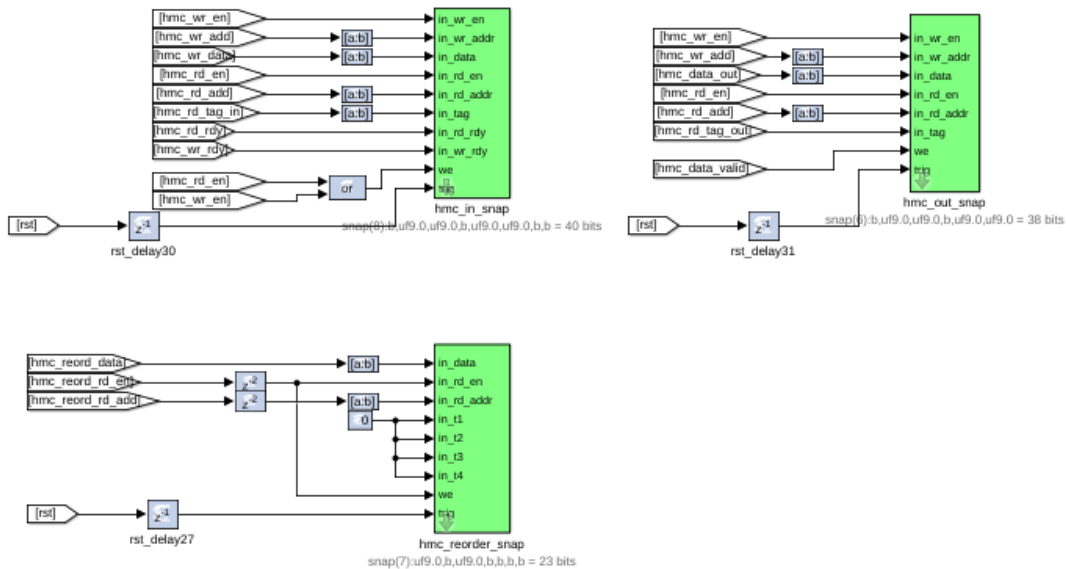


Buffers to capture HMC write, HMC read and HMC reordered read data

The HMC write data (input), HMC read data (output) and HMC reordered data need to be connected to bitfield snapshot blocks for data capture analysis (located in CASPER DSP Blockset->Scopes), as shown below. These blocks (hmc_in_snap, hmc_out_snap and hmc_reorder_snap) are identical internally. Using these blocks, we can capture data as it is written and compare it to the data we have read and finally to the data that has been reordered.

Bitfield snapshot blocks are a standard way of capturing snapshots of data in the CASPER tool-set. A bitfield snap block contains a single shared BRAM allowing capture of 128-bit words.

SNAP BLOCKS FOR DATA CAPTURE (HMC INPUT AND HMC OUTPUTS)



The ctrl register in a snap block allows control of the capture. The least significant bit enables the capture. Writing a rising edge to this bit primes the snap block for capture. The 2nd least most significant bit allows the choice of a trigger source. The trigger can come from an external source or be internal and immediately. The 3rd most least significant bit allows you to choose the source of the valid signal associated with the data. This may also be supplied externally or be immediately enabled.

The basic principle of the snap block is that it is primed by the user and then waits for a trigger at which point it captures a block of data and then waits to be primed again. Once primed the addr output register returns an address of 0 and will increment as data is written into the BRAMs. Upon completion the addr register will contain the final

address. Reading this value will show that the capture has completed and the results may be extracted from the shared BRAMs.

In the case of this tutorial, the arming and triggering is done via software. The trigger is the rst signal. The “we” signal on the snapshot blocks is the data valid signal. Configure and connect the snap blocks as shown above. The Convert (cast) blocks should all be to 9 bits. The delays should be as shown above, as this aligns the data correctly. The following settings should be used for the bitfield snapshot blocks: storage medium should be BRAM, number of samples (“2^?”) should be 13, Data width 64, all boxes unchecked except “use DSP48s to implement counters”, Pre-snapshot delay should be 0.

HMC status registers

We shall now look at some registers to monitor the progress of our HMC writing and reading. We shall be able to check how many HMC write and read requests were issued and compare it to actual data read out of the memory via registers. We shall be able to check if the HMC is handling the throughput for the writing and reading via registers.

Write status registers

- “hmc_wr_cnt” is attached to a counter that increments when the HMC write enable signal is asserted ‘1’. It keeps a count of the number of write requests.
- “hmc_empty_wr_cnt” is attached to a counter that will increment only when the HMC write enable signal and HMC write ready signal are asserted ‘1’. This is optional.
- “hmc_wr_err” is a register that allows us to check if the HMC is meeting the write throughput by incrementing a counter every time the write enable signal is asserted ‘1’ when the HMC write ready signal is deasserted ‘0’ i.e. the HMC yellow block is still busy reading from the FIFO and is not ready for more write data.

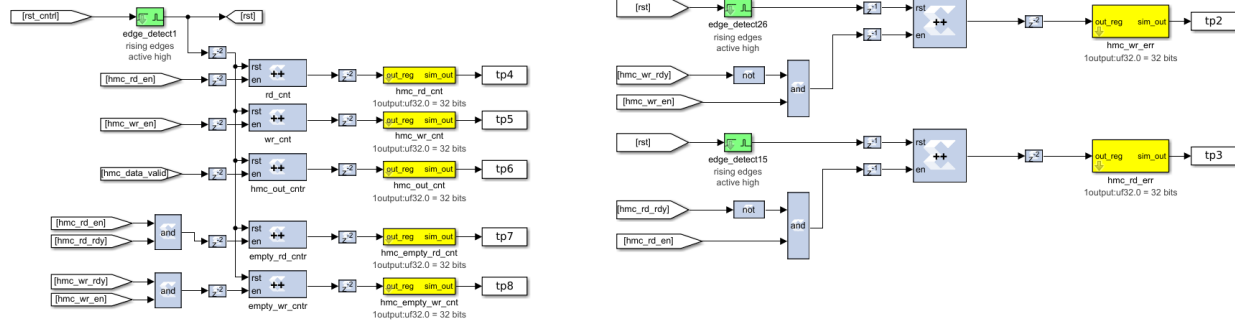
Read status registers

- “hmc_rd_cnt” is attached to a counter that increments when the HMC read enable signal is asserted ‘1’. It keeps a count of the number of read requests.
- “hmc_empty_rd_cnt” is attached to a counter that will increment only when the HMC read enable signal and HMC read ready signal are asserted ‘1’. This is optional.
- “hmc_out_cnt” is attached to a counter that will increment only when the HMC data valid signal is asserted ‘1’. It keeps a count of the number of valid read data coming from the memory.
- “hmc_rd_err” is a register that allows us to check if the HMC is meeting the read throughput by incrementing a counter every time the read enable signal is asserted ‘1’ when the HMC read ready signal is deasserted ‘0’ i.e. the HMC yellow block is still busy reading from the FIFO and is not ready for more write data.

From tag rst_cntrl should go through an edge_detect block (rising and active high) to create a pulsed rst signal, which is used to trigger and reset the counters in the design. This is located in CASPER DSP Blockset -> Misc.

It should look as follows when you have added all the relevant registers:

DEBUGGING AND STATUS MONITORING

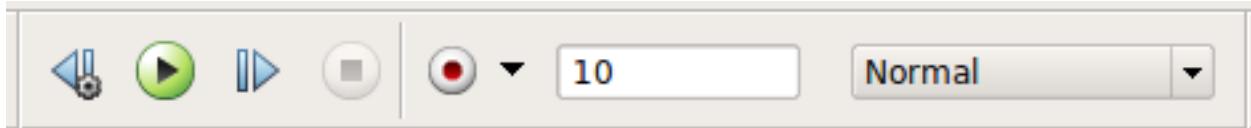


You should now have a complete Simulink design. Compare it with the complete hmc tutorial *.slx model provided to you before continuing if unsure.

Simulink Simulation

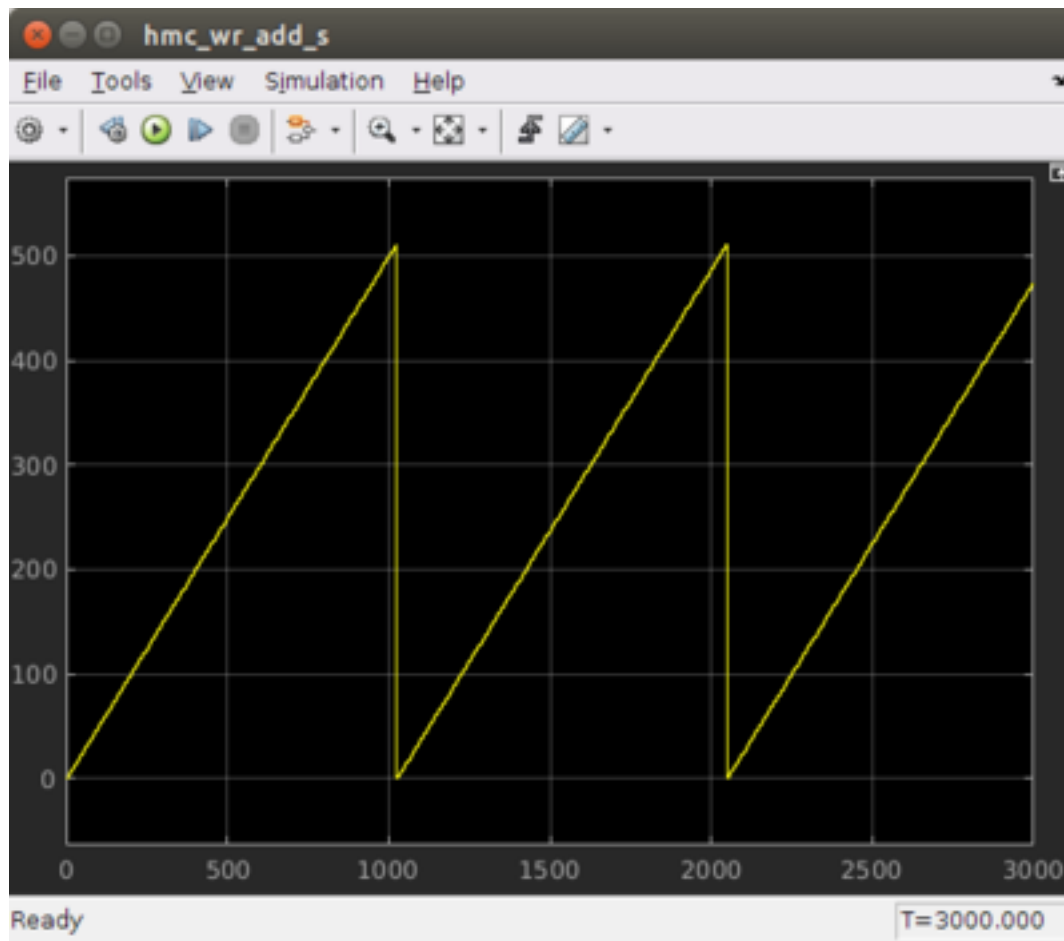
Press CTRL+D to compile the tutorial first and make sure there are no errors before simulating. If there are any errors then a diagnostic window will pop up and the errors can be addressed individually.

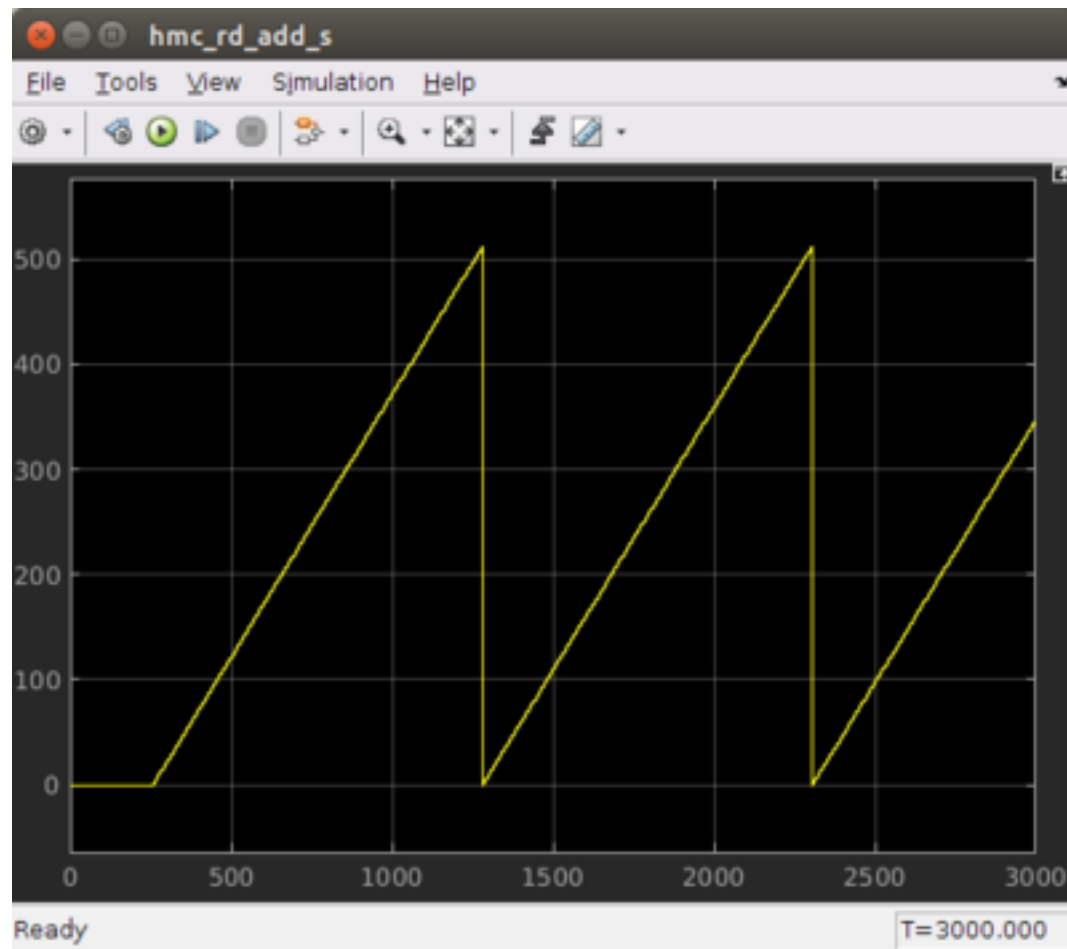
The design can be simulated with clock-for-clock accuracy directly from within Simulink. Set the number of clock cycles that you'd like to simulate and press the play button in the top toolbar. I would suggest make it 3000 in order to see a few cycles.

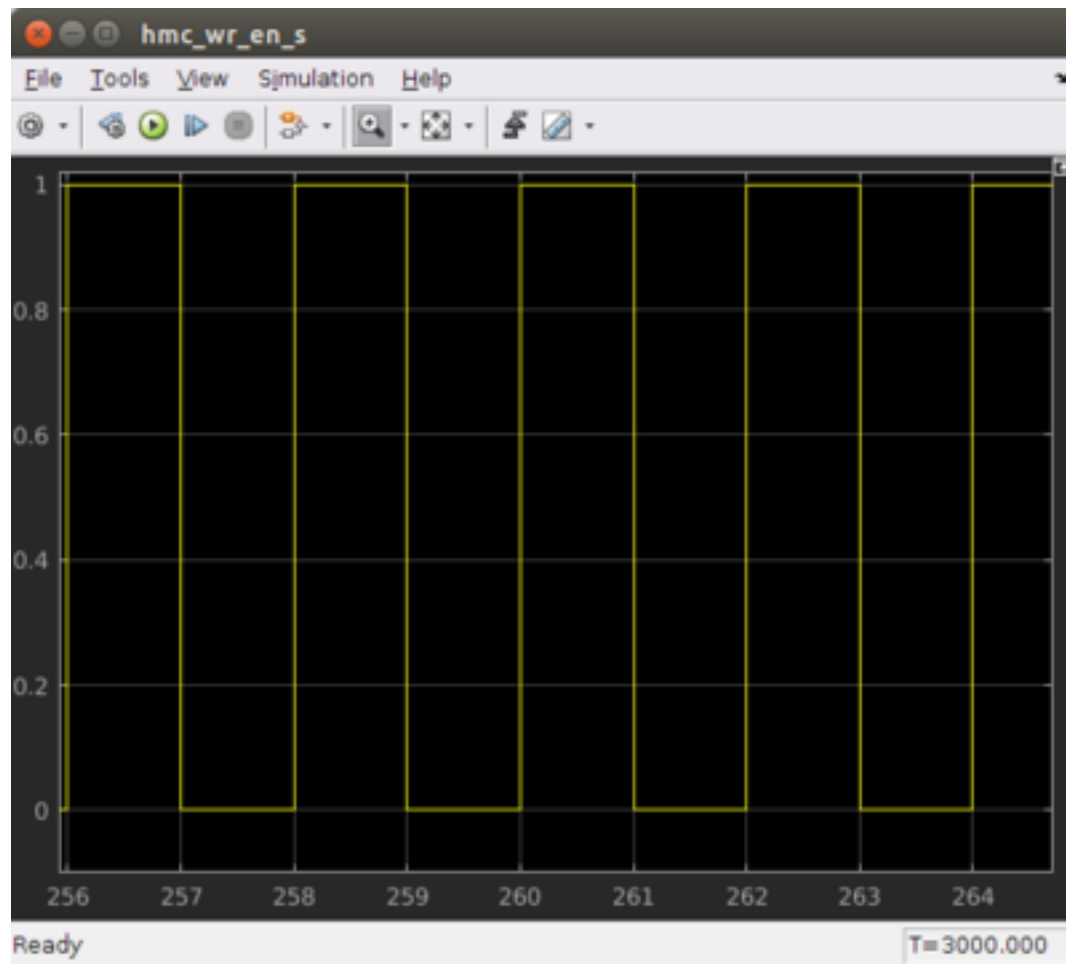


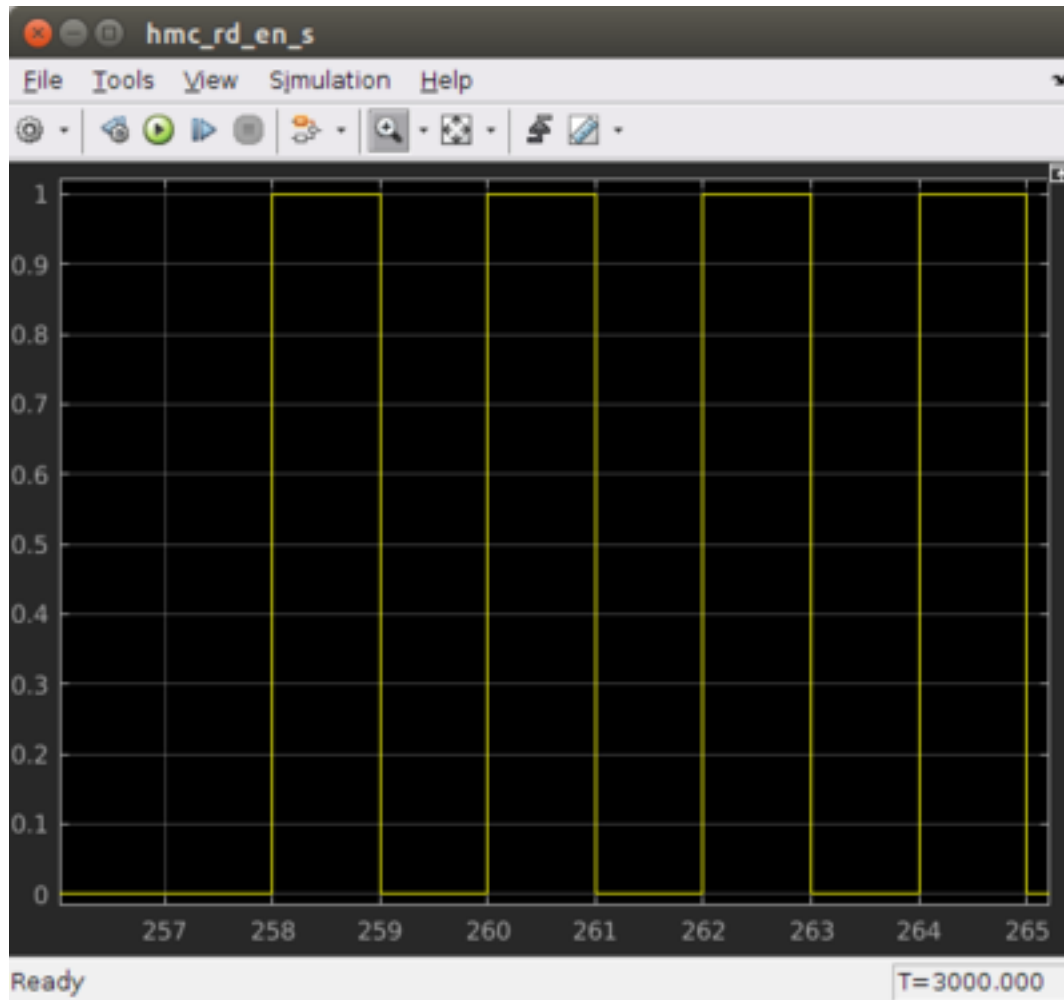
You can watch the simulation progress in the status bar in the bottom right. It may take a minute or two to simulate 3000 clock cycles.

Double-click on the scopes in the design to see what the signals look like on those lines. For example, the hmc_wr_add_s, hmc_rd_add_s, hmc_wr_en_s and hmc_rd_en_s scopes should look like below. You might have to press the Autoscale button to scale the scope appropriately. Note how the HMC read address is delayed by 256 clock cycles from the HMC write address. Note how the HMC write enable is aligned with the HMC read enable.

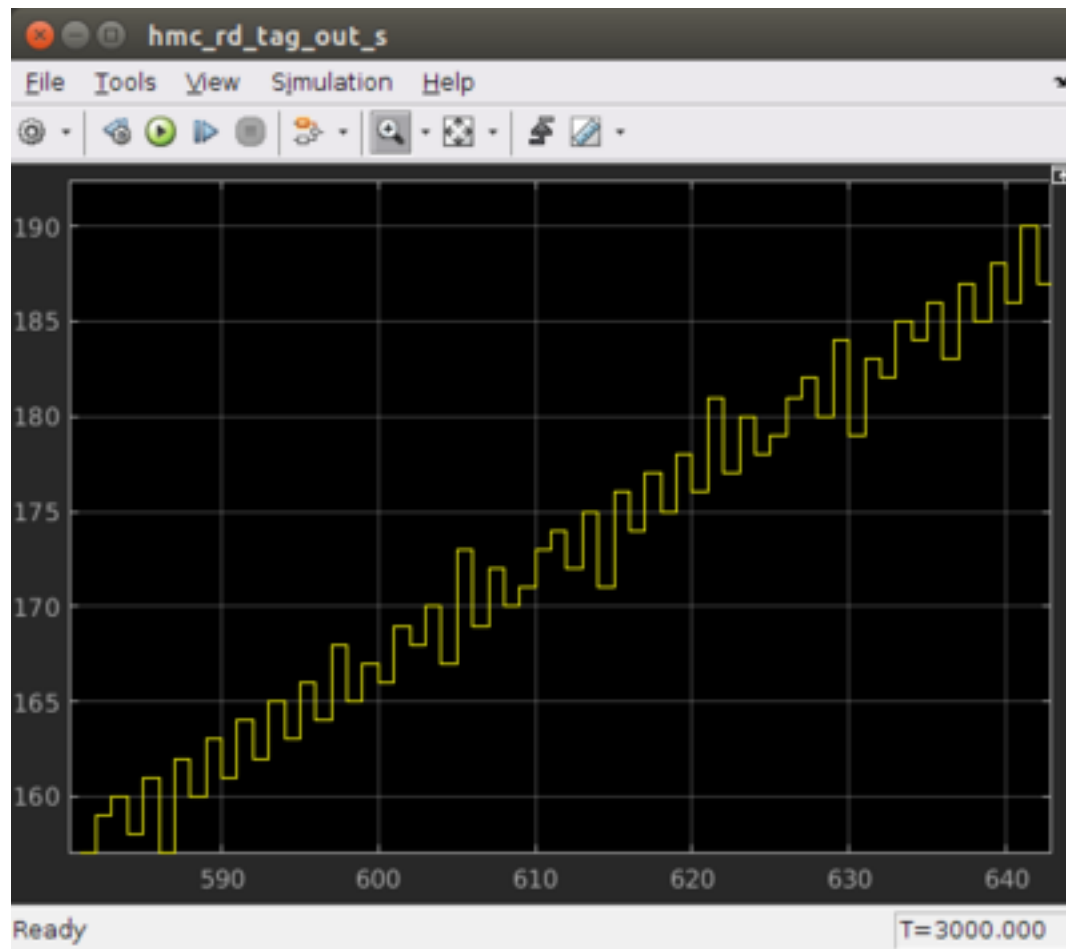


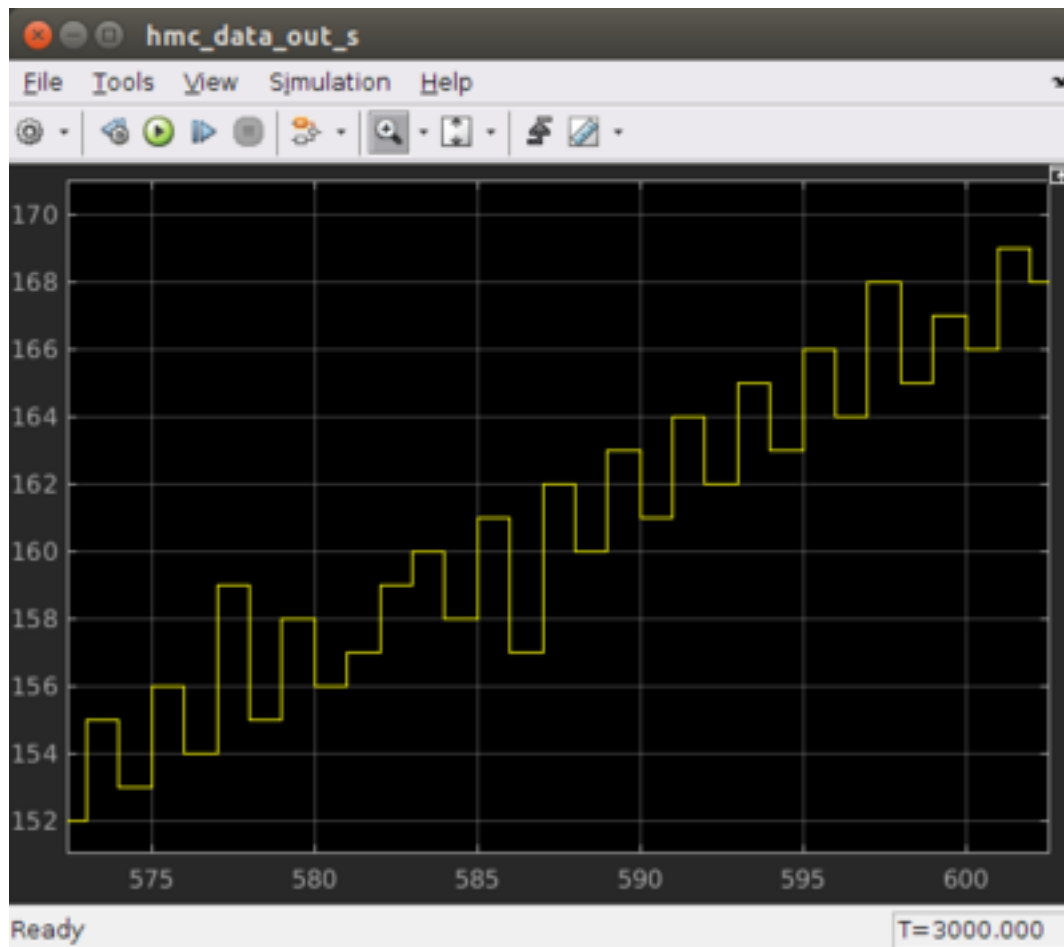




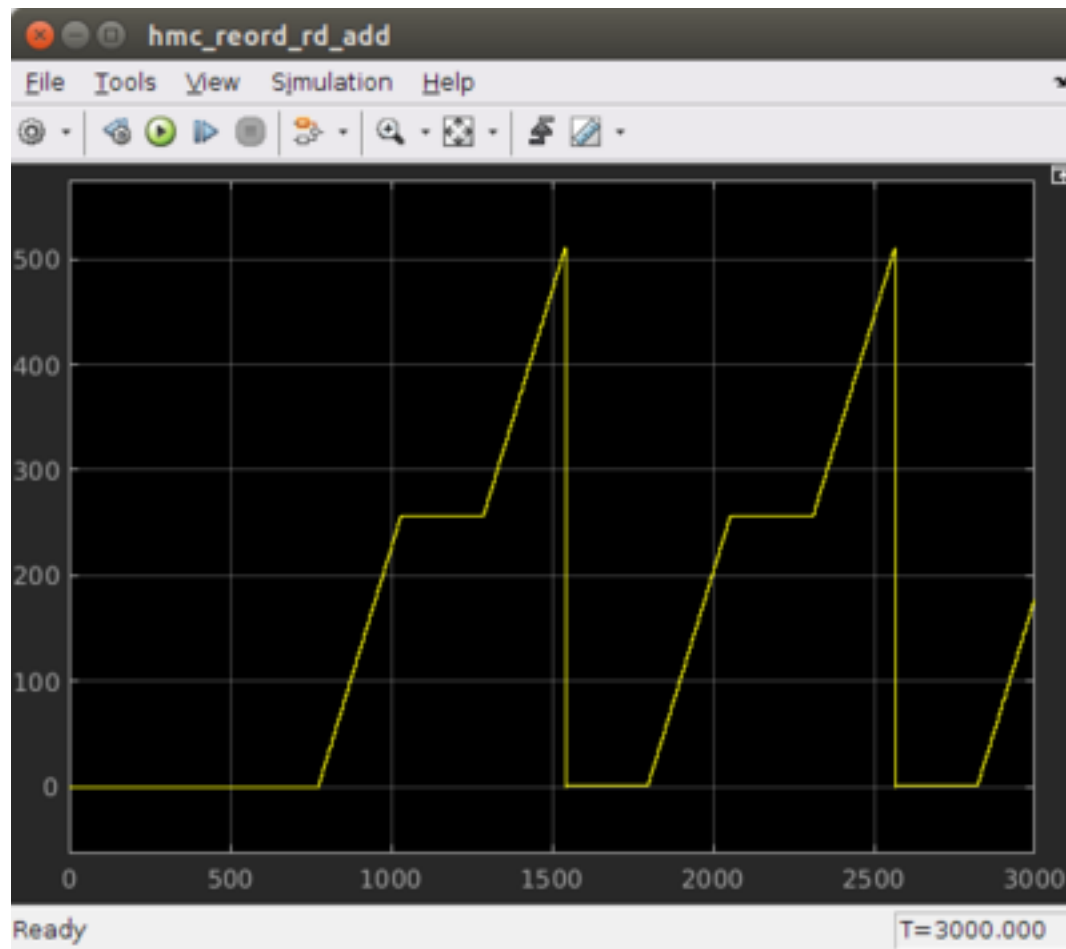


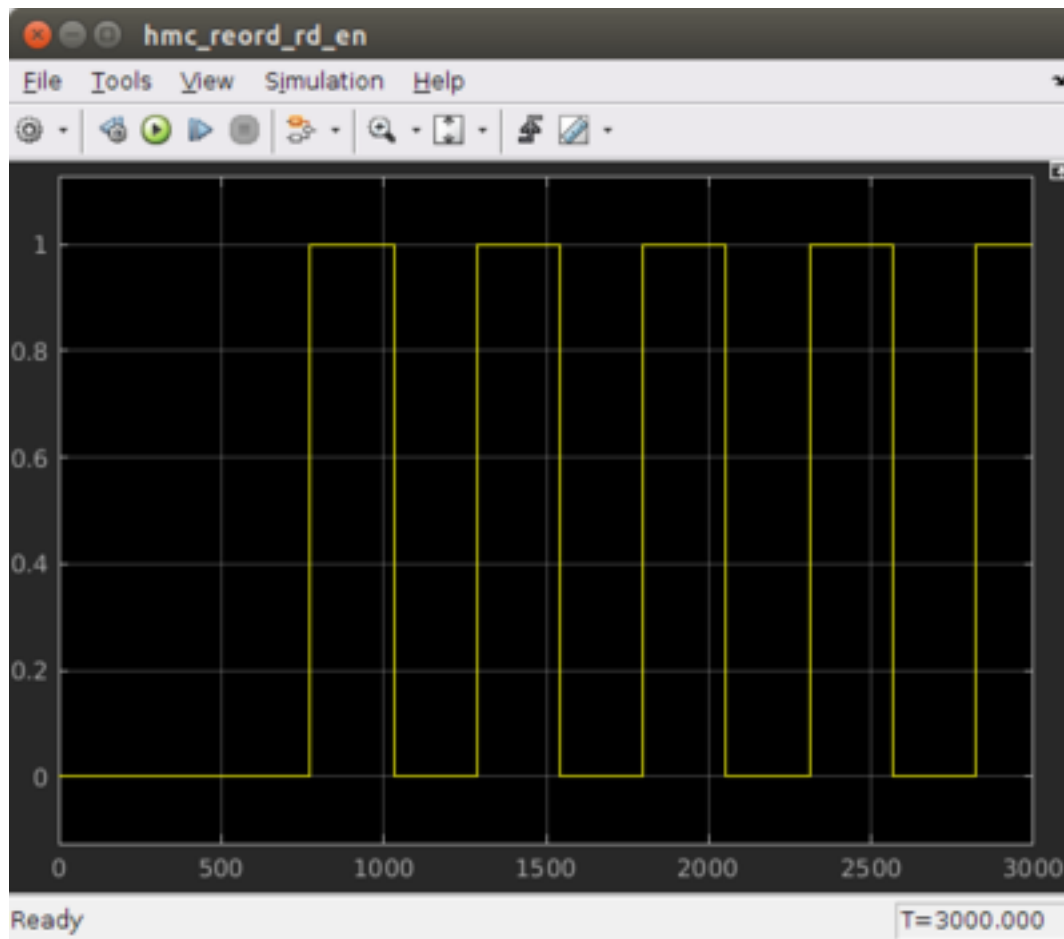
Double-click on the scopes at the output of the HMC yellow block. The `hmc_data_out_s` and `hmc_rd_tag_out_s` scopes should look like below. You might have to press the Autoscale button to scale the scope appropriately. Note how the HMC read tag output data and the HMC read data output are out of sequence. This data is useless to us in this form. It needs to be reordered.

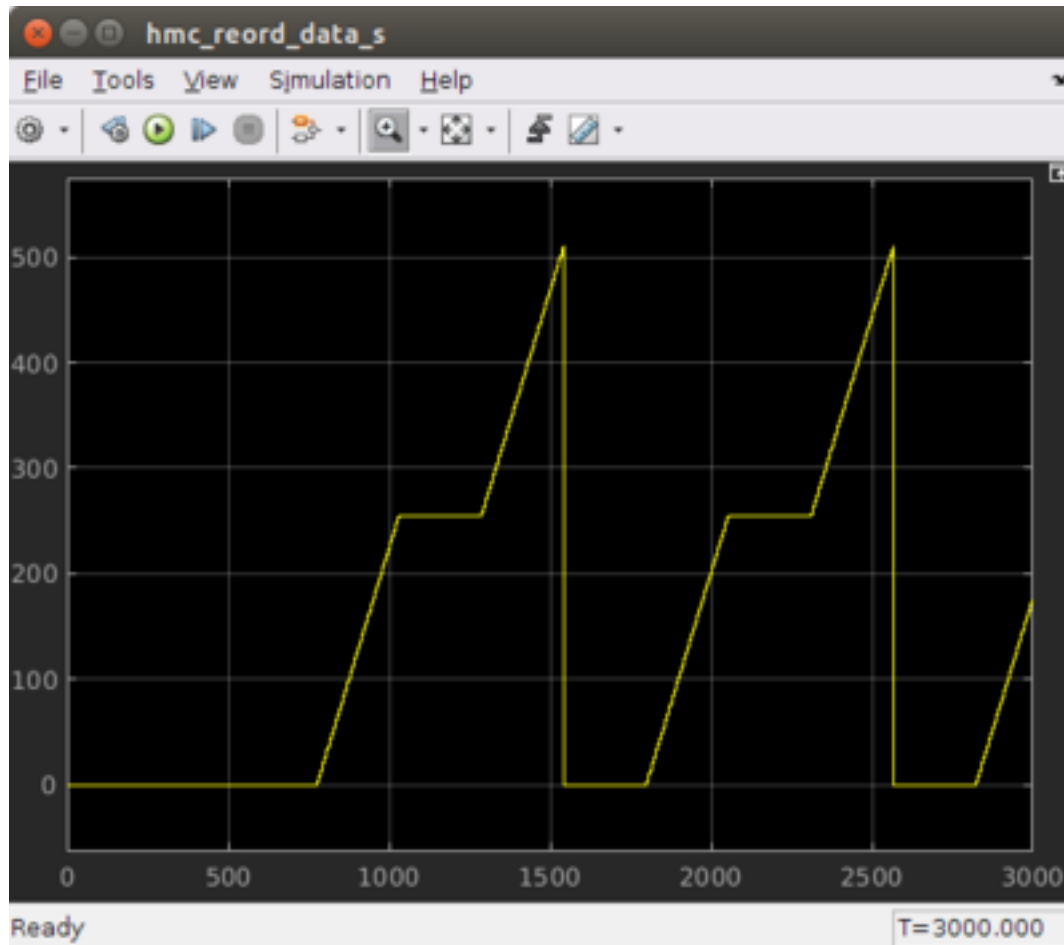




Finally let's double-click on the scopes at the output of the HMC reordering function. The `hmc_reord_rd_add_s`, `hmc_reord_rd_en_s` and the `hmc_reord_data_s` scopes should look like below. You might have to press the Autoscale button to scale the scope appropriately. Note how the data increases in a linear ramp and then stays at 255 for a period of time and then begins to increase in a linear ramp until it reaches 511 and then resets. If you compare with the HMC reorder read enable signal then the data is actually still a ramp, as not all of the data is valid. This makes sense as the HMC reordering logic will only read from the BRAM once the write pointer is at address 255 and then again when the write pointer is at 511. This prevents the write and read pointers from clashing as explained above.







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

Compilation

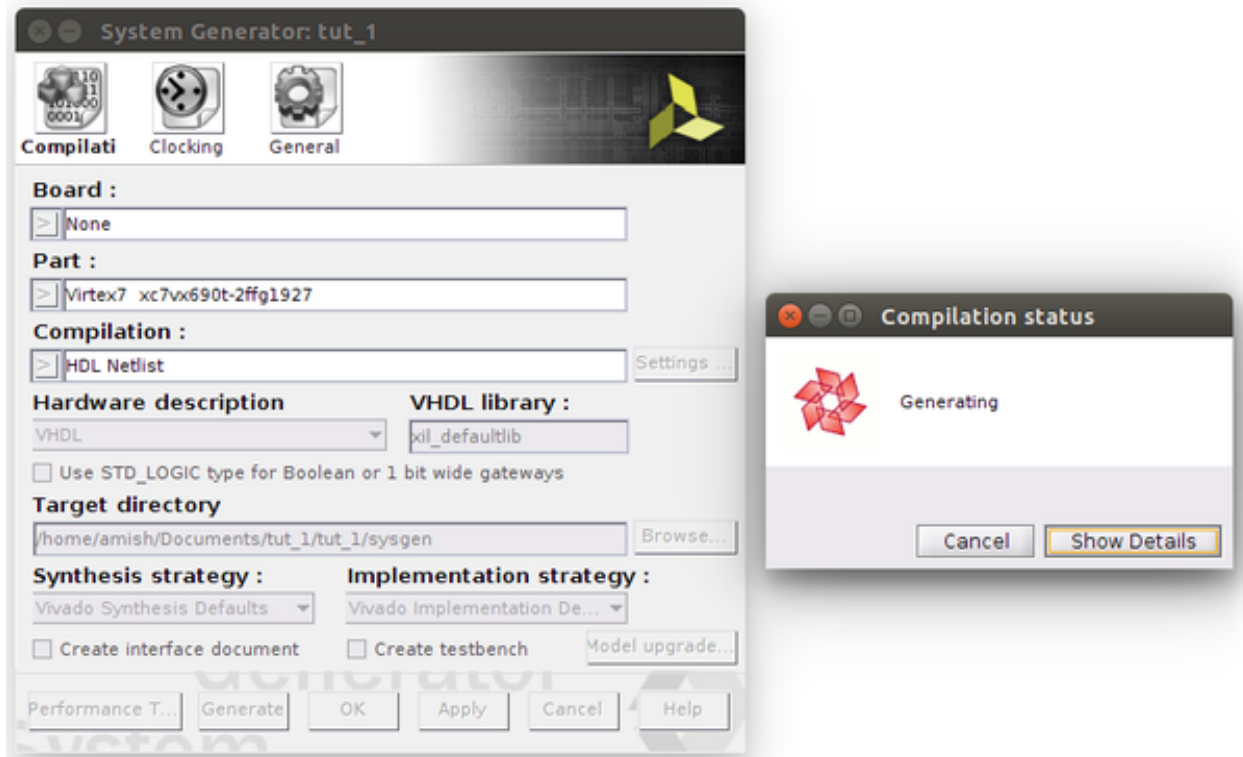
It is now time to compile your design into a FPGA bitstream. This is explained below, but you can also refer to the Jasper How To document for compiling your toolflow design. This can be found in the ReadtheDocs mlib_devel documentation link:

<https://casper-toolflow.readthedocs.io>

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window:

```
jasper
```

This will run the process to generate the FPGA bitstream and output Vivado compile messages to the MATLAB Command Line window along the way. During the compilation and build process Vivado's system generator will be run, and the windows below should pop up with the name of your slx file in the window instead of tut_1. The same applies below in the output file path - tut_1 will be replaced with the name of your slx file. In my case it is "tut_hmc".



Execution of this command will result in an output .bof and .fpg file in the 'outputs' folder in the working directory of your Simulink model. Note: Compile time is approximately 45-50 minutes, so a pre-compiled binary (.fpg file) is made available to save time.

```
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/
myproj/ outputs/ sysgen/
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/outputs/
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$ ls -lt
total 8336
-rw-rw-r-- 1 amish amish 1249412 Jul 25 14:18 tut_1_2017-7-25_1355.fpg
-rw-rw-r-- 1 amish amish 7280615 Jul 25 14:18 tut_1_2017-7-25_1355.bof
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$
```

Programming the FPGA

Reconfiguration of the SKARAB's SDRAM is done via the casperfpga python library. The casperfpga package for python, created by the SA-SKA group, wraps the Telnet commands in python and is used in the CASPER community. We will focus on programming and interacting with the FPGA using this method.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration, but if you are not working from the lab then refer to the How To Setup CasperFpga Python Packages document for setting up the python libraries for casperfpga. This can be found in the "casperfpga" repo wiki (to be deprecated) located in GitHub and the ReadtheDocs casperfpga documentation link:

[casperfpga README.md](#)

<https://casper-toolflow.readthedocs.io>

Copy your .fpg file to your NFS server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server.

```
scp path_to/your/model_folder/your_model_name/outputs/your_fpgfile.fpg user@server:/
↳path/to/test/folder/
```

Connecting to the board

SSH into the server that the SKARAB is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
ipython
```

Now import the fpga control library. This will automatically pull-in the KATCP library and any other required communications libraries.

```
import casperfpga
```

To connect to the SKARAB we create an instance of the SKARAB board; let's call it fpga. The wrapper's fpgaclient initiator requires just one argument: the IP hostname or address of the SKARAB board.

```
fpga = casperfpga.CasperFpga('skarab_name or ip_address')
```

The first thing we do is configure the FPGA.

```
fpga.upload_to_ram_and_program('your_fpgfile.fpg')
```

All the available/configured registers can be displayed using:

```
fpga.listdev()
```

The FPGA is now configured with your design. The registers can now be read back. For example, the HMC status register can be read back from the FPGA by using:

```
fpga.read_uint('hmc_status') or fpga.registers.hmc_status.read_uint();
```

The value returned should be 1, which indicates that the HMC has successfully completed initialisation, POST OK passes and data is not corrupted.

If you need to write to the reg_cntrl register then do the following:

```
fpga.registers.reg_cntrl.write(data_rate_sel= False), where data_rate_sel = False_
↳(29.44Gbps), data_rate_sel = True (58.88Gbps)

fpga.registers.reg_cntrl.write(rst = 'pulse'), this creates a pulse on the rst signal

fpga.registers.reg_cntrl.write(wr_rd_en= True) , where wr_rd_en = False (disable HMC_
↳write/read), wr_rd_en = True (Enable the HMC write/read)
```

You can read back the HMC mezzanine site, HMC die revisions and internal status registers by doing the following:

```
fpga.hmcs.hmc.mezz_site, this returns the mezzanine site that fpga.hmcs.hmc is
↳connected to

fpga.hmcs.hmc.get_hmc_revision(), this reads back the HMC firmware and product
↳revisions

fpga.hmcs.hmc.get_hmc_status(), this reads back the HMC status of the OpenHMC
↳Controller - refer to OpenHMC controller document to understand the status bits
↳located under "hmc" in [SKARAB Docs] (https://github.com/casper-astro/casper-
↳hardware/tree/master/FPGA_Hosts/SKARAB/docs) (master branch)
```

Manually typing these commands by hand will be cumbersome, so it is better to create a Python script that will do all of this for you. This is described below.

Running a Python script and interacting with the FPGA

A pre-written python script, "tut_hmc.py" is provided. The code within the python script is well commented and won't be explained here. The user can read through the script in his/her own time. In summary, this script programs the fpga with your compiled design (.fpg file), writes to the control registers, initiates the HMC write & read process, reads back the HMC snap shot captured data and status registers while displaying them to the screen for analysis. In order to run this script you will need to edit the file and change the target SKARAB IP address and the *.fpg file, if they are different. The script is run using:

```
python tut_hmc.py
```

If everything goes as expected, you should see a whole bunch of text on your screen - this is the output of the snap block and status register contents.

Analysing the Display Data

You should see something like this:

```
user@server:~$ python tut_hmc.py
connecting to SKARAB...
done
programming the SKARAB...
done
arming snapshot blocks...
done
triggering the snapshots and reset the counters...
done
enabling the HMC write and read process...
done
reading the snapshots...
done
disabling the HMC write and read process...
done
reading back the status registers...
hmc rd cnt: 55527004
hmc wr cnt: 55527004
hmc out cnt: 55527004
hmc wr err: 0
hmc rd err: 0
hmc status: 1
```

(continues on next page)

(continued from previous page)

```

rx crc err cnt: 0
hmc error status: 0
done
Displaying the snapshot block data...
HMC SNAPSHOT CAPTURED INPUT
-----
Num wr_en wr_addr wr_data wr_rdy rd_en rd_addr rd_tag rd_rdy
[0] 1 1 1 1 0 0 0 1
[1] 1 2 2 1 0 0 0 1
[2] 1 3 3 1 0 0 0 1
[3] 1 4 4 1 0 0 0 1
[4] 1 5 5 1 0 0 0 1
[5] 1 6 6 1 0 0 0 1
[6] 1 7 7 1 0 0 0 1
[7] 1 8 8 1 0 0 0 1
[8] 1 9 9 1 0 0 0 1
[9] 1 10 10 1 0 0 0 1
[10] 1 11 11 1 0 0 0 1
....
[589] 1 78 78 1 1 462 462 1
[590] 1 79 79 1 1 463 463 1
[591] 1 80 80 1 1 464 464 1
[592] 1 81 81 1 1 465 465 1
[593] 1 82 82 1 1 466 466 1
[594] 1 83 83 1 1 467 467 1
[595] 1 84 84 1 1 468 468 1
[596] 1 85 85 1 1 469 469 1
[597] 1 86 86 1 1 470 470 1
[598] 1 87 87 1 1 471 471 1
[599] 1 88 88 1 1 472 472 1
HMC SNAPSHOT CAPTURED OUTPUT
-----
Num hmc_read_tag_out hmc_data_out
[1] 1 1
[2] 2 2
[3] 3 3
[4] 4 4
[5] 5 5
[6] 6 6
[7] 7 7
[8] 8 8
[9] 9 9
[10] 10 10
[11] 12 12
[12] 11 11
[13] 13 13
....
[588] 75 75
[589] 77 77
[590] 78 78
[591] 79 79
[592] 80 80
[593] 81 81
[594] 82 82
[595] 83 83
[596] 84 84
[597] 85 85

```

(continues on next page)

(continued from previous page)

```

[598] 86 86
[599] 87 87
HMC REORDER SNAPSHOT CAPTURED OUTPUT
-----
Num rd_en rd_addr data_out
[1] 1 1 1
[2] 1 2 2
[3] 1 3 3
[4] 1 4 4
[5] 1 5 5
[6] 1 6 6
[7] 1 7 7
[8] 1 8 8
[9] 1 9 9
[10] 1 10 10
[11] 1 11 11
[12] 1 12 12
[13] 1 13 13
[14] 1 14 14
[15] 1 15 15
....
[588] 1 76 76
[589] 1 77 77
[590] 1 78 78
[591] 1 79 79
[592] 1 80 80
[593] 1 81 81
[594] 1 82 82
[595] 1 83 83
[596] 1 84 84
[597] 1 85 85
[598] 1 86 86
[599] 1 87 87

```

The above results show that the HMC is meeting the 29.44Gbps throughput, as the HMC write error register (`hmc_wr_err`) and HMC read error register (`hmc_rd_err`) is 0, which means the HMC is always ready for data when the HMC write/read request occurs. Note that the HMC read count (`hmc_rd_cnt`), HMC write count (`hmc_wr_cnt`) and HMC read out count (`hmc_out_cnt`) are all equal, which is expected. Compare the HMC snapshot output data and the HMC reorder snapshot captured output data - notice how the HMC snapshot output data is out of sequence in places and the HMC snapshot reorder data is in sequence again. There is no missing data. This is how the HMC should work.

Edit the `tut_hmc.py` script again and change the data rate to 58.88Gbps. Rerun as above and this time notice that the difference in the above registers and snapshot data. What do you see? You should see that HMC read count, HMC write count and HMC read out count values do not match. The HMC write error register and HMC read error register should be non zero, which indicates that the HMC is asserting write and read requests when the HMC write and read ready signals are not asserted, which means the FIFO is not being cleared fast enough. The HMC read output data will still be out of sequence, but data will be lost. This can be clearly seen in the HMC reorder snapshot captured output.

Other notes

- iPython includes tab-completion. In case you forget which function to use, try typing `library_name.tab`
- There is also onboard help. Try typing `library_name.function?`
- Many libraries have onboard documentation stored in the string `library_name.doc`

- KATCP in Python supports asynchronous communications. This means you can send a command, register a callback for the response and continue to issue new commands even before you receive the response. You can achieve much greater speeds this way. The Python wrapper in the corr package does not currently make use of this and all calls are blocking. Feel free to use the lower layers to implement this yourself if you need it!

Conclusion

This concludes the HMC Interface Tutorial for SKARAB. You have learned how to utilize the HMC interface on a SKARAB to write and read data to/from the HMC Mezzanine Card. You also learned how to further use Python to program the FPGA and control it remotely using casperfpga.

1.1.9 Tutorial 3: Wideband Spectrometer

Introduction

A spectrometer is something that takes a signal in the time domain and converts it to the frequency domain. In digital systems, this is generally achieved by utilising the FFT (Fast Fourier Transform) algorithm.

When designing a spectrometer for astronomical applications, it's important to consider the science case behind it. For example, pulsar timing searches will need a spectrometer which can dump spectra on short timescales, so the rate of change of the spectra can be observed. In contrast, a deep field HI survey will accumulate multiple spectra to increase the signal to noise ratio. It's also important to note that "bigger isn't always better"; the higher your spectral and time resolution are, the more data your computer (and scientist on the other end) will have to deal with. For now, let's skip the science case and familiarize ourselves with an example spectrometer.

Setup

This tutorial comes with a completed model file, a compiled bitstream, ready for execution on Skarab, as well as a Python script to configure the Skarab and make plots. [Here](#)

Spectrometer Basics

When designing a spectrometer there are a few main parameters of note:

- **Bandwidth:** The width of your frequency spectrum, in Hz. This depends on the sampling rate; for complex sampled data this is equivalent to:

$$BW = \text{sampling rate} = \frac{1}{\text{sampling period}}$$

In contrast, for real or Nyquist sampled data the rate is half this:

$$BW = \frac{\text{sampling rate}}{2} = \frac{1}{2 \times \text{sampling period}}$$

as two samples are required to reconstruct a given waveform .

- **Frequency resolution:** The frequency resolution of a spectrometer, Δf , is given by

$$\Delta f = \frac{BW}{no. channels},$$

and is the width of each frequency bin. Correspondingly, Δf is a measure of how precise you can measure a frequency.

- **Time resolution:** Time resolution is simply the spectral dump rate of your instrument. We generally accumulate multiple spectra to average out noise; the more accumulations we do, the lower the time resolution. For looking at short timescale events, such as pulsar bursts, higher time resolution is necessary; conversely, if we want to look at a weak HI signal, a long accumulation time is required, so time resolution is less important.

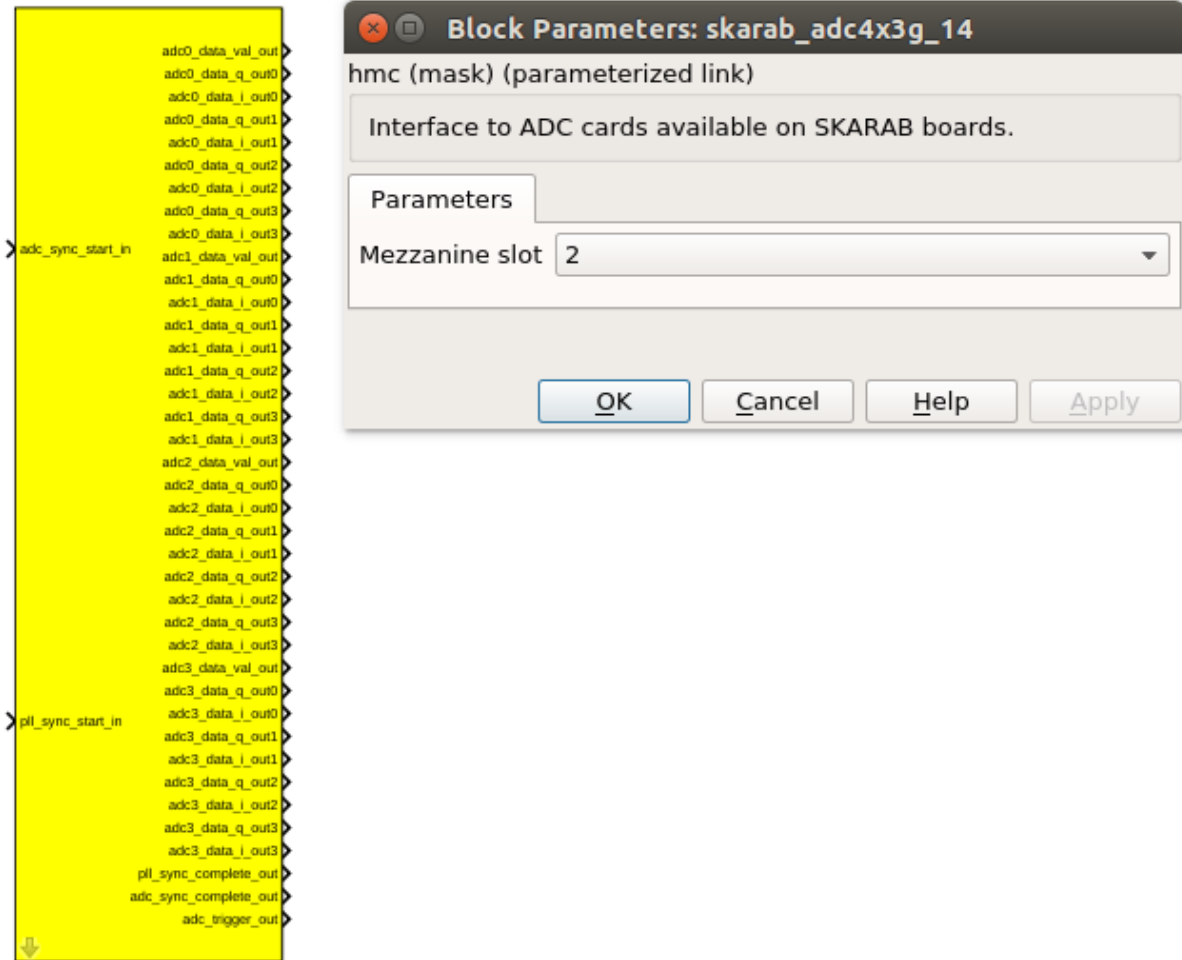
Simulink / CASPER Toolflow

Simulink Design Overview

If you're reading this, then you've already managed to find all the tutorial files. By now, I presume you can open the model file and have a vague idea of what's happening. The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some "blockumentation" in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- In the slx file, you'll notice a subsystem block in the top left corner of the design. If you click into it, you'll see it contains a 40 GbE core. That core instantiates the board support package and the microblaze controller. Therefore, the core is a must for any Skarab design and is akin to the Skarab platform block.
- The all important Xilinx token is placed to allow System Generator to be called to compile the design.
- In the MSSGE block, the hardware type is set to "SKARAB:xc7vz690t" and clock rate is specified as 187.5MHz. This frequency is specially chosen to avoid overflows on the ADC. Implementing other clock frequencies will require you to use the data valid port leaving the ADC yellow block.
- The input signal is digitised by the ADC, resulting in eight parallel time samples of 16 bits each clock cycle: four in i and four in q. The ADC runs at 3 GHz but decimates by a factor of four, which gives a 375 MHz nyquist sampled spectrum. The Skarab ADC uses a digital downconverter, which has a default frequency of 1 GHz. The output range is a signed number in the range -1 to +1 (ie 15 bits after the decimal point). This is expressed as fix_16_15.
- Unlike the other CASPER spectrometer tutorials, we use the complex FFT block here. The Skarab ADC produces demultiplexed i and q channels that are concatenated and fed to the FFT block.
- You may notice Xilinx delay blocks dotted all over the design. It's common practice to add these into the design as it makes it easier to fit the design into the logic of the FPGA. It consumes more resources, but eases signal timing-induced placement restrictions.
- The real and imaginary (sine and cosine value) components of the FFT are plugged into power blocks, to convert from complex values to real power values by squaring.
- The requantized signals then enter the vector accumulators, simple_bram_vacc0 through simple_bram_vacc3, which are 64 bit vector accumulators. Accumulation length is controlled by the acc_cntrl block.
- The accumulated signal is then fed into software registers, mem1 through mem4.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

adc

The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter.

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 16 bit binary point numbers in the range of -1 to 1 and are then output by the ADC. This is achieved through the use of two's-complement representation with the binary point placed after the seven least significant bits. This means we can represent numbers from -32768 through to 32767 including the number 0. Simulink represents such numbers with a `fix_16_15` moniker.

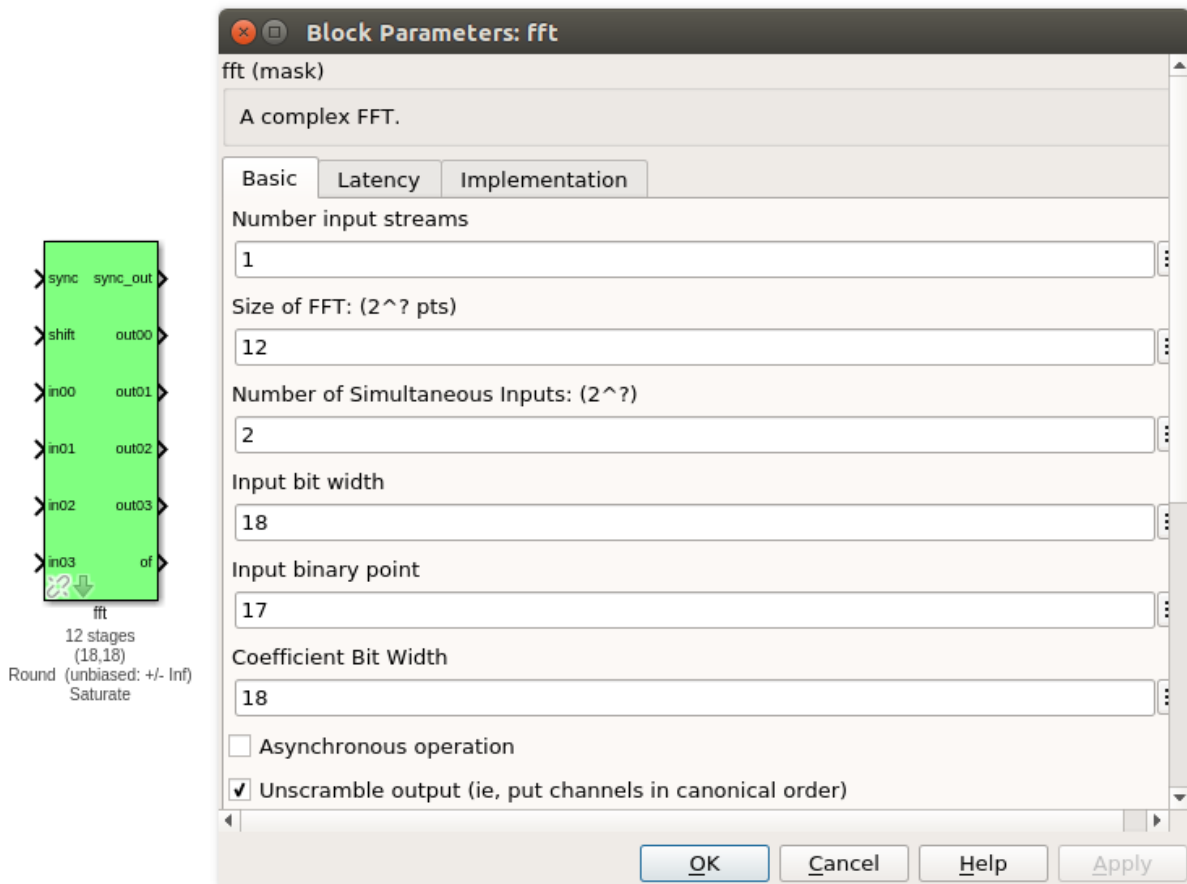
ADCs often internally bias themselves to halfway between 0 and -1. This means that you'd typically see the output of an ADC toggling between zero and -1 when there's no input. It also means that unless otherwise calibrated, an ADC will have a negative DC offset.

The Skarab ADC is clocked at 3.0 GHz. There is a decimation factor of 4, so the sample rate is 750 MHz. The i and q channels each have a demux factor of 4, so the FPGA is clocked at 187.5 MHz. The bandwidth for a 750 MHz sample rate is 375 MHz, as Nyquist sampling requires two samples (or more) each second.

INPUTS**OUTPUTS**

The Skarab ADC has four channels and a series of eight outputs for each. The outputs for a channel comprise four demultiplexed i's and four demultiplexed q's. The i0 port is concatenated with the q0 port to form a complex stream using the real/imaginary-to-complex block. Similarly, the i1 port is concatenated with the q1 port, i2 with q2, and i3 with q3.

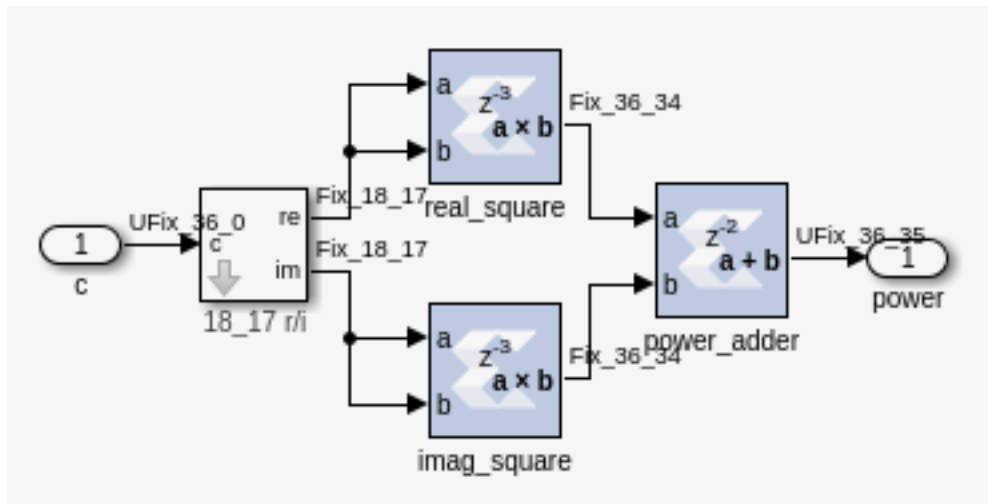
fft



The FFT block is the most important part of the design to understand. The cool green of the FFT block hides the complex and confusing FFT butterfly biphase algorithms that are under the hood. You do need to have a working knowledge of it though, so I recommend reading Chapter 8 and Chapter 12 of Smith's free online DSP guide at (<http://www.dspguide.com/>). Parts of the documentation below are taken from the [[Block_Documentation | block documentation]] by Aaron Parsons and Andrew Martens.

INPUTS/OUTPUTS

PARAMETERS

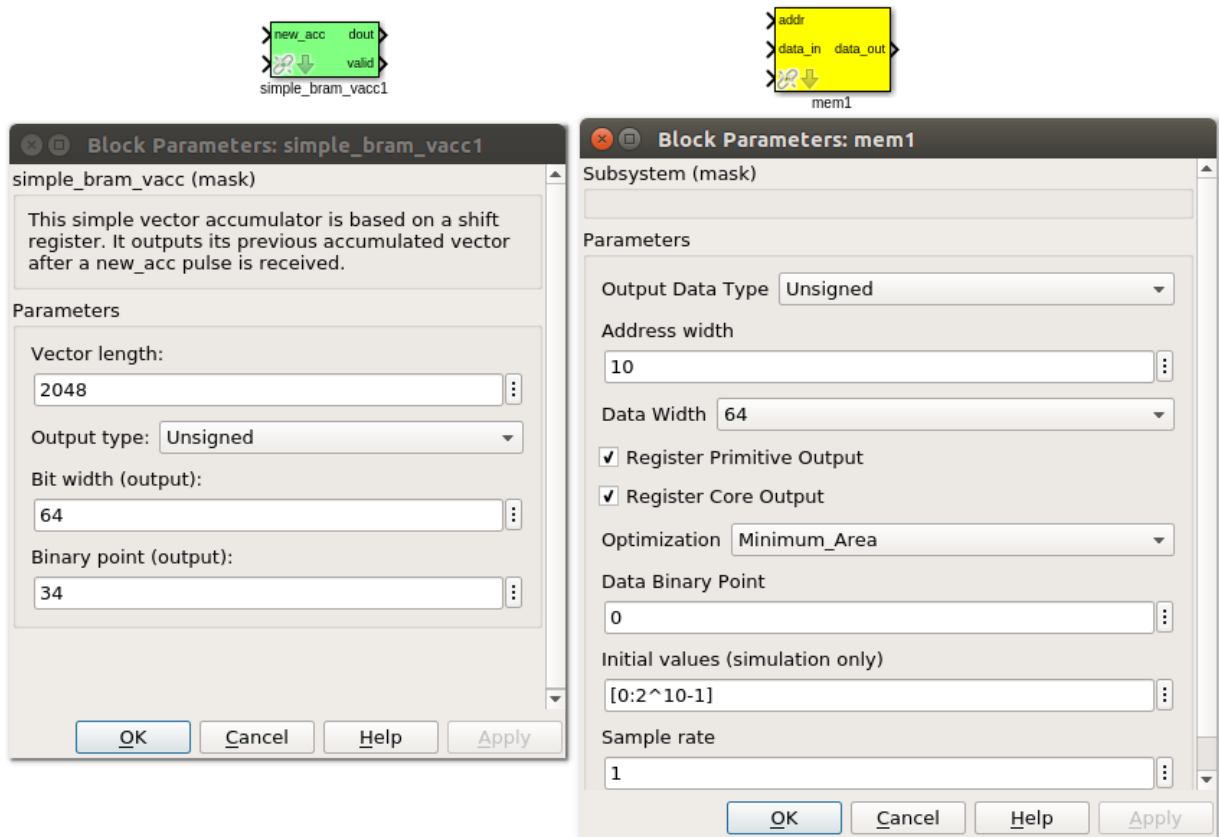
power

The power block computes the power of a complex number. The power block typically has a latency of 5 and will compute the power of its input by taking the sum of the squares of its real and imaginary components. The power block is written by Aaron Parsons and online documentation is by Ben Blackman.

In our design, there are two power blocks, which compute the power of the odd and even outputs of the FFT. The output of the block is 36.34 bits; the next stage of the design re-quantizes this down to a lower bitrate.

INPUTS/OUTPUTS**PARAMETERS**

simple_bram_vacc



The `simple_bram_vacc` block is used in this design for vector accumulation. Vector growth is approximately 28 bits each second, so if you wanted a really long accumulation (say a few hours), you'd have to use a block such as the `qdr_vacc` or `dram_vacc`. As the name suggests, the `simple_bram_vacc` is simpler so it is fine for this demo spectrometer. The FFT block demultiplexed frequency bins directly to the accumulator and memory blocks. These streams are multiplexed in software using the `tut_spec.py` script.

PARAMETERS

INPUTS/OUTPUTS

Software Registers

There are a few `control registers`, led blinkers, and `snap` block dotted around the design too:

- **cnt_rst**: Counter reset control. Pulse this high to reset all counters back to zero.
- **acc_len**: Sets the accumulation length. Have a look in `tut3.py` for usage.
- **sync_cnt**: Sync pulse counter. Counts the number of sync pulses issued. Can be used to figure out board uptime and confirm that your design is being clocked correctly.
- **acc_cnt**: Accumulation counter. Keeps track of how many accumulations have been done.
- **led0_sync**: Back on topic: the `led0_sync` light flashes each time a sync pulse is generated. It lets you know your ROACH is alive.
- **led1_new_acc**: This lights up `led1` each time a new accumulation is triggered.

- **led2_acc_clip**: This lights up led2 whenever clipping is detected.

There are also some [snap](#) blocks, which capture data from the FPGA fabric and makes it accessible to the Power PC. This tutorial doesn't go into these blocks (in its current revision, at least), but if you have the inclination, have a look at their [documentation](#).

If you've made it to here, congratulations, go and get yourself a cup of tea and a biscuit, then come back for part two, which explains the second part of the tutorial – actually getting the spectrometer running, and having a look at some spectra.

Configuration and Control

Hardware Configuration

The tutorial comes with a pre-compiled bof file, which is generated from the model you just went through (tut_spec.fpg). All communication and configuration will be done by the python control script called tut_spec.py.

Next, you need to set up your Skarab. Switch it on, making sure that:

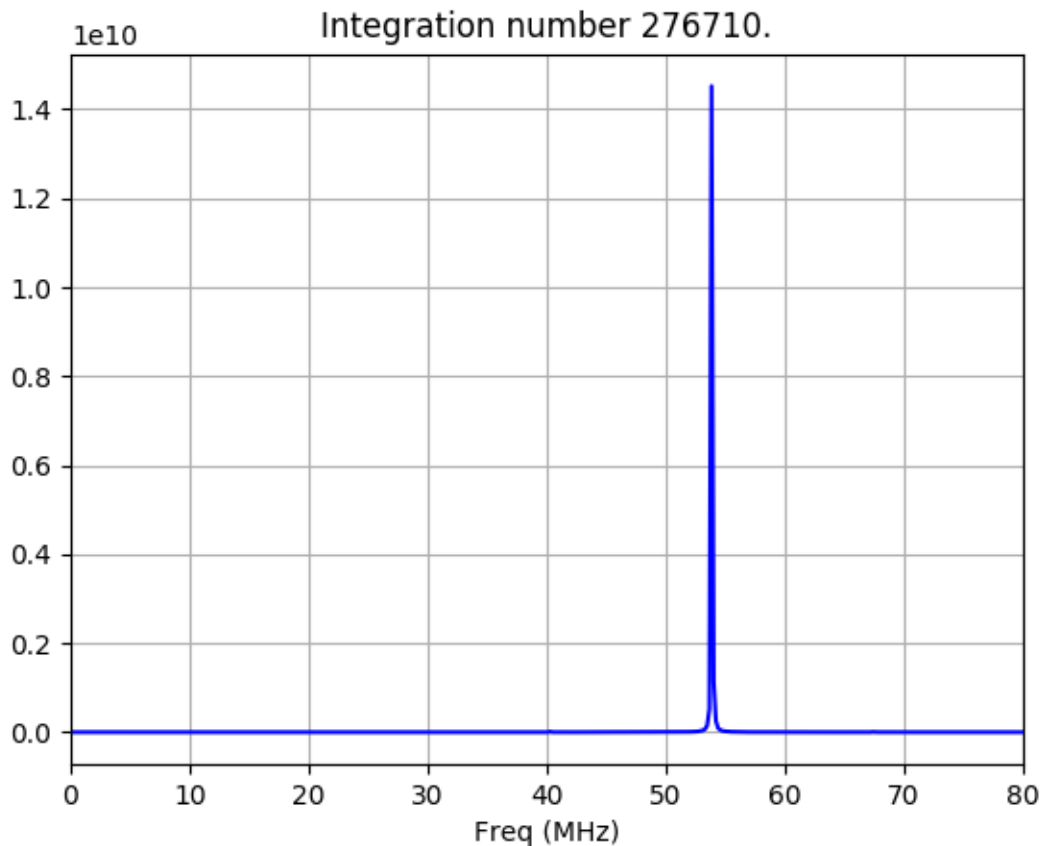
- Your tone source is set within the band of the ADC. The default digital downconverter setting is 1 GHz, so signals within 375 MHz of this frequency should pass. In the tut_spec.py script, 1 GHz is mapped to DC. In our setup, we set the tone frequency to 1.054 GHz. If you use a different tone frequency, be sure to update the `freq_range_mhz = [0, 80]` in the Python script so that the plot covers the range of your tone.

The tut_spec.py spectrometer script

Once you've got that done, it's time to run the script. First, check that you've connected the ADC to ZDOK0, and that the clock source is connected to clk_i of the ADC. Now, if you're in linux, browse to where the tut3.py file is in a terminal and at the prompt type

```
./tut_spec.py <skarab IP or hostname> -l <accumulation length> -b <fpgfile name>
```

replacing with the IP address of your Skarab, is the number of accumulations, and with your fpgfile. You should see a spectrum like this:



Take some time to inspect the `tut_spec.py` script. It is quite long, but don't be intimidated. Most of the script is configuration for the ADC. The import lines begin after the `#START OF MAIN` comment. There, you will see that the script

- Instantiates the `casperfpga` connection with the Skarab
- Uploads the fpg file
- Sets the ADC
- Records ADC snapshots, interleaves them and writes to a file `adcN_data.txt` where N is 0..4
- Plots the spectral outputs of the memory blocks

Conclusion

If you have followed this tutorial faithfully, you should now know:

- What a spectrometer is and what the important parameters for astronomy are.
- Which CASPER blocks you might want to use to make a spectrometer, and how to connect them up in Simulink.
- How to connect to and control a Skarab spectrometer using python scripting.

Red Pitaya

1. Introduction Tutorial [Step-by-Step](#) or [Completed](#)
2. ADC and DAC Interface Tutorial [Step-by-Step](#) or [Completed](#)

3. Spectrometer Tutorial *Step-by-Step* or *Completed*

1.1.10 Tutorial 1: Introduction to Simulink

In this tutorial, you will create a simple Simulink design using both standard Xilinx system generator blockset, as well as library blocks specific to CASPER boards (so-called “Yellow Blocks”). At the end of this tutorial you will know:

- How to generate an fpg file,
- Program it to a CASPER FPGA board (specifically the [Red Pitaya](#)), and
- Interact with your running hardware design using [casperfpga](#) via an interactive Python Interface.

Creating Your Design

Create a New Model

Start MATLAB via executing the startsg command, as described [here](#). This ensures that necessary Xilinx and CASPER libraries are loaded into development environment by Simulink. When MATLAB starts up, open Simulink by typing simulink on the MATLAB command line. Start a new model, and save it with an appropriate name. **With Simulink, it is very wise to save early and often.**

There are some Matlab limitations you should be aware-of right from the start:

- **Do not use spaces in your filenames** or anywhere in the file path as it will break the toolflow.
- **Do not use capital letters in your filenames** or anywhere in the file path as it will break the toolflow.
- **Beware block paths that exceed 64 characters.** This refers to not only the file path, but also the path to any block within your design.
 - For example, if you save a model file with a name ~/some_really_long_filename.slx, and have a block called in a submodule the longest block path would be: some_really_long_filename_submodule_block.
 - If you use lots of subsystems, this can cause problems.

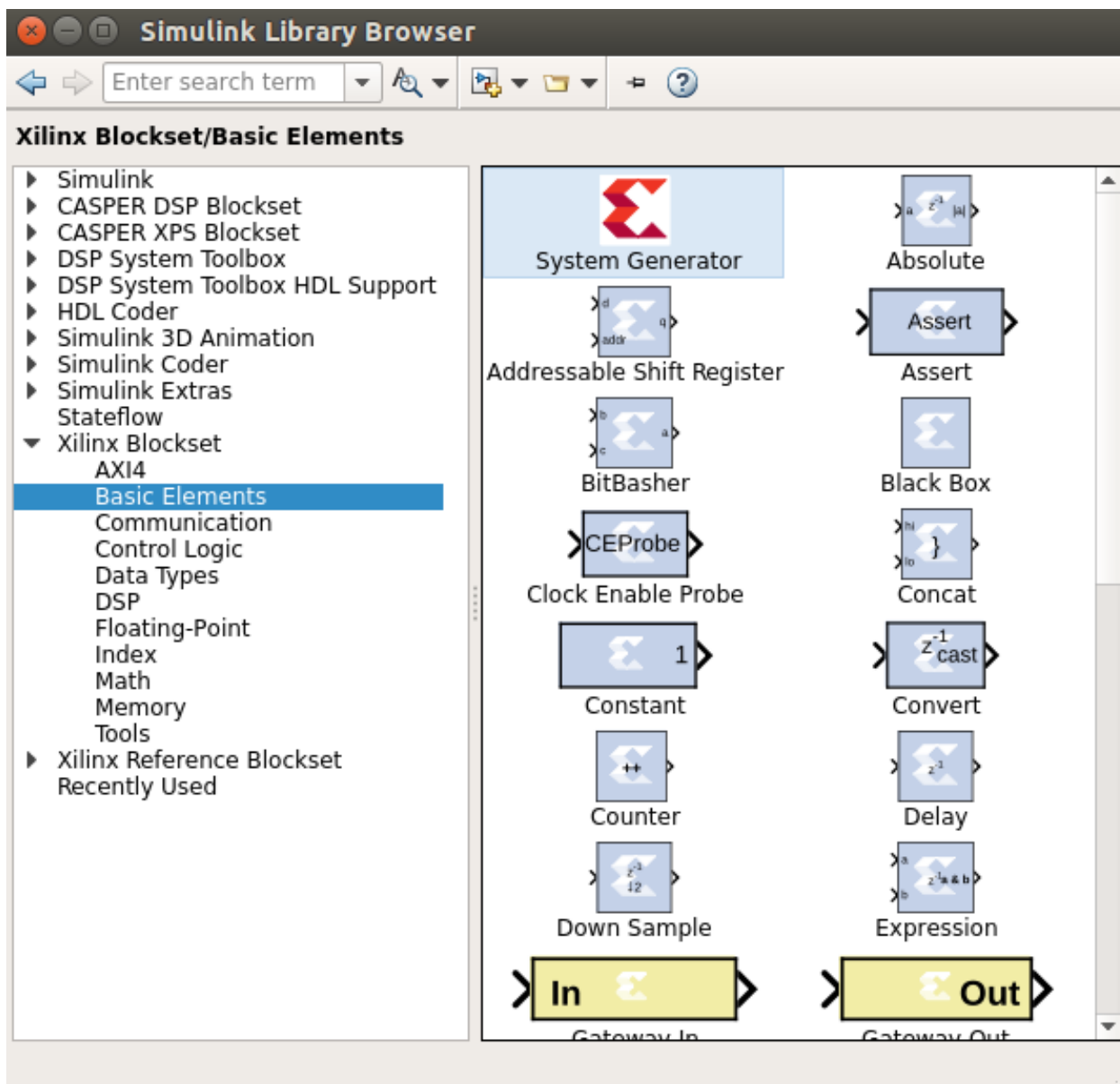
Library organization

There are three libraries which you will use when you design firmware in Simulink. More information on the toolflow itself can be found [here](#).

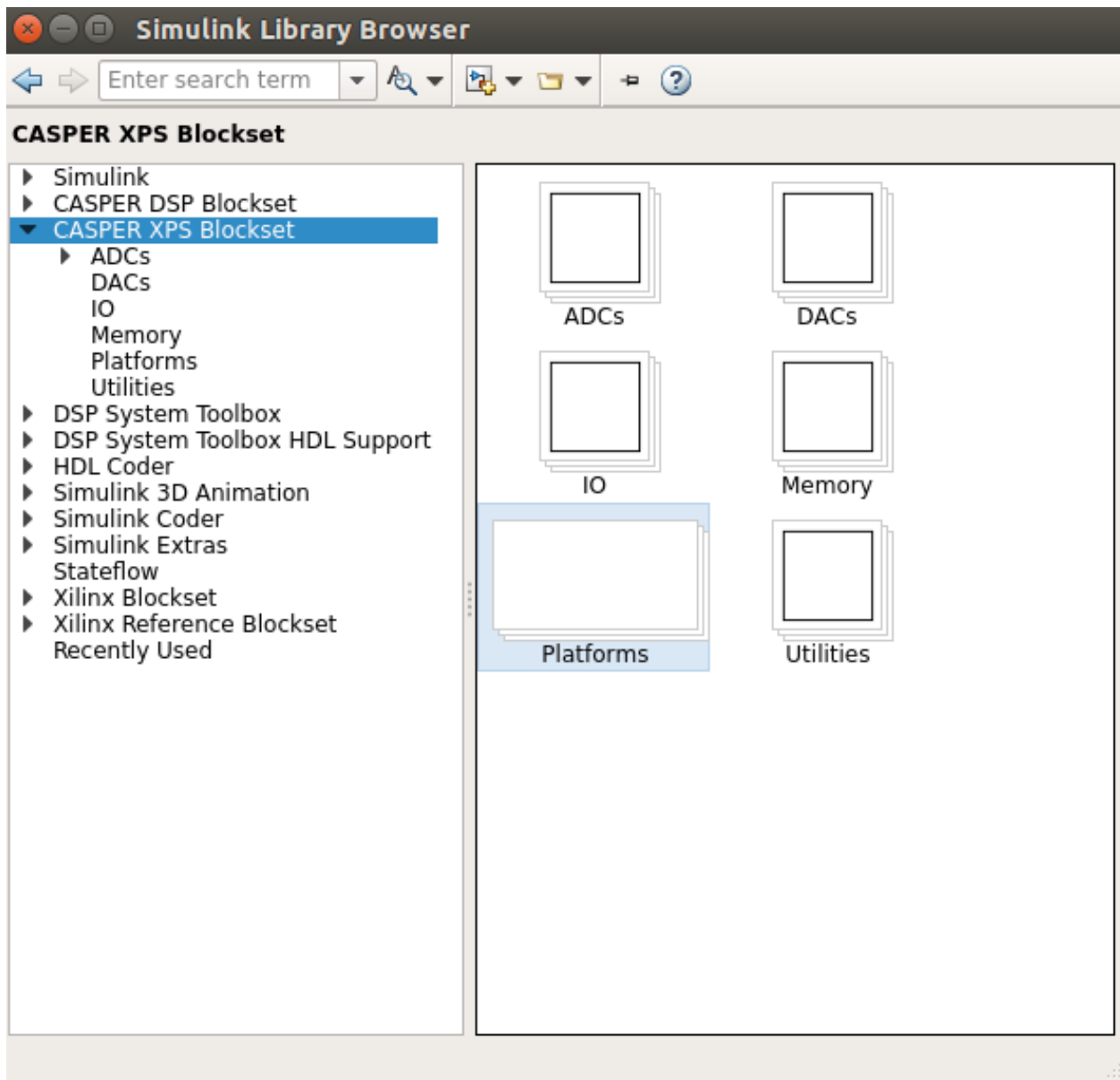
1. The **CASPER XPS Library** contains “Yellow Blocks” – these are blocks which encapsulate interfaces to hardware (ADCs, Memory chips, CPUs, Ethernet ports, etc.)
2. The **CASPER DSP Library** contains (mostly green) blocks which implement DSP functions such as filters, FFTs, etc.
3. The **Xilinx Library** contains blue blocks which provide low-level functionality such as multiplexing, delay-ing, adding, etc. The Xilinx library also contains the super-special System Generator block, which contains information about the type of FPGA you are targeting.

Add Xilinx System Generator and XSG core config blocks

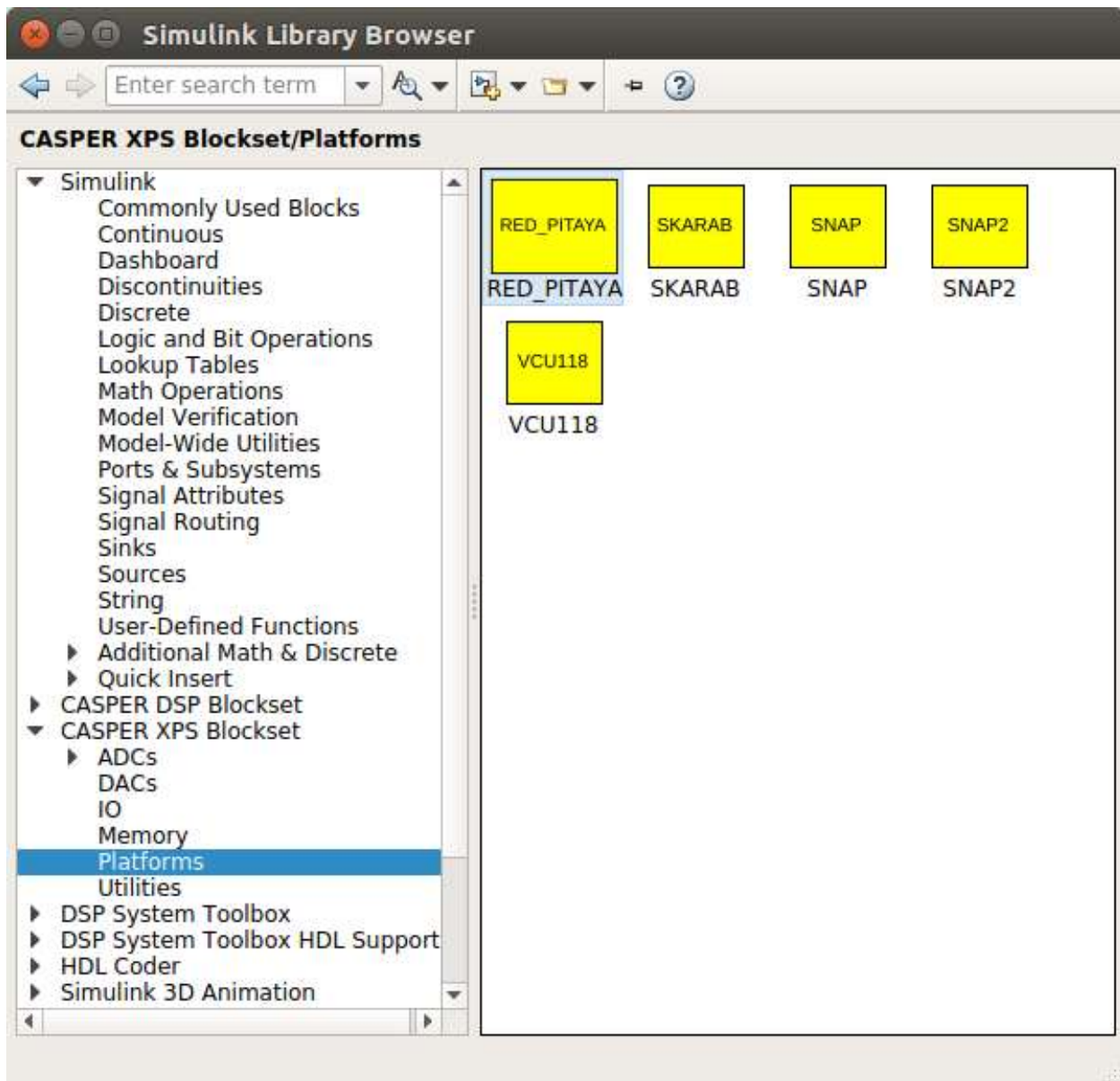
Add a System generator block from the Xilinx library by locating the Xilinx Blockset library’s Basic Elements sub-section and dragging a System Generator token onto your new file.



Do not configure it directly, but rather add a platform block representing the system you are compiling for. These can be found in the CASPER XPS System Blockset library. For Red Pitaya (and later) platforms, you need a block which matches the platform name, which can be found in the library under “platforms”, as shown below.



casper_xps_select



casper_xps_select

Double click on the platform block that you just added. The Hardware Platform parameter should match the platform you are compiling for. Once you have selected a board you need to choose its clock source. The Red Pitaya Platform Yellow Block has default parameters which do not need to be changed for this tutorial. However, a good rule of thumb for designs including ADCs, you probably want the FPGA clock to be derived from the sampling clock.

The configuration Yellow Block knows what FPGA corresponds to which platform and will automatically configure the System Generator block which you previously added.

The System Generator and XPS Config blocks are required by all CASPER designs

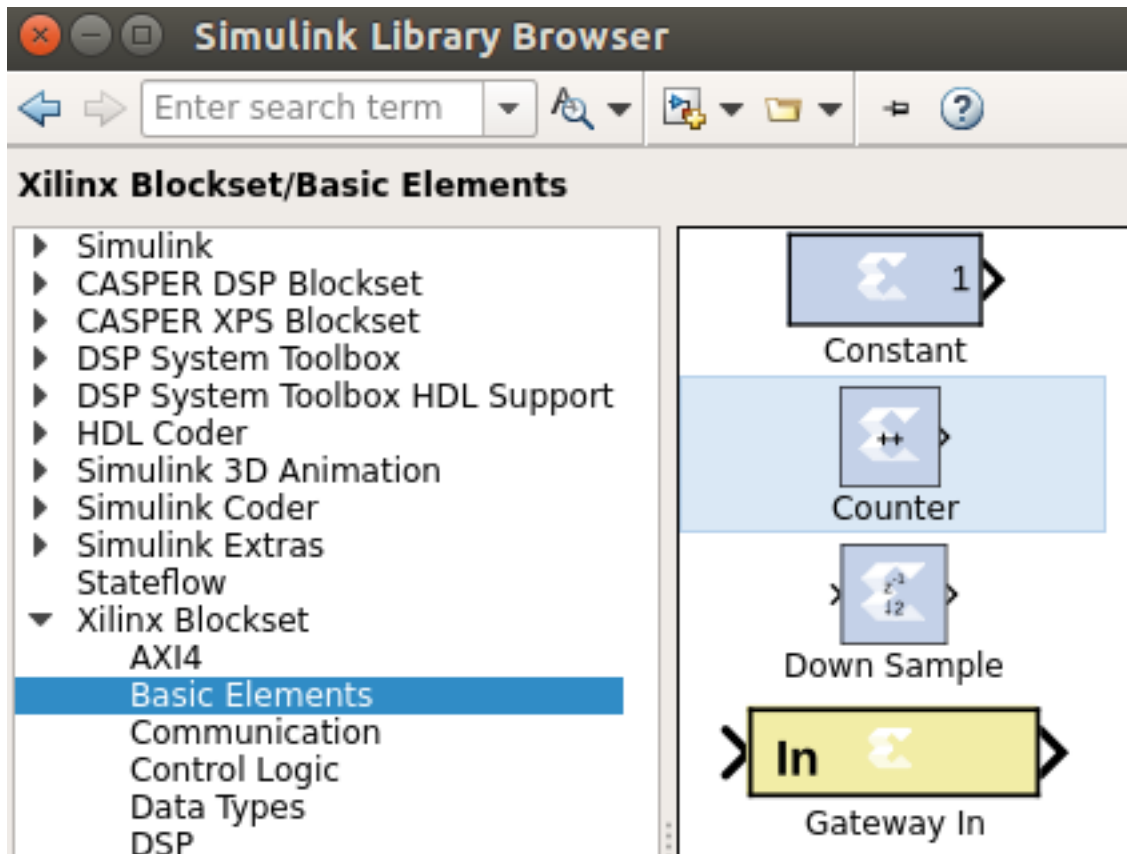
Flashing LED

To demonstrate the basic use of hardware interfaces we will make an LED flash. With the FPGA running at ~100MHz (or greater), the most significant bit (MSB) of a 27 bit counter will toggle approximately every 0.67 seconds. We can output this bit to an LED on your board. Most (all?) CASPER platforms have at least four LEDs, with the exact

configuration depending on the board. We will make a small circuit connecting the top bit of a 27 bit counter to one of these LEDs. When compiled this will make the LED flash with a 50% duty cycle approximately once a second.

Add a counter

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



xilinx_select_counter.png

Double-click it and set it for free running, 27 bits, unsigned. This means it will count from 0 to $2^{27} - 1$, and will then wrap back to zero and continue.

counter_led (Xilinx Counter)

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☐ Provide synchronous reset port

☐ Provide enable port

Explicit Sample Period

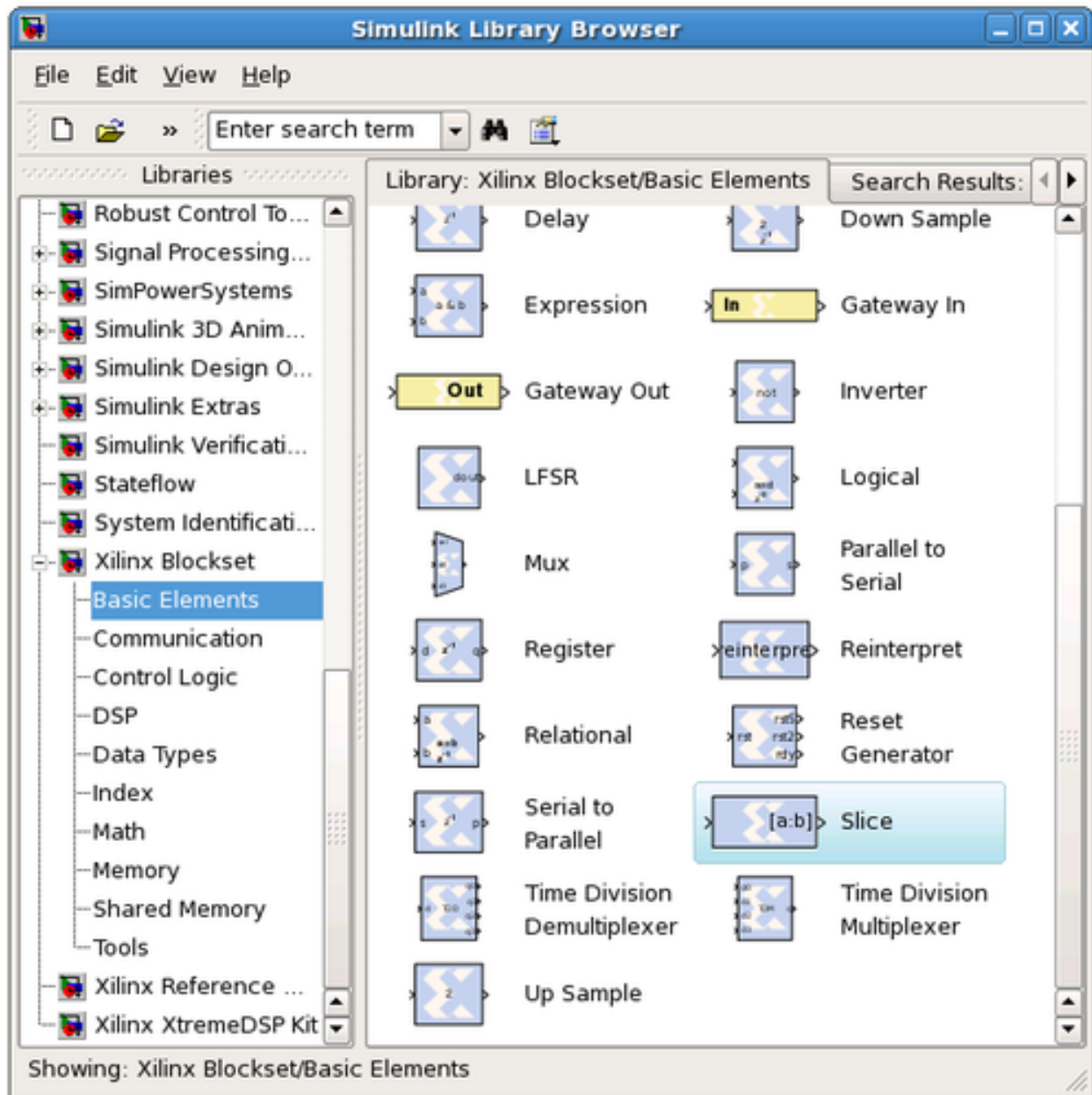
Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

OK **Cancel** **Help** **Apply**

Add a slice block to select out the msb

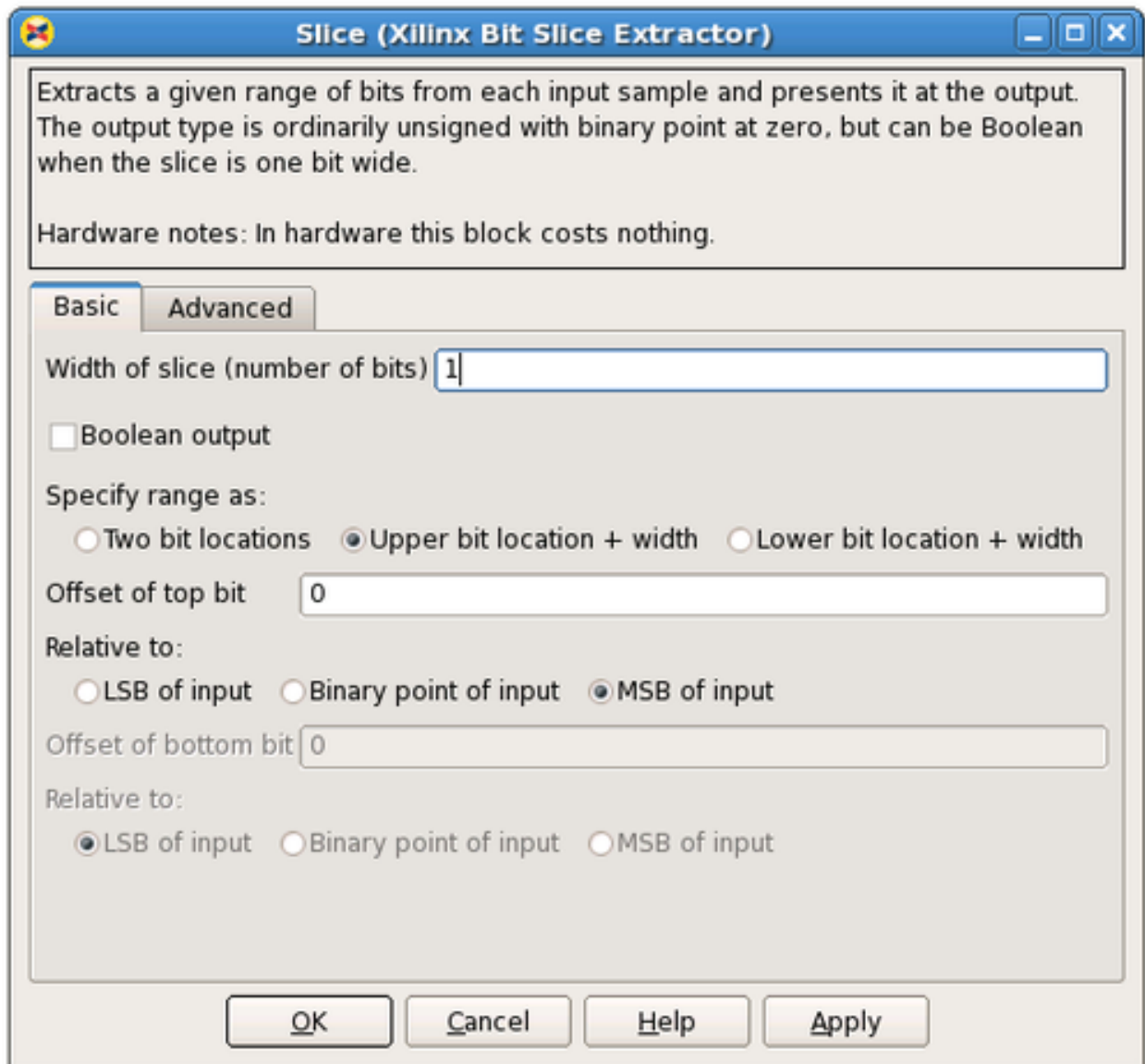
We now need to select the **most significant bit** (msb) of the counter. We do this using a slice block, which Xilinx provides. Xilinx Blockset -> Basic Elements -> Slice.



Slice_select.png

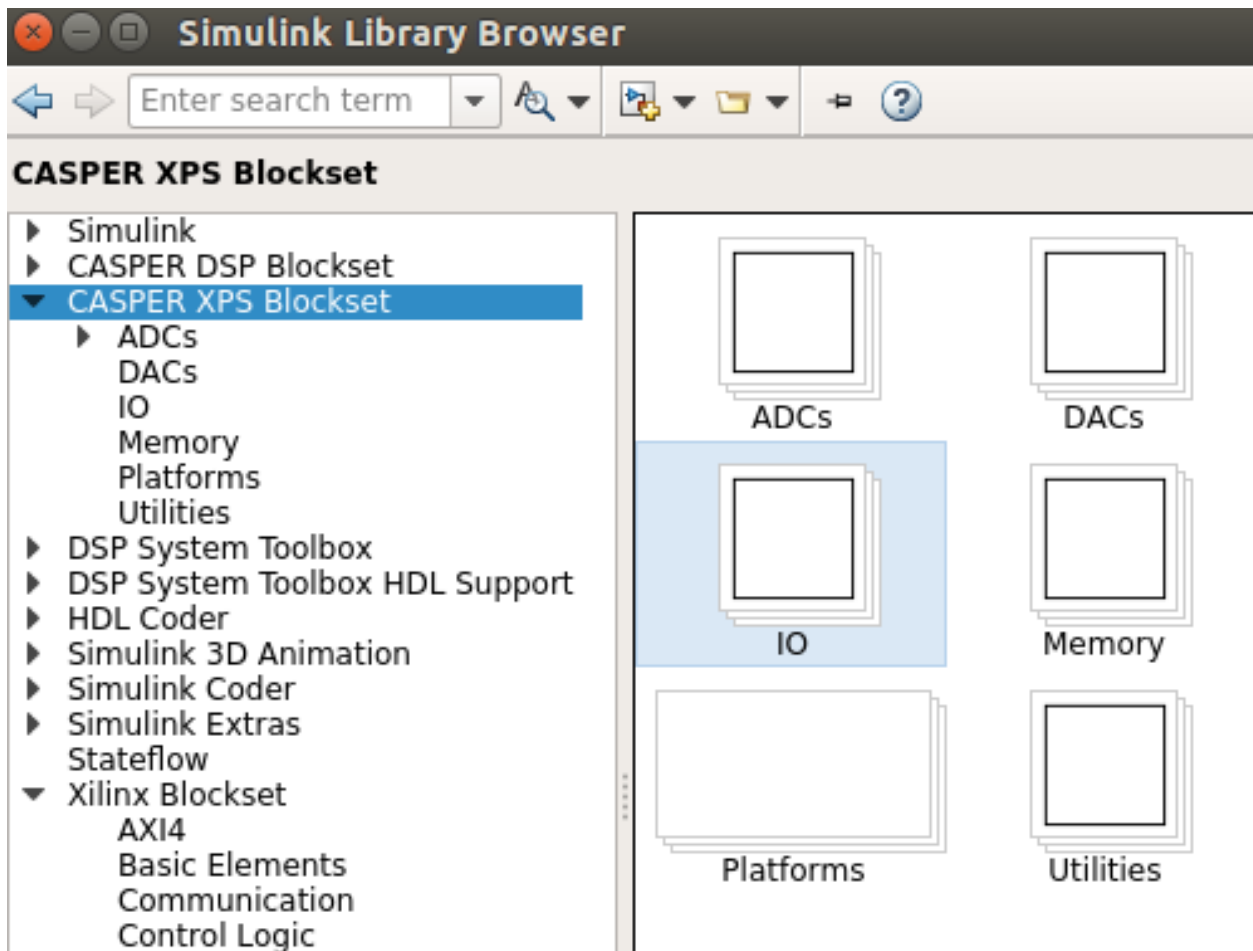
Double-click on the newly-added slice block. There are multiple ways to select which bit(s) you want. In this case, it is simplest to index from the upper end and select the first bit. If you wanted the **least significant bit** (LSB), you can also index from that position. You can either select the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

Set it for 1 bit wide with offset from top bit at zero. As you might guess, this will take the 27-bit input signal, and output just the top bit.

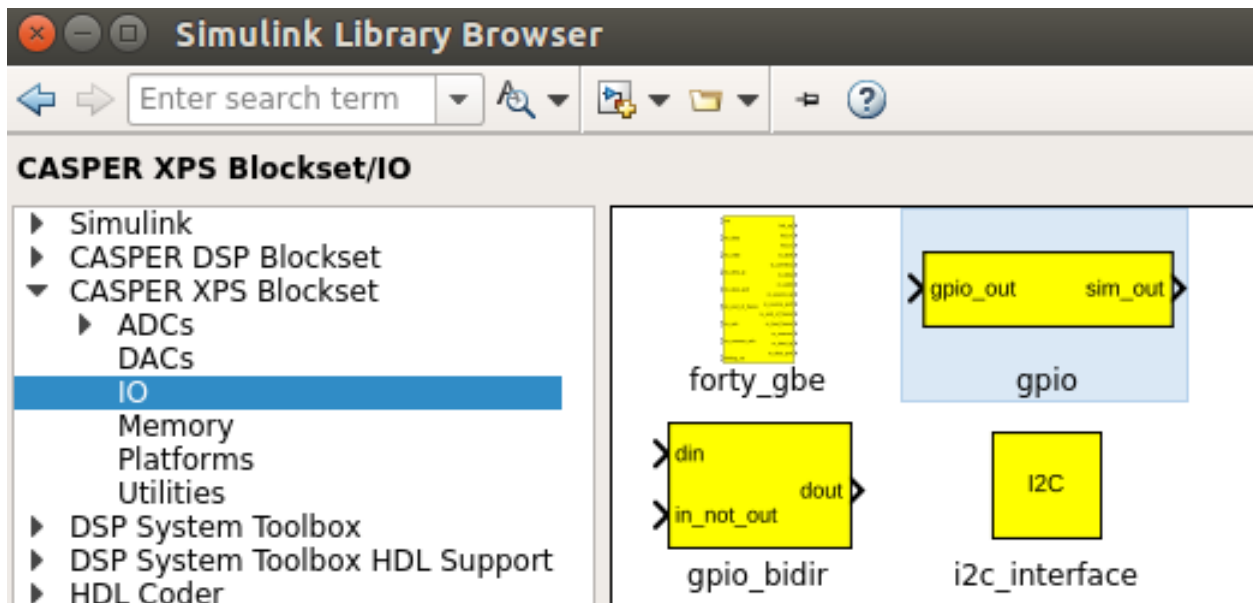


Add a GPIO block

From: CASPER XPS library -> gpio.





casper_xps_select



casper_xps_select

In order to send the 1-bit signal you have sliced off to an LED, you need to connect it to the right FPGA output pin. To do this you can use a GPIO (general-purpose input/output) block from the XPS library, this allows you to route a signal from Simulink to a selection of FPGA pins, which are addressed with user-friendly names. Set it to use Red Pitaya's LED bank as output. Once you've chosen the LED bank, you need to pick *which* LED you want to output to.

Set the GPIO bit index to 0 (the first LED) and the data type to Boolean with bitwidth 1. This means your simulink input is a 1 bit Boolean, and the output is LED0.

 **Block Parameters: gpio**

gpio (mask) (link)

GPIO interfaces for CASPER hardware.

For usage of LEDs on SKARAB boards, please note that the board will boot into 'BSP mode' by default. To change the control of the Front Panel LEDs to your design, execute the command 'control_front_panel_leds_write()' on your SKARAB CASPERFPGA object via the transport layer.

For more information please click help below.

Parameters

I/O group led

Custom I/O group

I/O direction out

Data Type Boolean

Data bitwidth

Data binary point

GPIO bit index

Sample period

☐ Use DDR

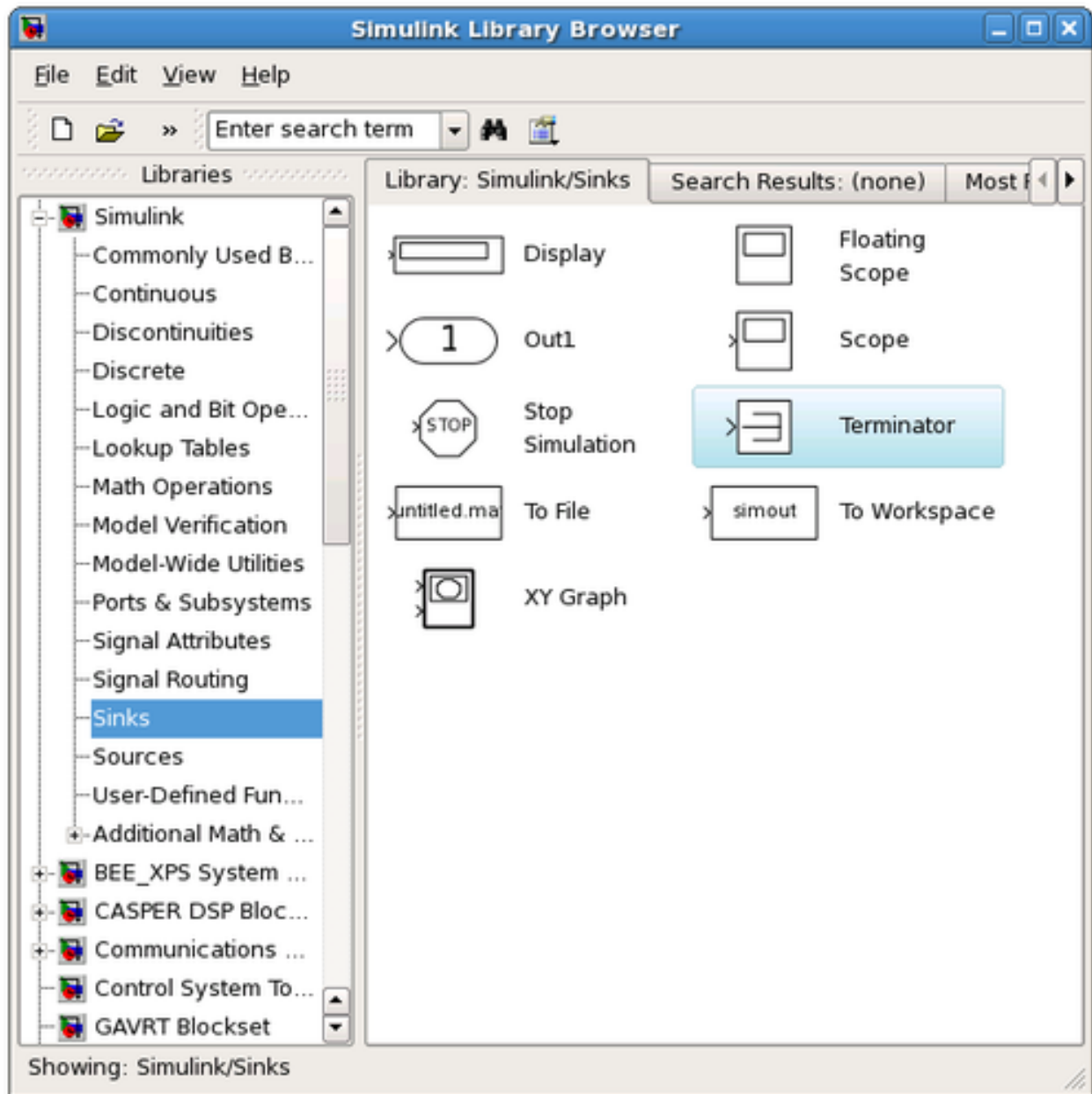
☒ Pack register in the pad

Register clock phase 0

Termination method None

Add a terminator

To prevent warnings (from MATLAB & Simulink) about unconnected outputs, terminate all unused outputs using a *Terminator*:



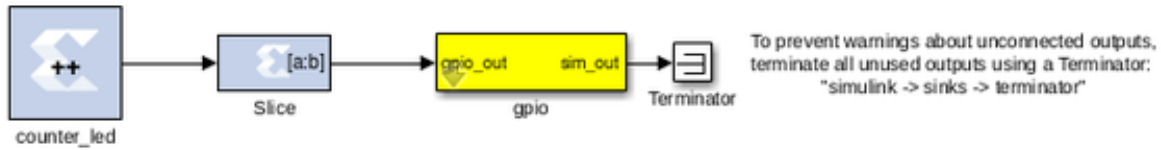
From: Simulink -> Sinks -> Terminator

You can also use the Matlab function `XIAddTerms`, run in the MATLAB prompt, to automatically terminate your unused outputs.

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.

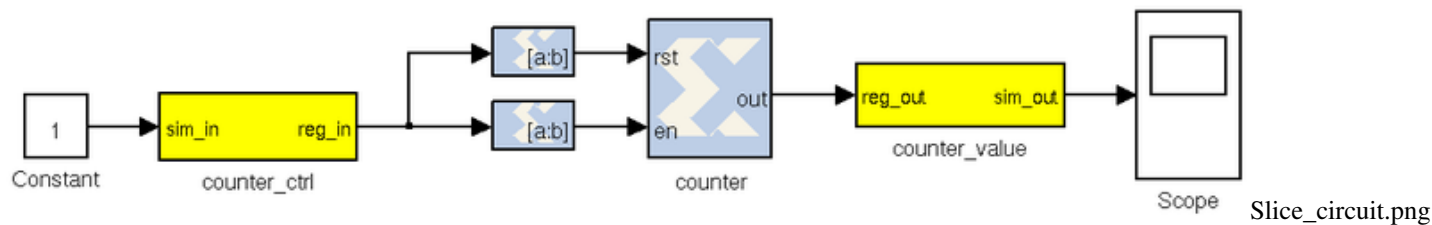
To connect the blocks simply click and drag from the ‘output arrow’ on one block and drag it to the ‘input arrow’ of another block. Connect the blocks together: Counter -> Slice -> gpio as showing in diagram below.



Remember to save your design often.

Software control

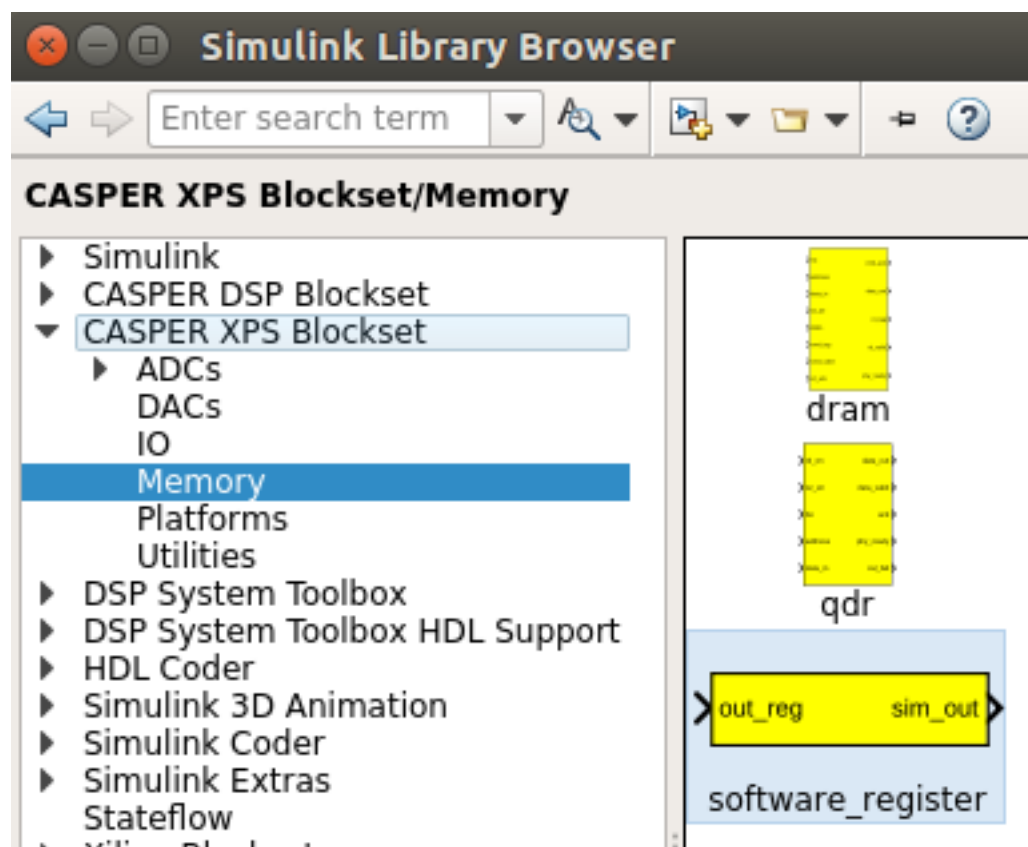
To demonstrate the use of software registers to control the FPGA from a computer, we will add registers so that the counter in our design can be started, stopped, and reset from software. We will also add a register so that we can monitor the counter's current value too. By the end of this section you will create a system that looks like this:



Add the software registers



We need two software registers:

1. To control the counter, and
2. To read its current value.



casper_xps_select_memory_swreg.png

Set the I/O direction to *From Processor* on the first one (counter control) to enable a value to be set from software and sent to your FPGA design. Set it to *To Processor* on the second one (counter value) to enable a value to be sent from the FPGA to software. Set both registers to a bitwidth of 32 bits.

 **Block Parameters: counter_ctrl**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction From Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts

0



Bitfield types, ufix=0, fix=1, bool=2

0

☒ Provide sim input/output?

1.1. Vivado

☐ Print format string?

 **Block Parameters: counter_value**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction To Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts

0

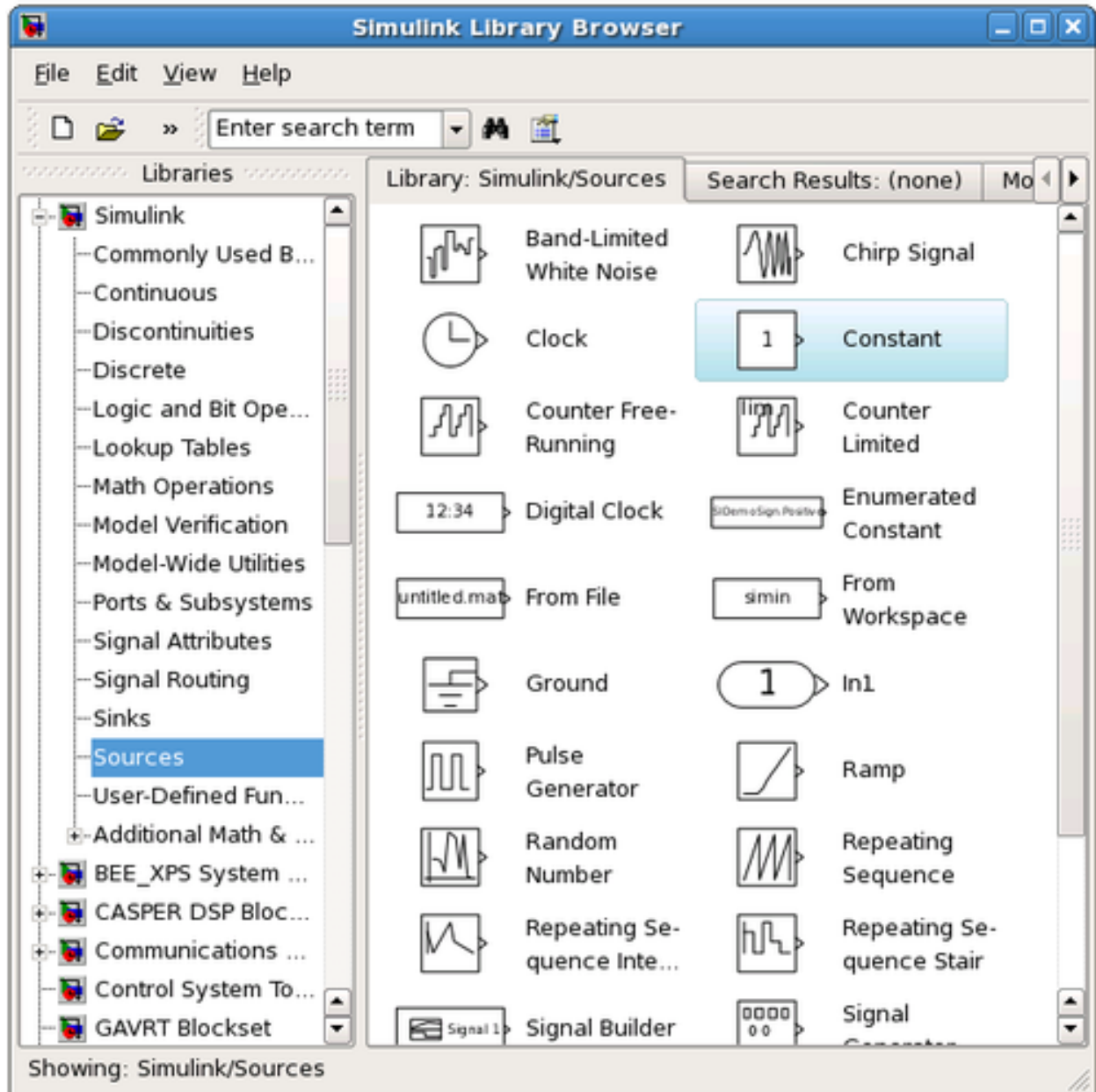
Bitfield types, ufix=0, fix=1, bool=2

0

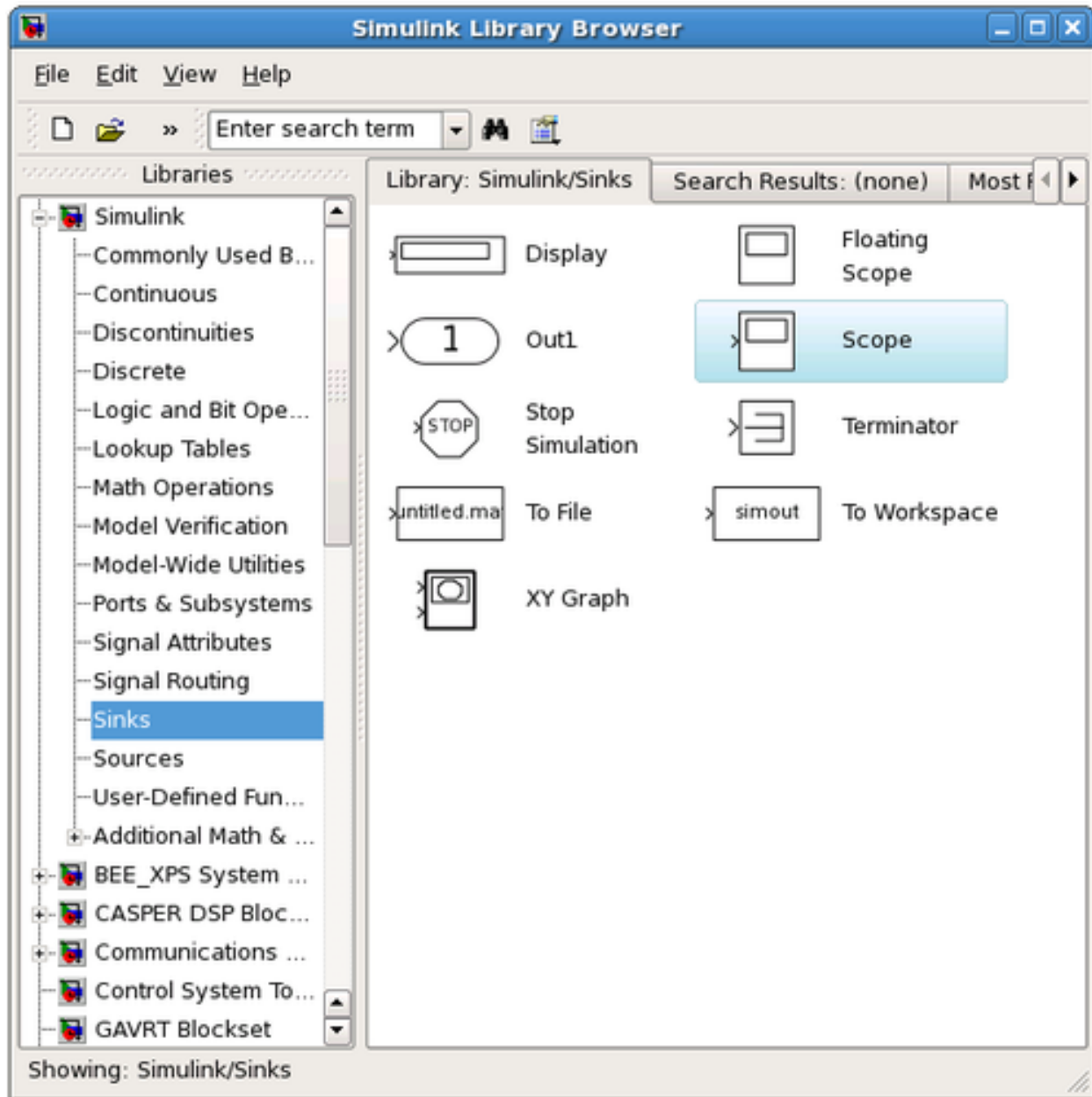
Rename the registers to something sensible, the names you give them here are the names you will use to access them from software. Do not use spaces, slashes and other special characters in these. Perhaps *counter_ctrl* and *counter_value* to represent the control and output registers respectively.

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

For now, set the *sim_reg* port to constant one using a Simulink-type constant. This can be found in *Simulink -> Sources*, and will enable the counter during simulations.



During simulation we can monitor the counter's value using a scope (*Simulink -> Sinks*):




Here is a good point to note that all blocks from the *Simulink* library are usually white in colour, and will not be compiled into hardware. i.e. They are present for simulation only. Xilinx blocks are usually blue in colour with the Xilinx logo, and will be compiled to hardware.

You need to use *gateway* blocks whenever connecting a Simulink-provided block (like a scope or sine-wave generator) to and from a Xilinx block. This will sample and quantize the Simulink signals so that they are compatible with the Xilinx world. Some blocks (like the software register) provide a gateway internally, so you can feed the input of a software register with a Xilinx signal, and monitor its output with a Simulink scope. However, in general, you must manually insert these gateways where appropriate. Simulink will issue warnings for any direct connections between the Simulink and Xilinx domains.

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:

 **counter (Xilinx Counter)** [-] [] [X]

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Advanced** **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☒ Provide synchronous reset port

☒ Provide enable port

Explicit Sample Period

Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

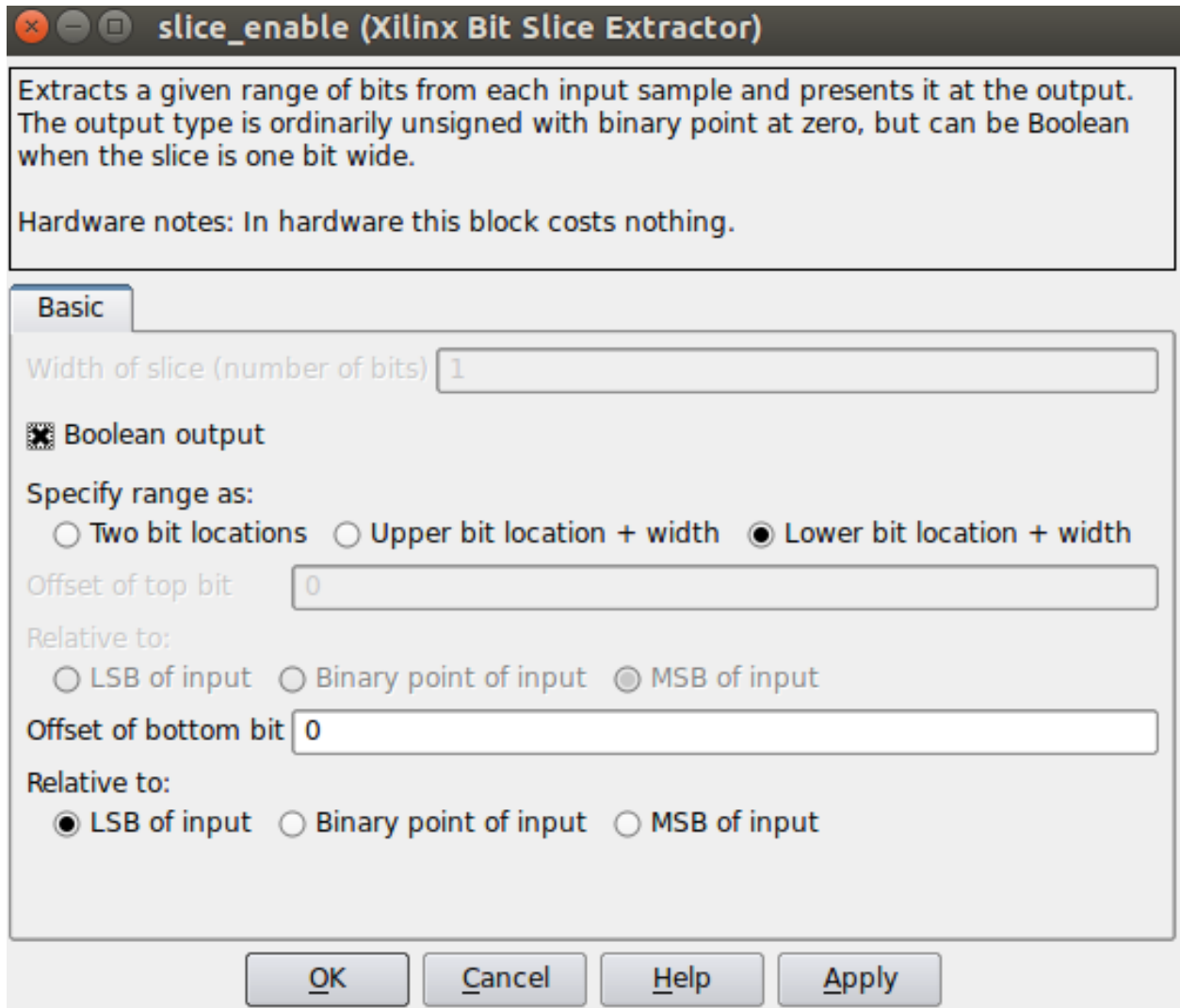
Add the slice blocks

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.

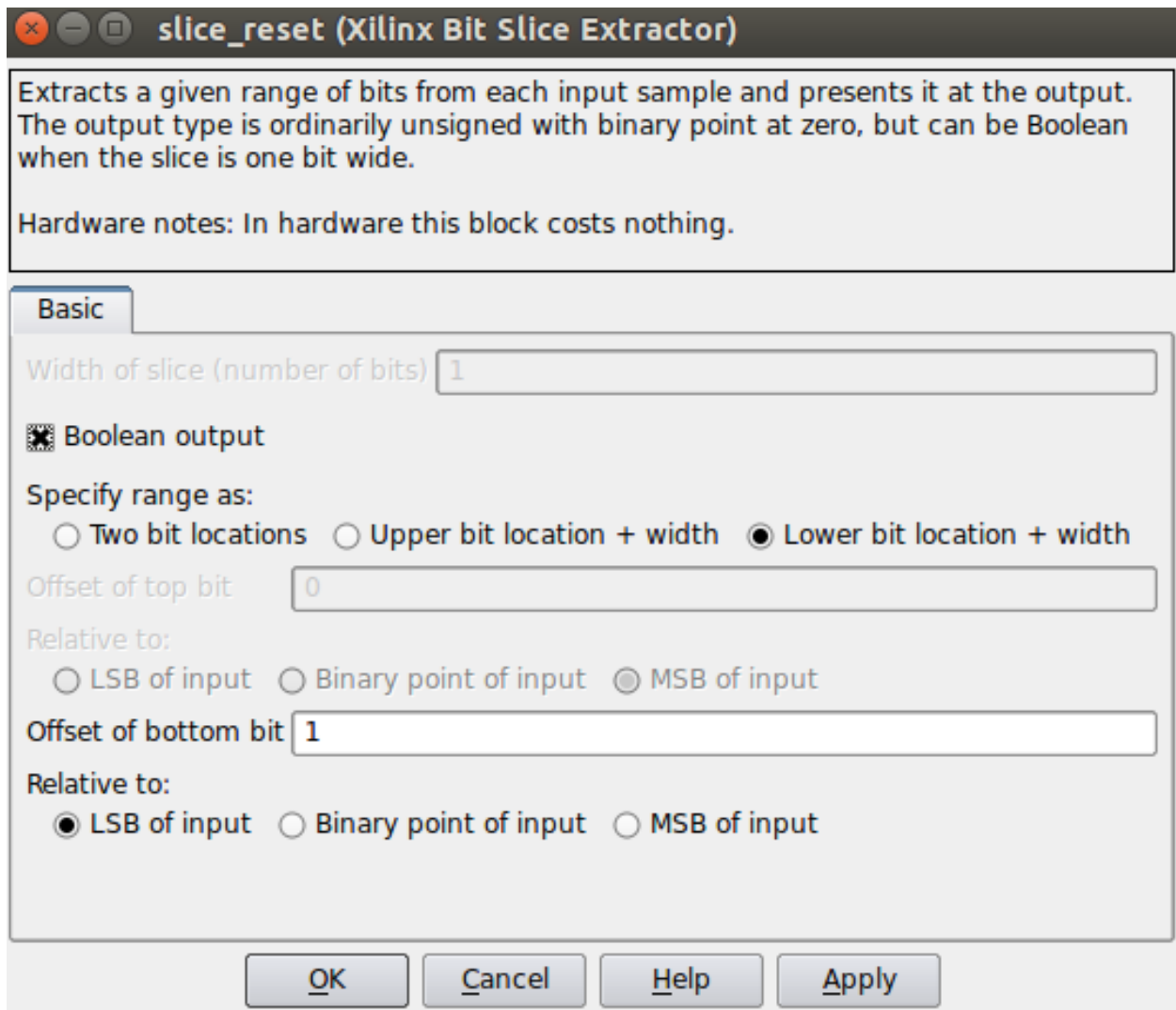
The enable and reset ports of the counter require boolean values (which Simulink interprets differently from ordinary 1-bit unsigned numbers). Configure the slices as follows:

Slice for enable:



casper_xps_param

Slice for reset:

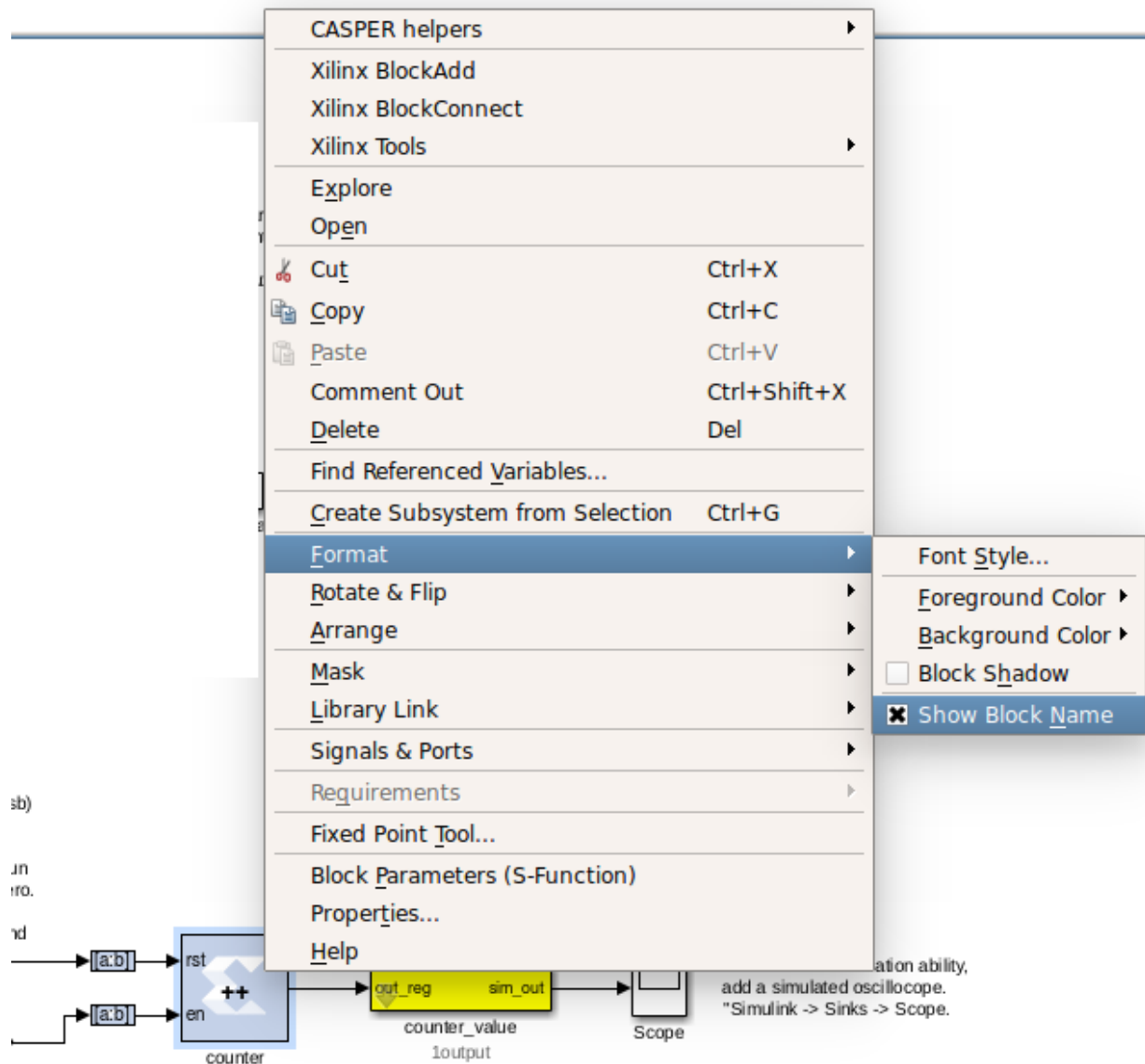


casper_xps_param

Connect 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.

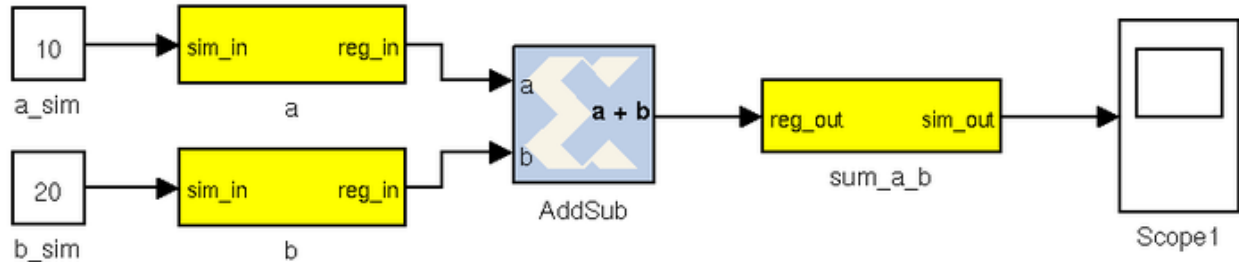
Do so by right-clicking and unchecking Format → Show Block Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



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 $a+b = \text{sum_a_b}$.



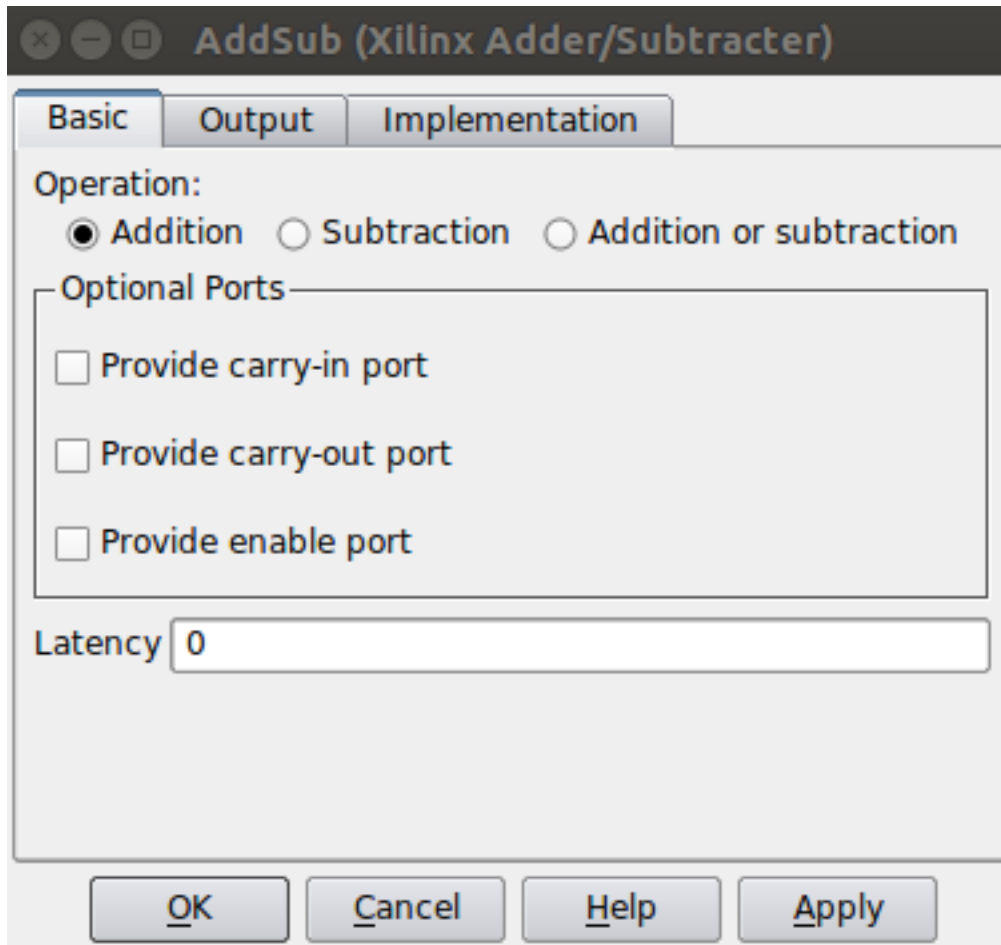
Add the software registers

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.

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.



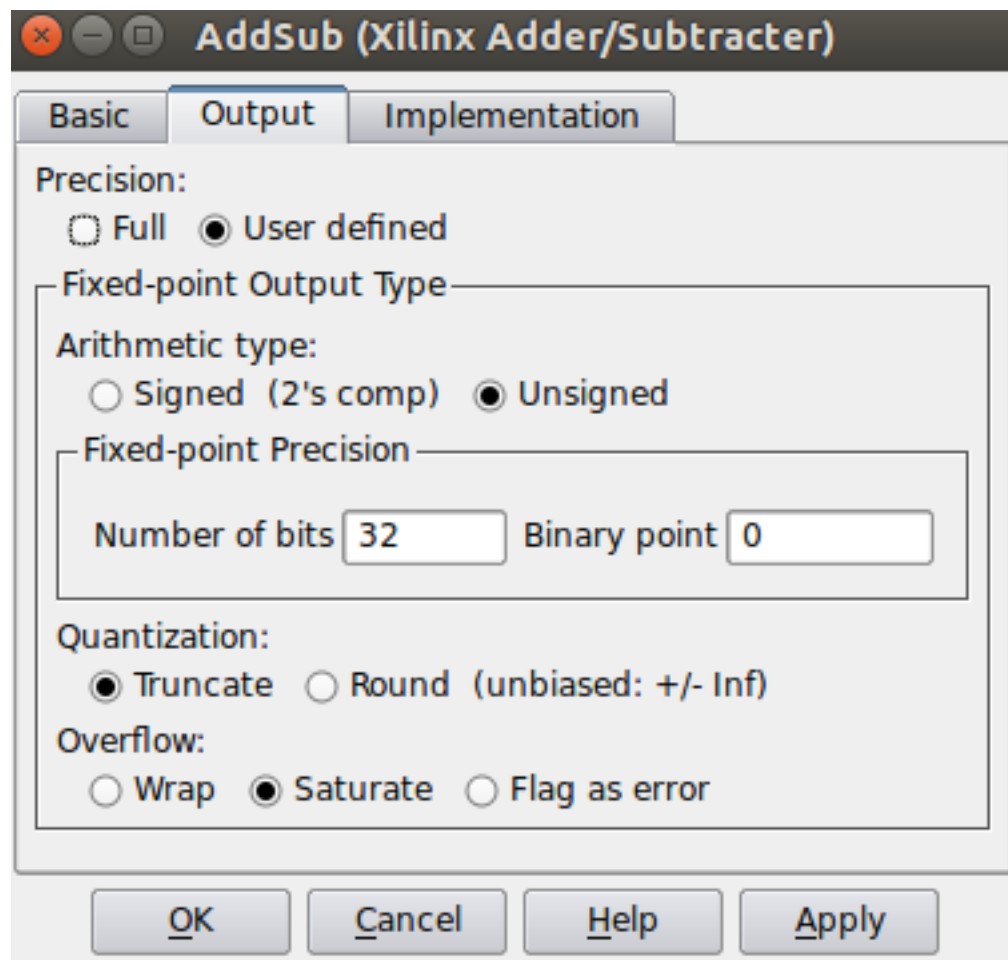
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

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 $2^{32} - 1$.

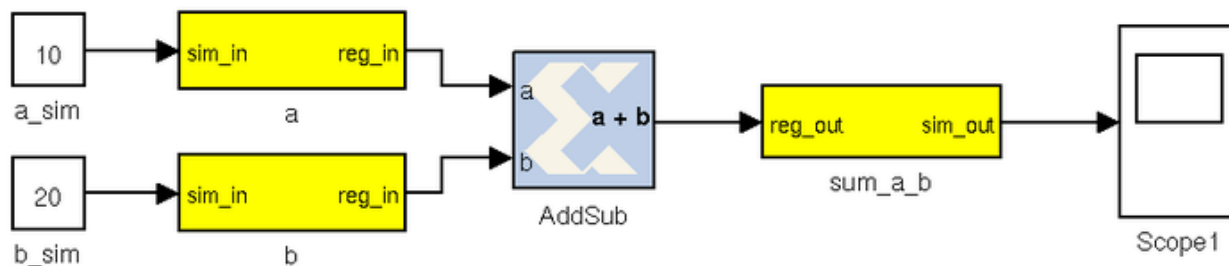


Add the scope and simulation inputs

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.

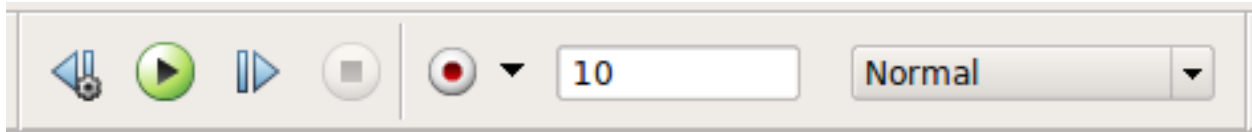
Connect it all together

Like this:



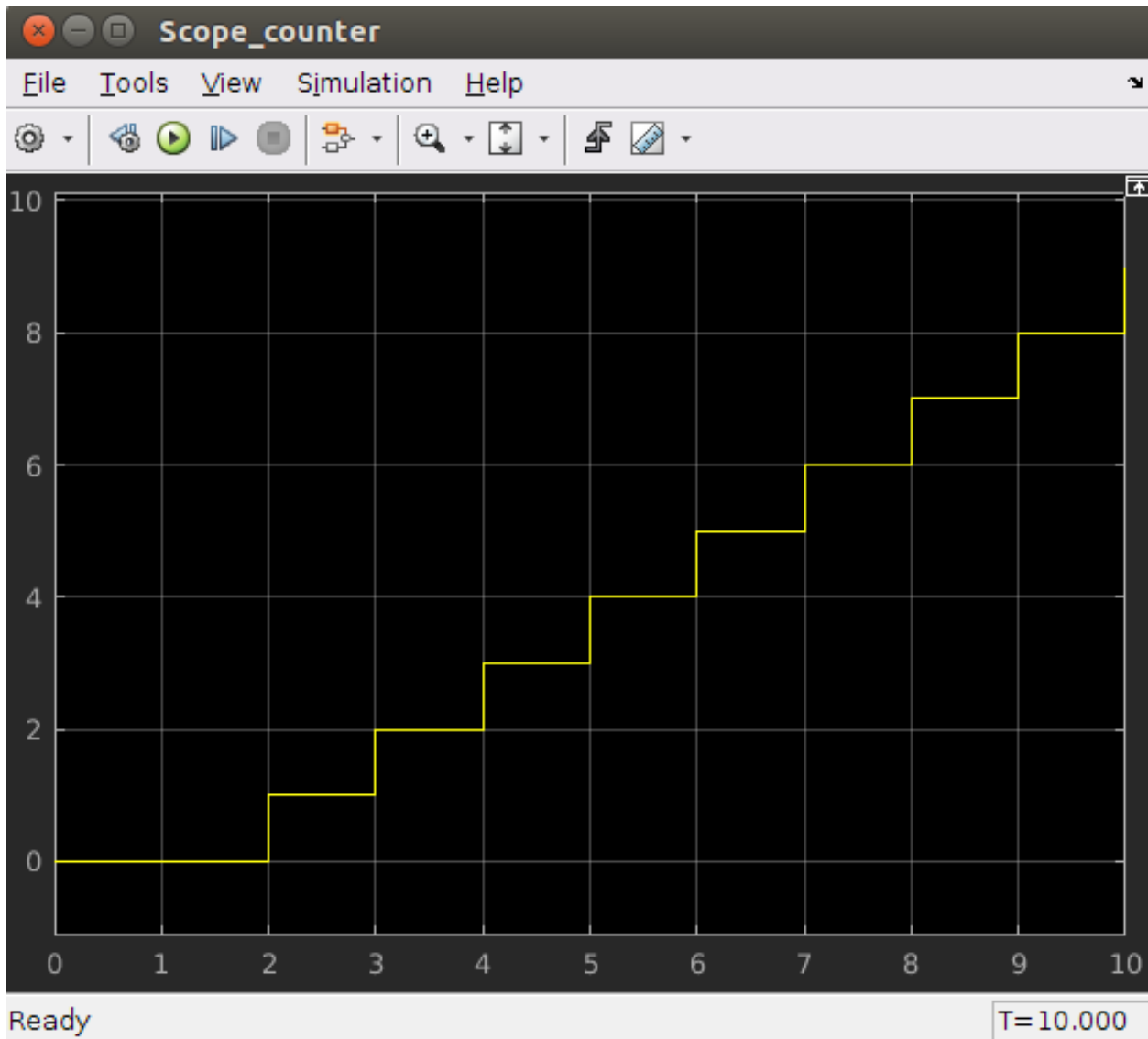
Simulating

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.

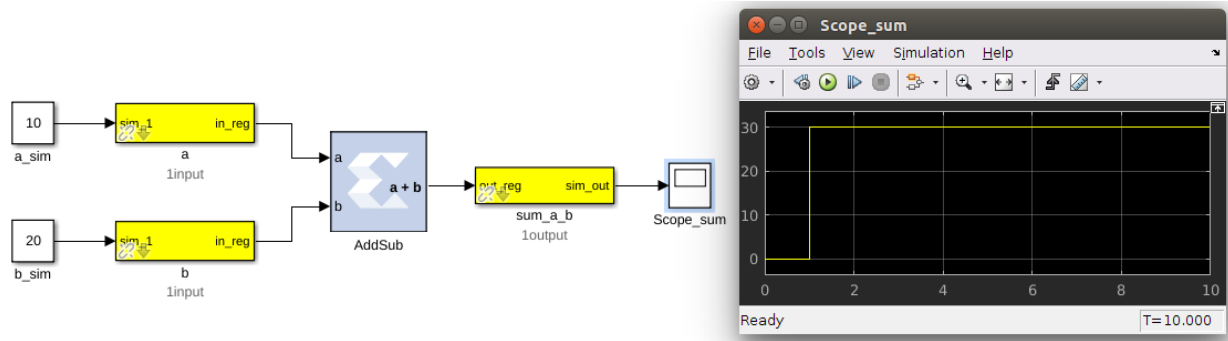


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

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...

Compiling

Essentially, you have constructed three completely separate little instruments.

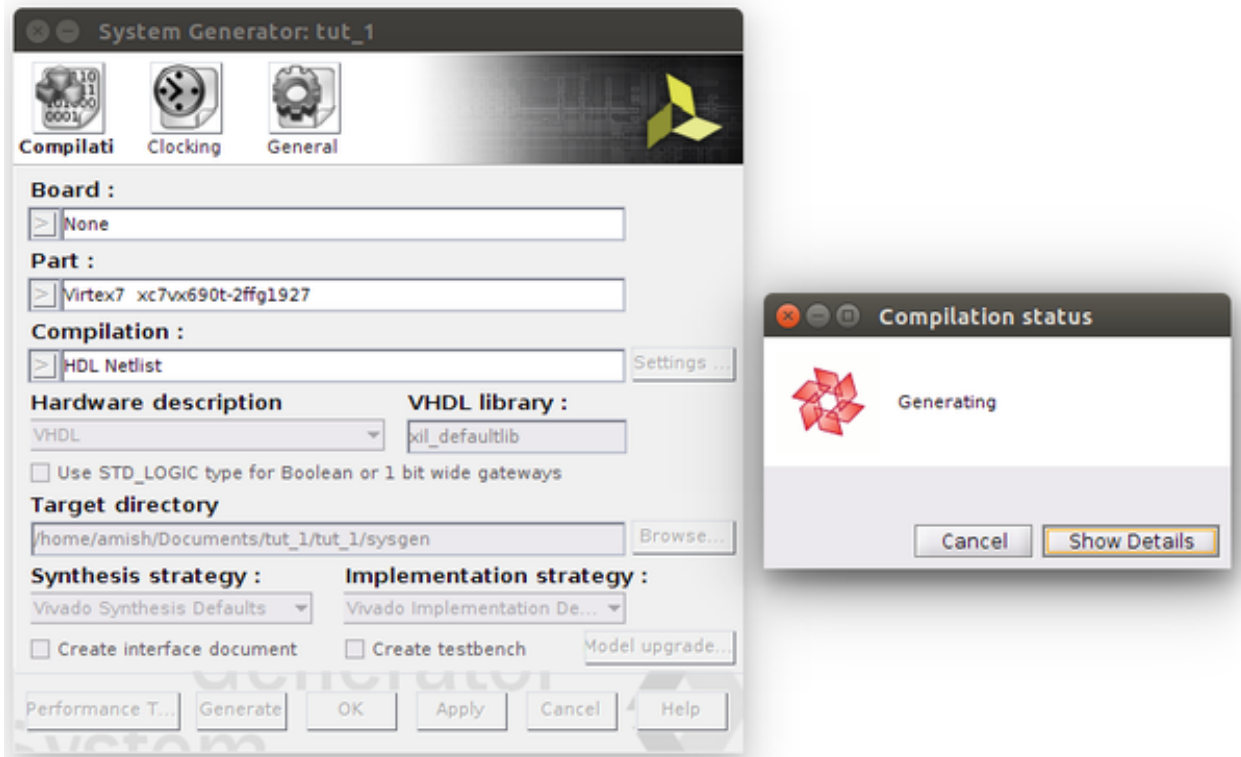
1. You have a flashing LED,
2. A counter which you can start/stop/reset from software, and
3. A simple adder.

These components are all clocked off the same clock source specified in your platform's properties, but they will operate independently.

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window. **THIS COMMAND DEPENDS WHICH PLATFORM YOU ARE TARGETING:**

```
>> jasper
```

When a GUI pops up, click "Compile!". This will run the complete build process, which consists of two stages. The first involving Xilinx's System Generator, which compiles any Xilinx blocks in your Simulink design to a circuit which can be implemented on your FPGA. While System Generator is running, you should see the following window pop up:



After this, the second stage involves synthesis of your design through Vivado, which goes about turning your design into a physical implementation and figuring out where to put the resulting components and signals on your FPGA. Finally the toolflow will create the final output fpg file that you will use to program your FPGA. This file contains the bitstream (the FPGA configuration information) as well as meta-data describing what registers and other Yellow Blocks are in your design. This file will be created in the ‘outputs’ folder in the working directory of your Simulink model. **Note: Compile time is approximately 15-20 minutes.**

```
$ cd red_pitaya/tut_intro/red_pitaya_tut_intro/outputs/
$ ls
red_pitaya_tut_intro-<datetime>.fpg red_pitaya_tut_intro-<datetime>.bof
```

Advanced Compiling

Once you are familiar with the CASPER toolflow, you might find you want to run the two stages of the compile separately. This means that MATLAB will become usable sooner, since it won’t be locked up by the second stage of the compile. If you want to do this, you can run the first stage of the compile from the MATLAB prompt with

```
>> jasper_frontend
```

After this is completed, the last message printed will tell you how to finish the compile. It will look something like:

```
$ python /path_to/mlib_devel/jasper_library/exec_flow.py -m /home/user/path_to/red_
pitaya/tut_intro/red_pitaya_tut_intro.slx --middleware --backend --software
```

You can run this command in a separate terminal after navigating to the tutorials_devel/vivado_2018/ directory and sourcing appropriate environment variables.

```
$ source startsg.local.hpw2019
$ source startsg
$ python /path_to/mlib_devel/jasper_library/exec_flow.py -m /home/user/path_to/red_
→pitaya/tut_intro/red_pitaya_tut_intro.slx --middleware --backend --software
```

Programming the FPGA

Reconfiguration of CASPER FPGA boards is achieved using the casperfpga python library, created by the SA-SKA group.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration. However, should you want to run these tutorials on your own machines, you should download the latest casperfpga libraries from [here](#).

Copy your .fpg file to your Server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server. Instructions to do this are available [here](#).

Connecting to the board

SSH into the server that the Red Pitaya board is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
$ ipython
```

Now import the fpga control library. This will automatically pull-in the KATCP library and any other required communications libraries.

```
import casperfpga
```

To connect to the board we create a CasperFpga instance; let's call it fpga. The CasperFpga constructor requires just one argument: the IP hostname or address of your FPGA board.

```
fpga = casperfpga.CasperFpga('red_pitaya_hostname or ip_address')
```

The first thing we do is program the FPGA with the .fpg file which your compile generated.

```
fpga.upload_to_ram_and_program('<your_fpgfile.fpg>')
```

Should the execution of this command return true, you can safely assume the FPGA is now configured with your design. You should see the LED on your board flashing. Go check! All the available/configured registers can be displayed using: `fpga.listdev()`. The adder and counter can be controlled by [writing to](#) and [reading from](#) registers added in the design using:

```
fpga.write_int('a',10)
fpga.write_int('b',20)
fpga.read_int('sum_a_b')
```

With any luck, the sum returned by the FPGA should be correct.

You can also try writing to the counter control registers in your design. You should find that with appropriate manipulation of the control register, you can make the counter start, stop, and return to zero.

```
fpga.write_int('counter_ctrl', 1)
fpga.read_uint('counter_value')
```

Conclusion

This concludes the first CASPER Tutorial. You have learned how to construct a simple Simulink design, program an FPGA board and interact with it with Python using `casperfpga`. Congratulations!

While the design you made might not be very complicated, you now have the basic skills required to build more complex designs which are covered in later tutorials.

1.1.11 Tutorial 2 : ADC and DAC Interface

AUTHORS: A. Isaacson

EXPECTED TIME: 2-4 hours

Introduction

In this tutorial, you will create a simple Simulink design which interfaces to both the dual channel ADC and interleaved DAC that are utilised on the Red Pitaya 125-10 boards - refer to [Red Pitaya Docs: ReadtheDocs](#) and [Red Pitaya Docs: Github](#) for more information. In addition, we will learn to control the design remotely, using a supplied Python library for KATCP.

In this tutorial, the user will be able to input a source sinusoidal signal from the signal generator on either channel 1 or channel 2 of the ADC input and be able to monitor these signals on the channel 1 and channel 2 DAC outputs using an oscilloscope. The user will be able to change the frequency and amplitude of the signal generator and notice the expected change on the oscilloscope if the Simulink design is connected correctly, as explained below. The user can verify the ADC digital data by using a logic analyser configured to read signed data and/or Simulink snap shots, which can be used to capture the ADC data. The python scripts can be utilised to read back this captured data and there are matlab scripts which can be utilised to display this data. The user will also be able to see what happens to the ADC and DAC data when the Digital Signal Processing (DSP) clock is increased from 125MHz to 200MHz.

Required Equipment

The following equipment is required:

1. Oscilloscope, 50MHz, +/-2Vpp, Qty 1
2. Signal Generator, 1MHz-50MHz, +/-2Vpp, Qty 1
3. Logic Analyzer, 1GSps, 2 pods with fly leads, Qty 1
4. SMA Cables, Male to Male, Qty 2

Background

The Red Pitaya 125-10 board consists of a dual channel 10 bit ADC and DAC - refer to [Red Pitaya Docs: ReadtheDocs](#) and [Red Pitaya Docs: Github](#) for more information on the Red Pitaya. There are currently two versions of the Red Pitaya (125-10 and the 125-14) - refer to [Red Pitaya Hardware Comparison](#) for differences between the boards.

The Red Pitaya 125-10 is fitted with a single Analog Devices dual channel ADC AD9608 device. The ADC samples at 125MSPS and each digital ADC output channel is 10 bits wide, 1.8V LVCMOS. Refer to [Red Pitaya Docs: Github](#) for the schematics of the 125-10. The ADC output is offset binary and converted to two's complement in the firmware running on the Zynq programmable logic (PL).

The Red Pitaya 125-10 is fitted with a single Analog Devices dual channel DAC AD9767 device. The DAC digital input is offset binary, 10 bits, LVCMOS 3.3V and is converted from two's complement inside the firmware running on the Zynq programmable logic. The second channel of the DAC is not connected, which means the DAC is utilised in the interleave mode - refer to DAC data sheet [Red Pitaya Docs: Github](#).

Feel free to spend some time reading the data sheets and looking at the schematics. This will be important for the bonus challenge at the end of the tutorial.

Create a new model

Start Matlab and open Simulink (either by typing 'simulink' on the Matlab command line, or by clicking on the Simulink icon in the taskbar). A template is provided for this tutorial with a pre-created firmware flashing LED function and Red Pitaya XSG core config or platform block and Xilinx System Generator block. Get a copy of this template and save it. Make sure the Red Pitaya XSG_core_config_block or platform block is configured for:

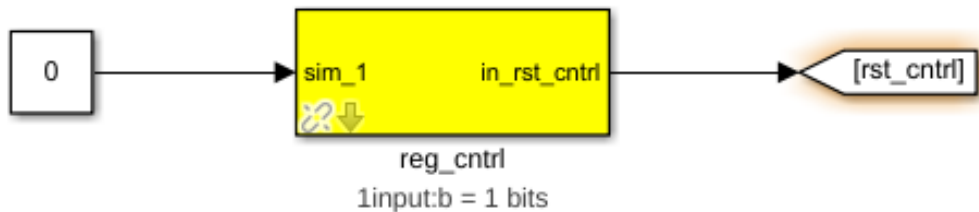
1. Hardware Platform: "RED_PITAYA:xc7z010"
2. User IP Clock source: "sys_clk"
3. User IP Clock Rate (MHz): 125 (125MHz clock derived from 125MHz ADC on-board clock). This clock domain is used for the Simulink design

The rest of the settings can be left as is. Click OK.

Add Reset logic

A very important piece of logic to consider when designing your system is how, when and what happens during reset. In this example we shall control our resets via a software register. We shall have one reset to reset the Simulink design counters and trigger the data capture snap blocks. Construct reset and control circuitry as shown below.

ADC/DAC SOFTWARE RESET CONTROL



Add a software register

Use a software register yellow block from the CASPER XPS Blockset -> Memory for the reg_cntrl block. Rename it to reg_cntrl. Configure the I/O direction to be From Processor. Attach one Constant block from the Simulink -> Sources section of the Simulink Library Browser to the input of the software register and make the value 0 as shown above.

Add Goto Block

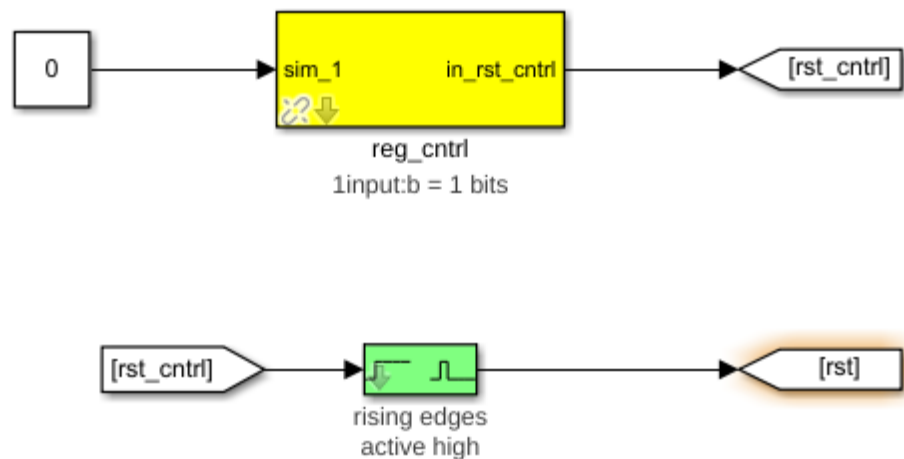
Add one Goto block from Simulink -> Signal Routing. Configure them to have the tags as shown (rst_cntrl). These tags will be used by associated From (also found in Simulink -> Signal Routing) blocks in other parts of the design. These help to reduce clutter in your design and are useful for control signals that are routed to many destinations. They should not be used a lot for data signals as it reduces the ease with which data flow can be seen through the system.

Add Edge_Detect block

Add From blocks from Simulink -> Signal Routing rst_cntrl should go through an edge_detect block (rising and active high) to create a pulsed rst signal, which is used to trigger and reset the counters in the design. This is located in CASPER DSP Blockset -> Misc. Add Goto block rst from Simulink -> Signal Routing.

It should look as follows when you have added all the relevant registers:

ADC/DAC SOFTWARE RESET CONTROL

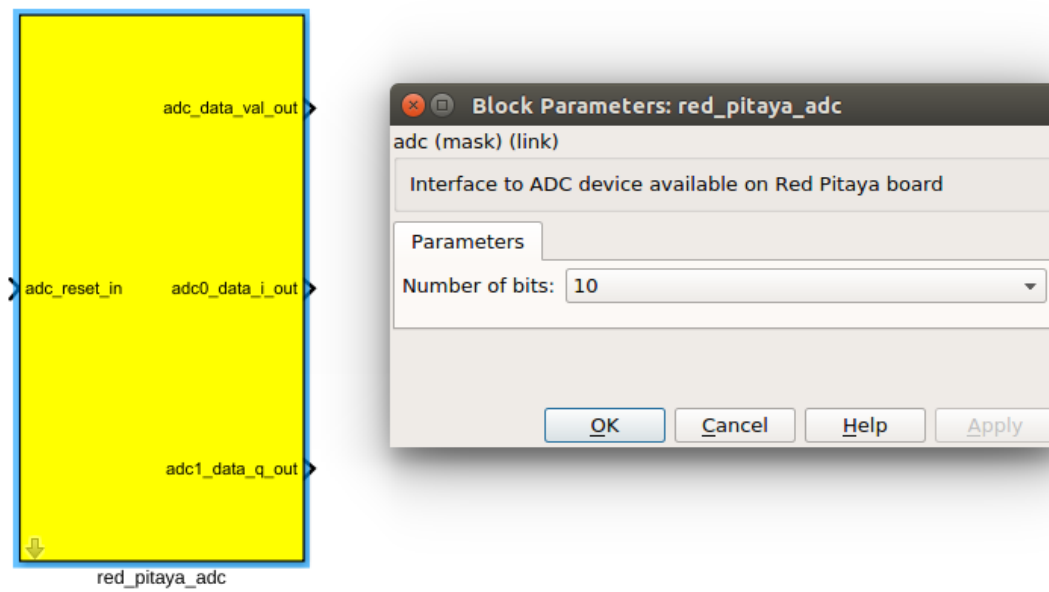


Add ADC and associated registers and gpio for debugging

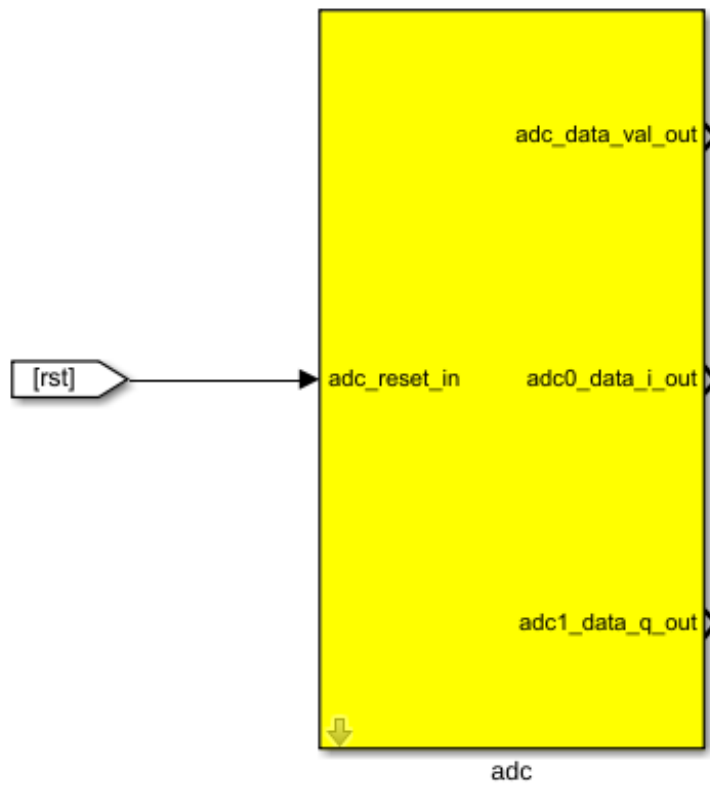
We will now add the ADC yellow block in order to interface with the ADC device on the Red Pitaya.

Add the ADC yellow block for digital to analog interfacing

Add a Red Pitaya ADC yellow block from the CASPER XPS Blockset -> ADCs, as shown below. It will be used to interface to the ADC device on the Red Pitaya. Rename it to `adc`. Double click on the block to configure it and set the number of bits to be 10 bits wide. This should be hard coded for now. This will need to be changed for the bonus challenge exercise below.

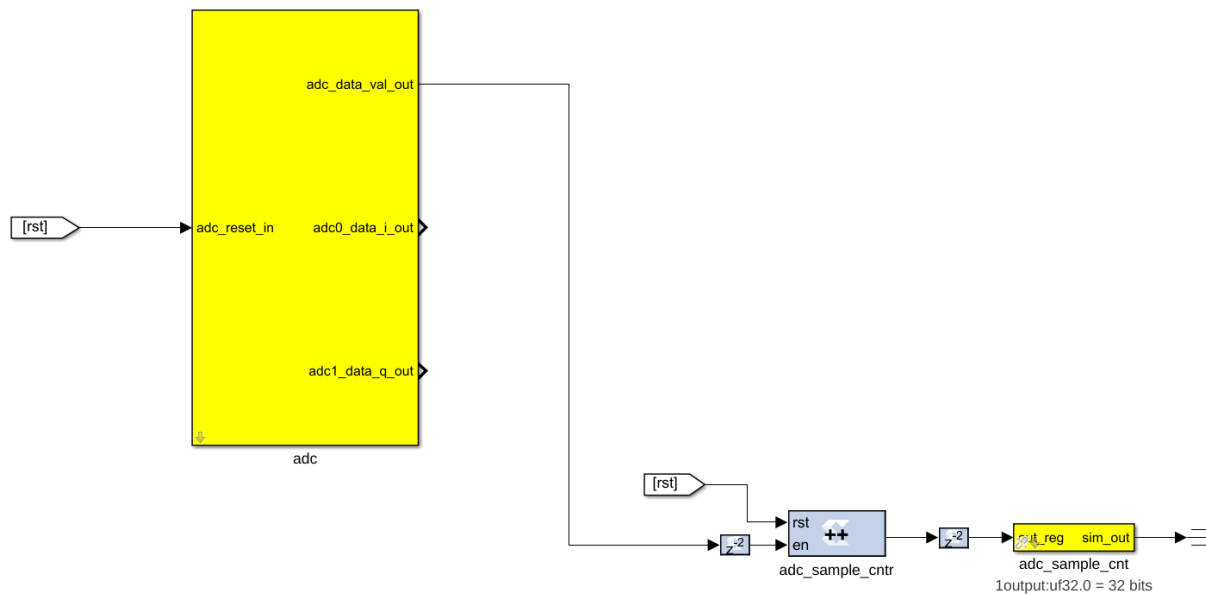


Add the From block as shown below, which connects the reset to the ADC yellow block.

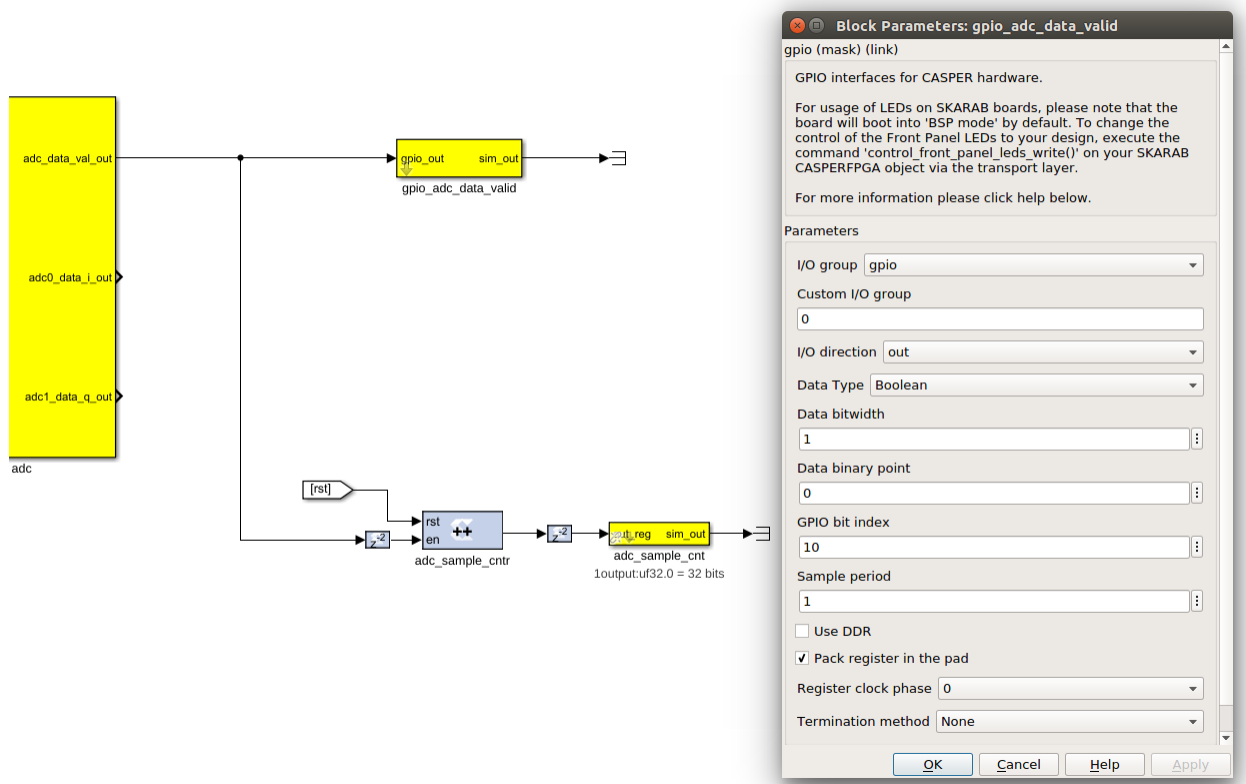


Add registers and gpio to provide ADC debugging

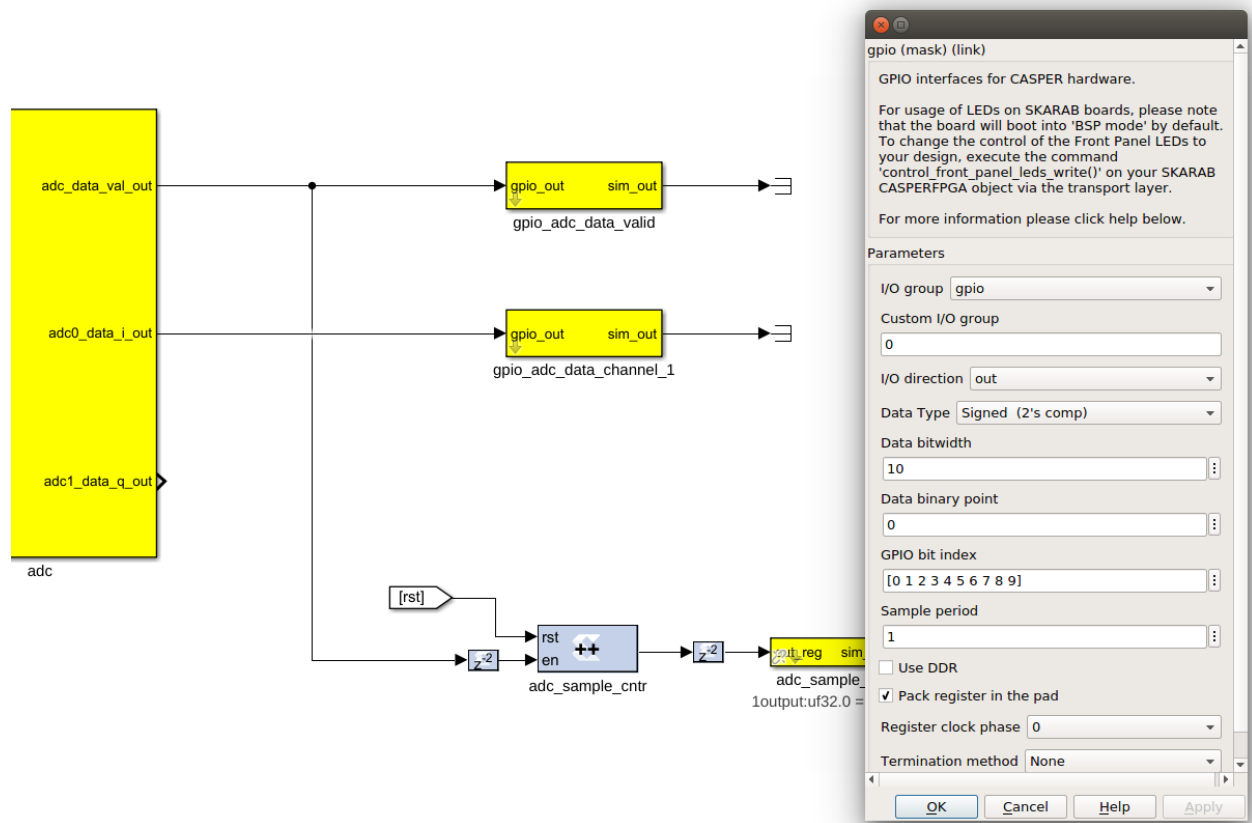
Add one yellow-block software register to provide an ADC sample counter (32 bits). Name it as shown below. The register should be configured to send its value to the processor. Connect them to the ADC yellow block as shown below. Delay blocks are added through the design to show the different types of Xilinx Simulink blocks that are available, but they are not really needed - you will find these under Xilinx Blockset -> Basic Elements in the Simulink Library Browser.



In the event that you will be using the logic analyser, you will need to route the ADC data (channel 1) to the Logic Analyser connector (E1) on the Red Pitaya. In order to do this you will need to add two gpio yellow-blocks from CASPER XPS Blockset -> IO. The first GPIO is for the ADC data valid and the second GPIO yellow block is for the ADC channel 1 10 bit data. It should be connected as shown below for the first gpio for the ADC data valid:



It should be connected as shown below for the second gpio for the ADC 10 bit data on channel 1:



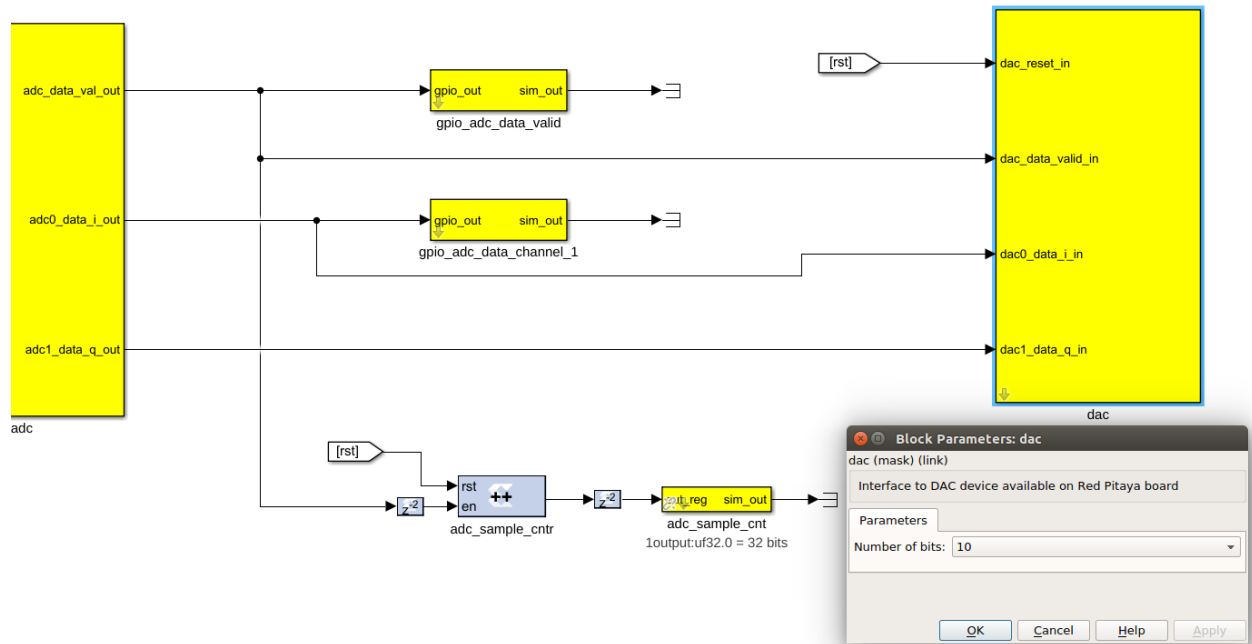
You will now be able to monitor the ADC data using the logic analyser connected to E1 on the Red Pitaya - refer to 125-10 schematics [Red Pitaya Docs: Github](#).

Add DAC

We will now add the DAC yellow block in order to interface with the DAC device on the Red Pitaya.

Add the DAC yellow block for digital to analog interfacing

Add a Red Pitaya DAC yellow block from the CASPER XPS Blockset -> DACs, as shown below. It will be used to interface to the DAC device on the Red Pitaya. Rename it to dac. Double click on the block to configure it and set the number of bits to be 10 bits wide. This should be hard coded for now. This will need to be changed for the bonus challenge exercise below. Add the From block, which connects the reset to the DAC yellow block and connect the ADC to the DAC as shown below.

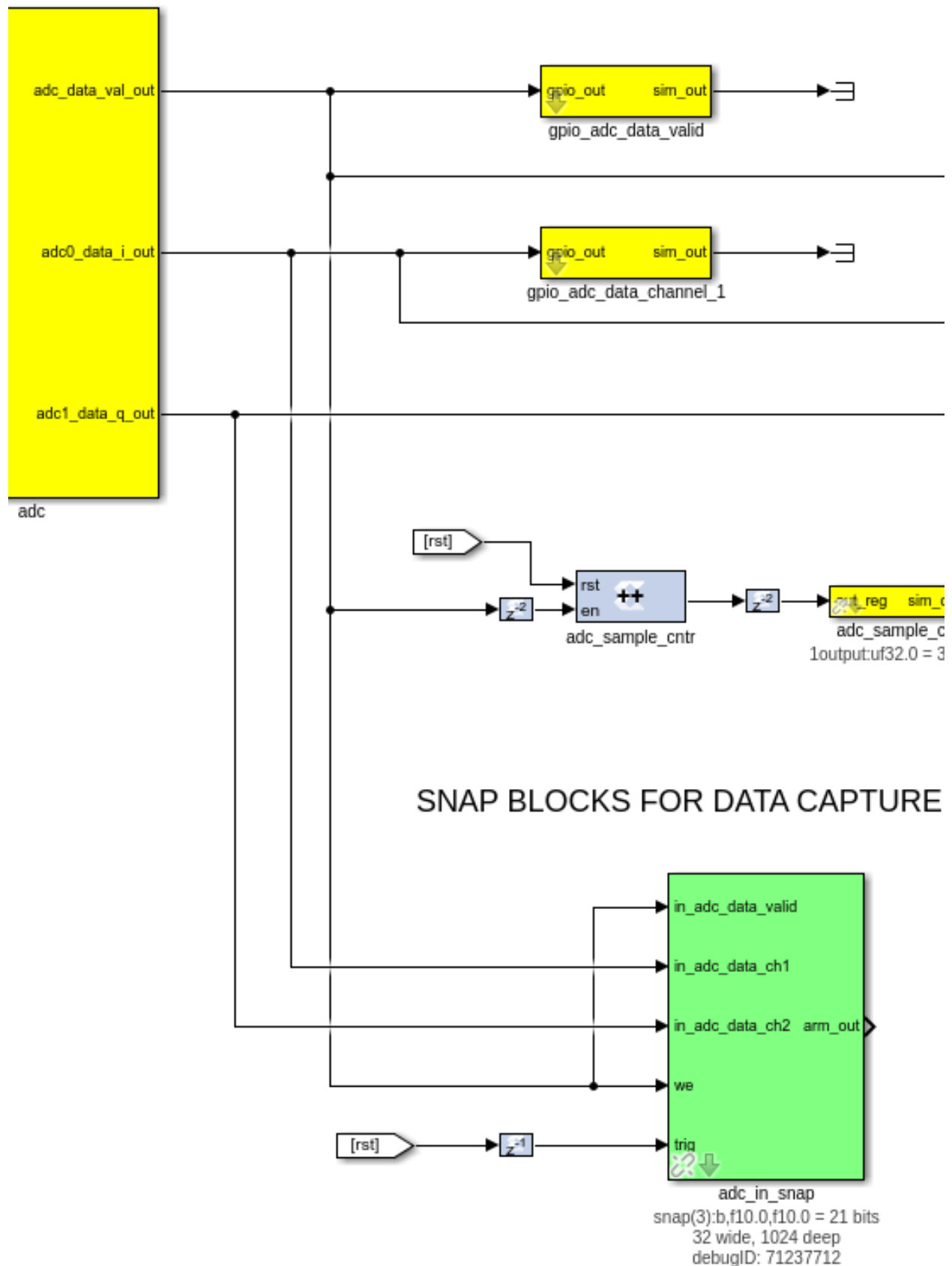


The ADC and DAC functionality is now implemented in your Simulink design. It is now time to setup the ADC data capturing snap shot blocks.

Buffers to capture ADC Data Valid, ADC Channel 1 and ADC Channel 2

The ADC data valid, ADC data channel 1 and ADC data channel 2 (output) need to be connected to a bitfield snapshot block for data capture analysis (located in CASPER DSP Blockset -> Scopes), as shown below. Using this block, we can capture 1024 ADC samples, read it back and store them as files. These files can then be plotted using a Matlab script and analysed.

Bitfield snapshot blocks are a standard way of capturing snapshots of data in the CASPER tool-set. A bitfield snap block contains a single shared BRAM allowing capture of 32-bit words (this is a limitation of the new AXI interfacing at the moment).



The ctrl register in a snap block allows control of the capture. The least significant bit enables the capture. Writing a rising edge to this bit primes the snap block for capture. The 2nd least most significant bit allows the choice of a trigger source. The trigger can come from an external source or be internal and immediately. The 3rd most least significant bit allows you to choose the source of the valid signal associated with the data. This may also be supplied externally or be immediately enabled.

The basic principle of the snap block is that it is primed by the user and then waits for a trigger at which point it captures a block of data and then waits to be primed again. Once primed the addr output register returns an address of 0 and will increment as data is written into the BRAMs. Upon completion the addr register will contain the final address. Reading this value will show that the capture has completed and the results may be extracted from the shared BRAMs.

In the case of this tutorial, the arming and triggering is done via software. The trigger is the rst signal. The “we” signal on the snapshot blocks is the ADC data valid signal. Configure and connect the snap blocks as shown above. The delay is not really necessary, but if your data was not aligned then you could use delays for this. The following settings should be used for the bitfield snapshot blocks: storage medium should be BRAM, number of samples (“2^?”) should be 10, Data width 32, all boxes unchecked except “use DSP48s to implement counters”, Pre-snapshot delay should be 0.

You should now have a complete Simulink design. Compare it with the complete ADC and DAC interface tutorial *.slx model provided to you before continuing if unsure.

Compilation

Press CTRL+D to compile the tutorial first and make sure there are no errors before compiling. If there are any errors then a diagnostic window will pop up and the errors can be addressed individually.

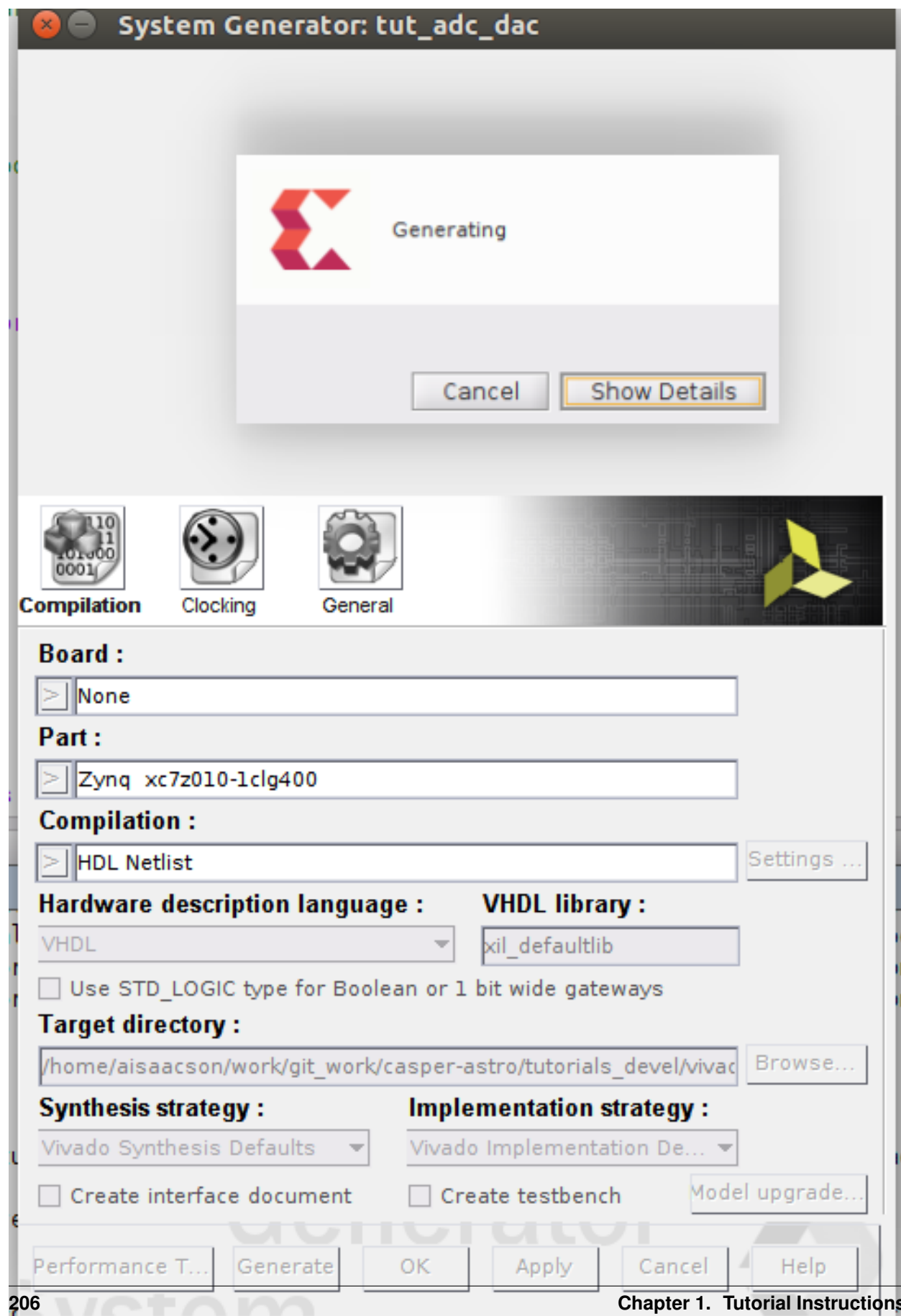
It is now time to compile your design into an FPGA bitstream. This is explained below, but you can also refer to the Jasper How To document for compiling your toolflow design. This can be found in the ReadtheDocs mlib_devel documentation link:

<https://casper-toolflow.readthedocs.io>

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window:

```
jasper_frontend
```

This will enable Vivado’s system generator to be run, and the windows below should pop up with the name of your slx file in the window instead of tut_1. The same applies below in the output file path - tut_1 will be replaced with the name of your slx file. In my case it is “tut_adc_dac”.



The next step is to open a terminal session (ctrl+T) and change directory to where your git clone of the tutorials_devel is located “./tutorials_devel/vivado_2018/red_pitaya” and source the “startsg startsg.local” script as shown below:

```

alsaacson@adam-cm: ~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya
8 files changed, 23 insertions(+), 23 deletions(-)
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/adc_sample_cnt.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/adc_yellow_block_with_reset.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/adc_yellow_clock_param.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/dac_yellow_clock_param.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/gpio_adc_data_channel_1.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/gpio_adc_data_valid.png
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ git push origin workshop2019
Username for 'https://github.com': AdamI75
Password for 'https://AdamI75@github.com':
Counting objects: 20, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (20/20), 341.20 KiB | 0 bytes/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/casper-astro/tutorials_devel.git
fd3973b..edbb170 workshop2019 -> workshop2019
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ cd vivado
vivado/ vivado_2018/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ cd vivado
vivado/ vivado_2018/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya$ source startsg startsg.local.hartrao2019

```

The next step is to cut the python text from the bottom of the MATLAB Command Line Window and paste it into your terminal and press enter. It should look as follows:

```

alsaacson@adam-cm: ~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/adc_yellow_block_with_reset.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/adc_yellow_clock_param.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/dac_yellow_clock_param.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/gpio_adc_data_channel_1.png
create mode 100644 docs/_static/img/red_pitaya/tut_adc_dac/gpio_adc_data_valid.png
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ git push origin workshop2019
Username for 'https://github.com': AdamI75
Password for 'https://AdamI75@github.com':
Counting objects: 20, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (17/17), done.
Writing objects: 100% (20/20), 341.20 KiB | 0 bytes/s, done.
Total 20 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), completed with 5 local objects.
To https://github.com/casper-astro/tutorials_devel.git
fd3973b..edbb170 workshop2019 -> workshop2019
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ cd vivado
vivado/ vivado_2018/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel$ cd vivado
vivado/ vivado_2018/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya$ vim startsg.local.hartrao2019
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya$ python /home/alsaacson/work/git_work/ska_sa/p
rojects/mlib_devel/jasper_library/exec_flow.py -m /home/alsaacson/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/
tut_adc_dac/tut_adc_dac.s1x --middleware --backend --software

```

This will run the process to generate the FPGA bitstream and output Vivado compile messages to the terminal along the way.

Execution of this command will result in an output .bof and .fpg file in the ‘outputs’ folder in the working directory of your Simulink model. Note: Compile time is approximately 10-15 minutes, so a pre-compiled binary (.fpg file) is made available to save time.

```

alsaacson@adam-cm: ~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac/tut_adc_dac/outputs
drwxrwxr-x 4 alsaacson alsaacson 4096 Jun 25 10:43 tut_adc_dac
-rw-r--r-- 1 alsaacson alsaacson 93806 Jun 25 10:26 tut_adc_dac.slx
-rw-r--r-- 1 alsaacson alsaacson 87110 Jun 25 10:43 tut_adc_dac.slx.autosave
-rw-rw-r-- 1 alsaacson alsaacson 991 Jun 25 10:32 tut_adc_dac_sysgen_error.log
-rw-r--r-- 1 alsaacson alsaacson 39782 Jun 24 15:24 tut_adc_dac_template.slx
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac$ cd tut_adc_dac/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac/tut_adc_dac$ ls -la
total 44
drwxrwxr-x 4 alsaacson alsaacson 4096 Jun 25 10:43 .
drwxrwxr-x 3 alsaacson alsaacson 4096 Jun 25 10:43 ..
-rw-rw-r-- 1 alsaacson alsaacson 4391 Jun 25 10:42 design_info.tab
-rw-rw-r-- 1 alsaacson alsaacson 873 Jun 25 10:42 git_info.tab
-rw-rw-r-- 1 alsaacson alsaacson 461 Jun 25 10:42 jasper.log
-rw-rw-r-- 1 alsaacson alsaacson 10764 Jun 25 10:42 jasper.per
drwxrwxr-x 2 alsaacson alsaacson 4096 Jun 25 11:09 outputs
drwxrwxr-x 5 alsaacson alsaacson 4096 Jun 25 10:43 sysgen
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac/tut_adc_dac$ cd outputs/
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac/tut_adc_dac/outputs$ ls -la
total 2112
drwxrwxr-x 2 alsaacson alsaacson 4096 Jun 25 11:09 .
drwxrwxr-x 4 alsaacson alsaacson 4096 Jun 25 10:43 ..
-rw-rw-r-- 1 alsaacson alsaacson 2088160 Jun 21 11:24 tut_adc_dac_2019-06-21_1117.bof
-rw-rw-r-- 1 alsaacson alsaacson 62634 Jun 21 11:24 tut_adc_dac_2019-06-21_1117.fpg
alsaacson@adam-cm:~/work/git_work/casper-astro/tutorials_devel/vivado_2018/red_pitaya/tut_adc_dac/tut_adc_dac/outputs$

```

Programming the FPGA (Zynq PL)

Reconfiguration of the Red Pitaya's Zynq is done via the `casperfpga` python library. The `casperfpga` package for python, created by the SA-SKA group, wraps the Telnet commands in python. and is commonly used in the CASPER community. We will focus on programming and interacting with the Programmable Logic (PL) of the Zynq using this method.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration, but if you are not working from the lab then refer to the [How To Setup CasperFpga Python Packages](#) document for setting up the python libraries for `casperfpga`. This can be found in the “`casperfpga`” github repo located on GitHub and the ReadtheDocs `casperfpga` documentation link:

<https://github.com/casper-astro/casperfpga>

<https://casper-toolflow.readthedocs.io>

Copy your .fpg file to your NFS server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server.

```

scp path_to/your/model_folder/your_model_name/outputs/your_fpgfile.fpg user@server:/
↳path/to/test/folder/

```

Connecting to the board

SSH into the server that the Red Pitaya is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
ipython
```

Now import the Zynq control library. This will automatically pull-in the KATCP library and any other required communications libraries.


```
import casperfpga
```

To connect to the Red Pitaya we create an instance of the Red Pitaya board; let's call it `rp`. The wrapper's `fpgaclient` initiator requires just one argument: the IP hostname or address of the Red Pitaya board. The hostname of the Red Pitaya board can be found by looking at the top of Red Pitaya Ethernet connector e.g. `RP-f0495e.local` is the host name on my board. If you ping your host name from the terminal then you will see the IP address. You can do this from `ipython` by adding “! ping”.

```
rp = casperfpga.CasperFpga(host='red_pitaya_name or ip_address', port=7147)
```

The first thing we do is configure the FPGA (Zynq PL).

```
rp.upload_to_ram_and_program('your_fpgfile.fpg')
```

You should notice that LED0 on the Red Pitaya is flashing, which indicates that the firmware is running. Congratulations, you just configured your Red Pitaya successfully! :).

All the available/configured registers can be displayed using:

```
rp.listdev()
```

The Zynq PL is now configured with your design. The registers can now be read back. For example, the ADC sample count register can be read back from the FPGA by using:

```
rp.read_uint('adc_sample_cnt')
```

The value returned should be continually incrementing, which indicates that the ADC is successfully sampling the input data.

If you need to write to the `reg_cntrl` register then do the following:

```
rp.registers.reg_cntrl.write(rst_cntrl = 'pulse'), this creates a pulse on the rst_
↪signal
```

Try the following - add a “?” (leave out the brackets) to find out what the functions below does:

```
rp.is_connected()
rp.is_running()
```

Manually typing these commands by hand will be cumbersome, so it is better to create a Python script that will do all of this for you. This is described below.

Running a Python script and interacting with the Zynq PL

A pre-written python script, “`tut_adc_dac.py`” is provided. The code within the python script is well commented and won't be explained here. The user can read through the script in his/her own time. In summary, this script programs the Zynq PL with your compiled design (`.fpg` file), writes to the reset control register, reads back the ADC snap shot captured data and status registers while displaying them to the screen for analysis. It is also saves the ADC data as text data for displaying in Matlab. In order to run this script you will need to edit the file and change the target Red Pitaya IP address and the `*.fpg` file, if they are different. The script is run using:

```
python tut_adc_dac.py <red pitaya hostname/ip> -p -b <path to fpg file>
```

If everything goes as expected, you should see a whole bunch of text on your screen - this is the output of the snapshot block and status register contents.

Analysing the Display Data

You should see something like this:

```
user@server:~$ python tut_adc_dac.py
connecting to Red Pitaya...
done
programming the Red Pitaya...
done
arming snapshot block...
done
triggering the snapshot and reset the counters...
done
reading the snapshot...
done
writing ADC data to disk ...
done
reading back the status registers...
adc_sample_cnt: 3689577
done
Displaying the snapshot block data...
ADC SNAPSHOT CAPTURED INPUT
-----
Num adc_data_valid adc_data_ch1 adc_data_ch2
[0] 1 -10 -1
[1] 1 6 -1
[2] 1 -6 -1
[3] 1 0 -1
[4] 1 -1 -2
[5] 1 -3 -2
[6] 1 1 -1
[7] 1 -6 -1
[8] 1 3 -1
[9] 1 -6 -1
[10] 1 3 -1
....
[589] 1 -6 -2
[590] 1 1 -1
[591] 1 -3 -1
[592] 1 -2 -1
[593] 1 1 -1
[594] 1 -5 -1
[595] 1 4 -1
[596] 1 -9 -1
[597] 1 6 -1
[598] 1 -9 -1
[599] 1 5 -1
```

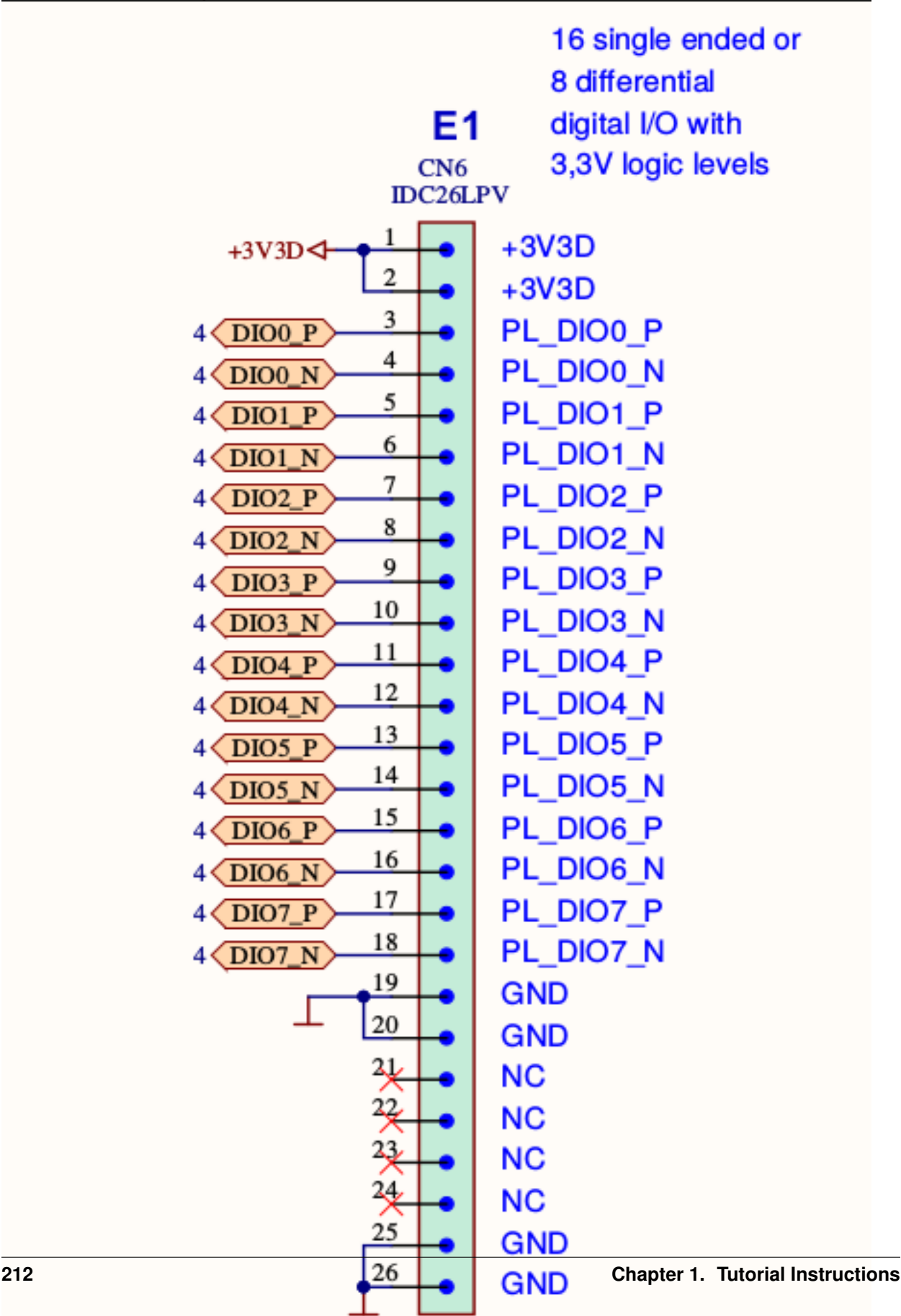
The above results show that the ADC is sampling continuously and that there is just noise at the ADC input, as no signal generator is connected. The ADC sample count should continue to increase each time this script file is run - try it. You can also try and compile your Simulink design with a higher clock e.g. 200MHz. Does the ADC data valid

signal stay high continually? If so, then why?

Setup the signal generator to have an amplitude of 2Vpp, a frequency of 4MHz and sinusoid signal. Check using the oscilloscope that this is correct. Once satisfied connect the one end of the SMA cable to the signal generator output and connect the other end to the Red Pitaya IN1. Remember to also connect the Red Pitaya OUT1 to the other channel of the oscilloscope (needs a BNC to SMA Female) using another SMA to SMA cable. Change the frequency and amplitude to 2MHz and to 1Vpp. What happens to the oscilloscope Red Pitaya OUT1 display? What is the frequency and amplitude now? Try run the python script and see what happens? What is different in your results now?

Now connect the signal generator to Red Pitaya IN2 and the other channel of the oscilloscope to Red Pitaya OUT2 and repeat the above tests. What do you notice?

Remember those gpio yellow blocks you added for the logic analyser? Connect the logic analyser fly leads to connector E1 on the Red Pitaya - do this while the system is powered off and the signal generator is off. Refer to diagram below for E1 connector pin outs:



Hint: Checkout schematics: pin 3 is the LSB and pin 12 is the MSB of the ADC data, pin 13 is the ADC data valid.
Hint: What trigger are you going to use to sample the ADC data on channel 1 - internal or external? Why?

Once you have connected everything up then configure the board and switch on the signal generator. Does the data displayed on the logic analyser look similar to the data captured using the python script? How can you display a sinusoid signal on the logic analyser?

There are 2 plots one for each ADC. Try compute the RMS LSB of each ADC channel. Is there anything besides noise when the channel is unterminated or terminated? Hint: edit the script to compute the standard deviation with the ADC inputs terminated.

Try and do an FFT (add windowing) of the captured ADC data by editing the script. What do you get? Does it make sense? Hint: treat each channel as real data.

Bonus Challenge

You have gone through the ADC and DAC tutorial on the 125-10 board. Now it is time to really test your hardware porting skills and port this tutorial to work on the 125-14 board. Good luck!

Hint: The Zynq pin outs are not the same. You will need to look at the 125-14 schematics. You will need to update the simulink yellow blocks, python blocks and red_pitaya.yaml script.

Other notes

- iPython includes tab-completion. In case you forget which function to use, try typing `library_name.tab`
- There is also onboard help. Try typing `library_name.function?`
- Many libraries have onboard documentation stored in the string `library_name.doc`
- KATCP in Python supports asynchronous communications. This means you can send a command, register a callback for the response and continue to issue new commands even before you receive the response. You can achieve much greater speeds this way. The Python wrapper in the `corr` package does not currently make use of this and all calls are blocking. Feel free to use the lower layers to implement this yourself if you need it!

Conclusion

This concludes the ADC and DAC Interface Tutorial for the Red Pitaya. You have learned how to utilize the ADC and DAC interface on the Red Pitaya to sample incoming data and convert this sampled data to an analog signal. You have learned to play around with Matlab's data manipulation using the signal processing toolbox. You also learned how to further use Python to program the Zynq PL, control the design and debug the design remotely using `casperfpga`. If you managed to successfully port your tutorial to work with the 125-14 Red Pitaya then you would have gained important hardware porting experience that will allow you to port to any hardware platform in the future.

1.1.12 Tutorial 3: Wide(-ish)band Spectrometer

Introduction

A spectrometer is something that takes a signal in the time domain and converts it to the frequency domain. In digital systems, this is generally achieved by utilising the FFT (Fast Fourier Transform) algorithm.

When designing a spectrometer for astronomical applications, it's important to consider the science case behind it. For example, pulsar timing searches will need a spectrometer which can dump spectra on short timescales, so the rate of change of the spectra can be observed. In contrast, a deep field HI survey will accumulate multiple spectra to increase

the signal to noise ratio. It's also important to note that "bigger isn't always better"; the higher your spectral and time resolution are, the more data your computer (and scientist on the other end) will have to deal with. For now, let's skip the science case and familiarize ourselves with an example spectrometer.

Setup

This tutorial comes with a completed model file, a compiled bitstream, ready for execution on Red Pitaya, as well as a Python script to configure the Red Pitaya and make plots. [Here](#)

Spectrometer Basics

When designing a spectrometer there are a few main parameters of note:

- **Bandwidth:** The width of your frequency spectrum, in Hz. This depends on the sampling rate; for complex sampled data this is equivalent to:

$$BW = \text{sampling rate} = \frac{1}{\text{sampling period}}$$

In contrast, for real or Nyquist sampled data the rate is half this:

$$BW = \frac{\text{sampling rate}}{2} = \frac{1}{2 \times \text{sampling period}}$$

as two samples are required to reconstruct a given waveform .

- **Frequency resolution:** The frequency resolution of a spectrometer, Δf , is given by

$$\Delta f = \frac{BW}{\text{no. channels}},$$

and is the width of each frequency bin. Correspondingly, Δf is a measure of how precise you can measure a frequency.

- **Time resolution:** Time resolution is simply the spectral dump rate of your instrument. We generally accumulate multiple spectra to average out noise; the more accumulations we do, the lower the time resolution. For looking at short timescale events, such as pulsar bursts, higher time resolution is necessary; conversely, if we want to look at a weak HI signal, a long accumulation time is required, so time resolution is less important.

Simulink / CASPER Toolflow

Simulink Design Overview

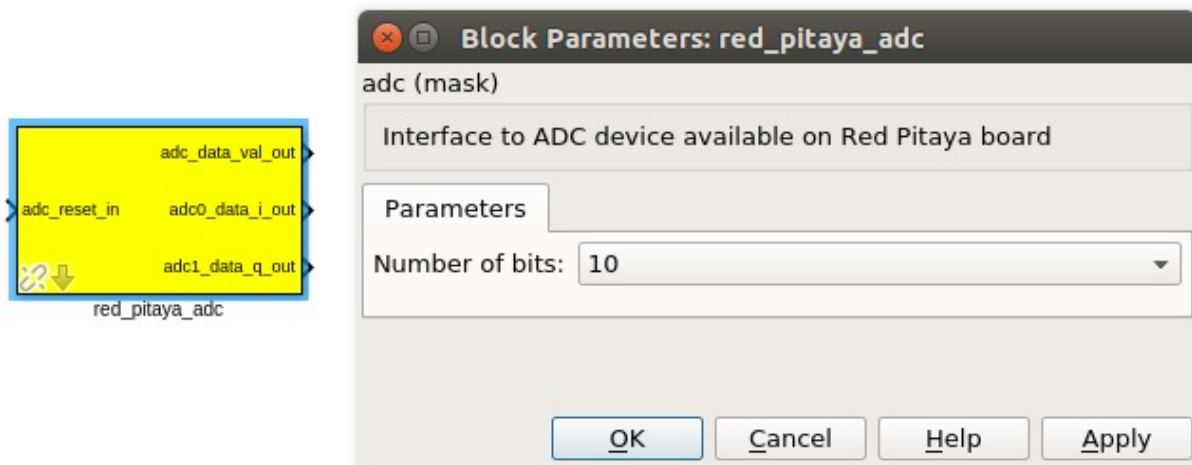
If you're reading this, then you've already managed to find all the tutorial files. By now, I presume you can open the model file and have a vague idea of what's happening. The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some "blockumentation" in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- The all important Xilinx token is placed to allow System Generator to be called to compile the design.

- In the MSSGE block, the hardware type is set to “RED_PITAYA_10:xc7z010” and clock rate is specified as 125MHz.
- In this tutorial, we use the 10-bit Red Pitaya. Make sure that the platform and ADC yellow blocks are configured for 10 bits.
- The input signal is digitised by the ADC. The ADC runs at 125MHz, which gives a 62.5MHz nyquist sampled spectrum. The RF inputs have a slightly narrow bandpass of 50MHz. The output range is a signed number in the range 0 to 1023 (ie 0 bits after the decimal point). This is expressed as fix_10_0. This is a little non-standard. You’ll notice that tutorials for other platforms have an ADC output between -1 and +1 and a binary point to match.
- The Xilinx FFT is configured for 256 channels.
- You may notice delay blocks dotted in places design. It’s common practice to add these into the design as it makes it easier to fit the design into the logic of the FPGA. It consumes more resources, but eases signal timing-induced placement restrictions.
- The real and imaginary (sine and cosine value) components of the FFT are plugged into power blocks, to convert from complex values to real power values by squaring.
- The requantized signals then enter the vector accumulators, which are simple_bram_vacc 32 bit vector accumulators. Accumulation length is controlled by the acc_cntrl block.
- The accumulated signal for each channel is then fed into a 32-bit snap block, accum0_snap and accum1_snap.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

adc



The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter. In Simulink, the ADC is represented by a yellow block.

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 10 bit binary point numbers in the range of -1 to 1 and are then output by the ADC. This is achieved through the use of two’s-compliment representation with the binary point placed after the seven least significant bits. This means we can represent numbers from -512/512 through to 511/512 including the number 0. Simulink represents such numbers with a fix_10_0 moniker.

ADCs often internally bias themselves to halfway between 0 and -1. This means that you’d typically see the output of an ADC toggling between zero and -1 when there’s no input. It also means that unless otherwise calibrated, an ADC will have a negative DC offset.

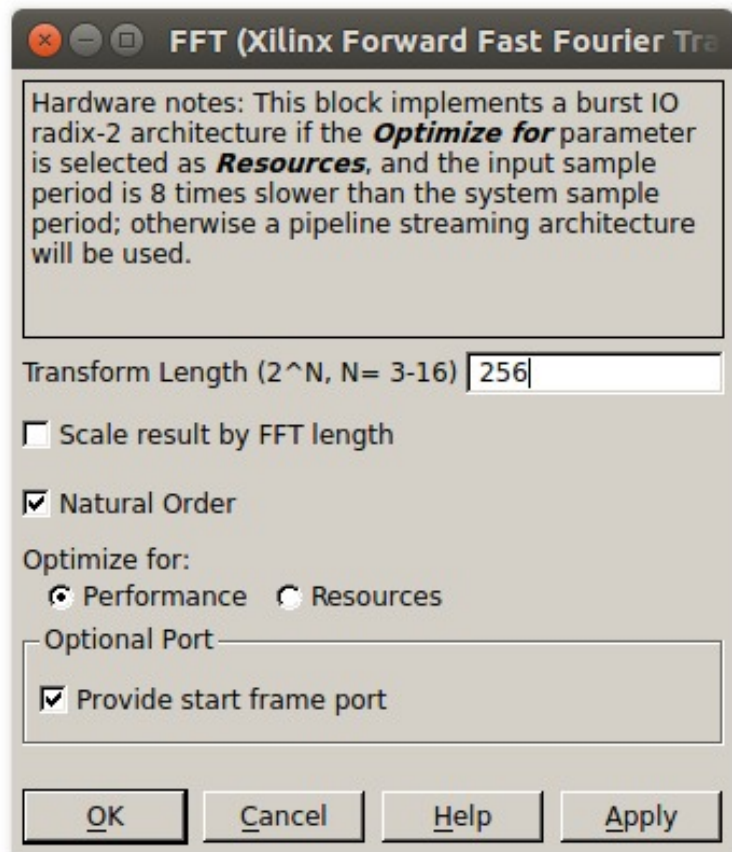
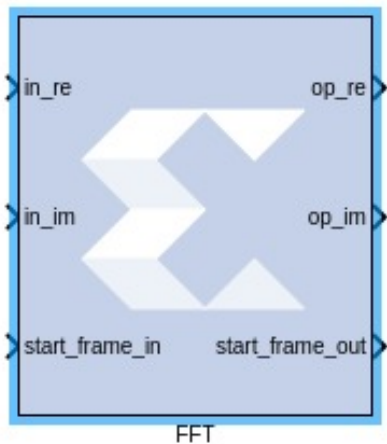
The ADC has to be clocked to four times that of the FPGA clock. In this design the ADC is clocked to 125MHz, generated from the Red Pitaya's system clock. This gives us a bandwidth of 62.5MHz, as Nyquist sampling requires two samples (or more) each second.

INPUTS

OUTPUTS

The ADC outputs are a data valid flag and two signals: i and q, which correspond to the coaxial inputs of the Red Pitaya (inputs 0 and 1).

Xilinx FFT

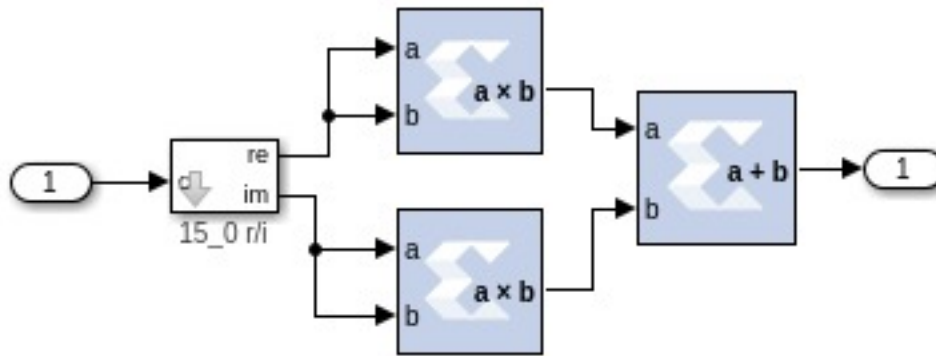


This Red Pitaya tutorial utilizes the Xilinx FFT block – one for each channel of the ADC. adc_di and adc_dq route to the real inputs on each FFT. Each block is configured for a 256 point transform.

INPUTS

OUTPUTS

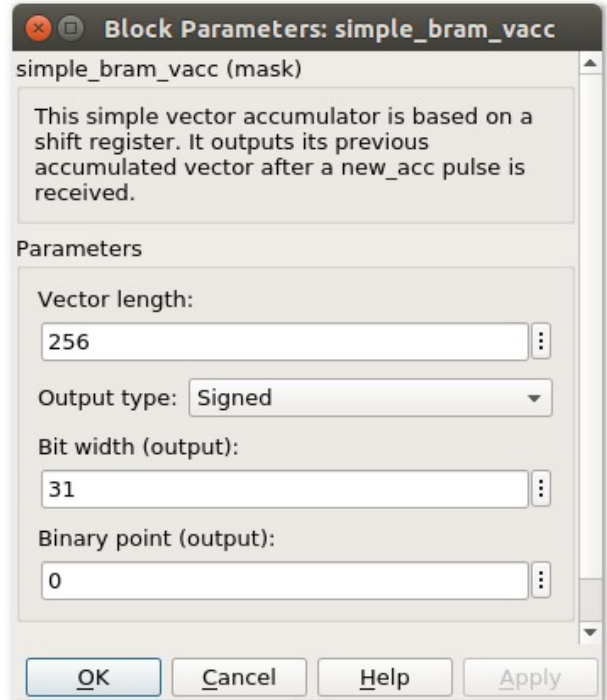
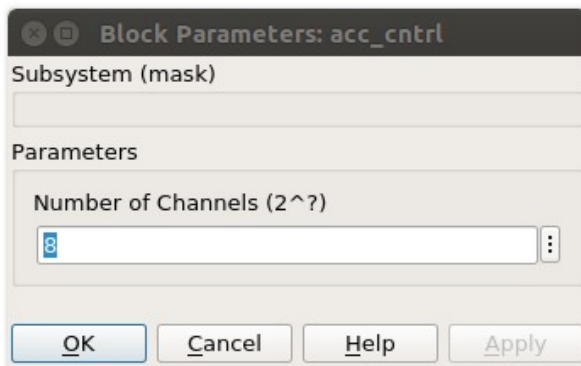
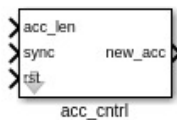
power



The power block computes the power of a complex number. Underneath the subsystem blocks, you see that the power block will compute the power of its input by taking the sum of the squares of its real and imaginary components. The output of the block is 31.0 bits.

INPUTS/OUTPUTS

simple_bram_vacc

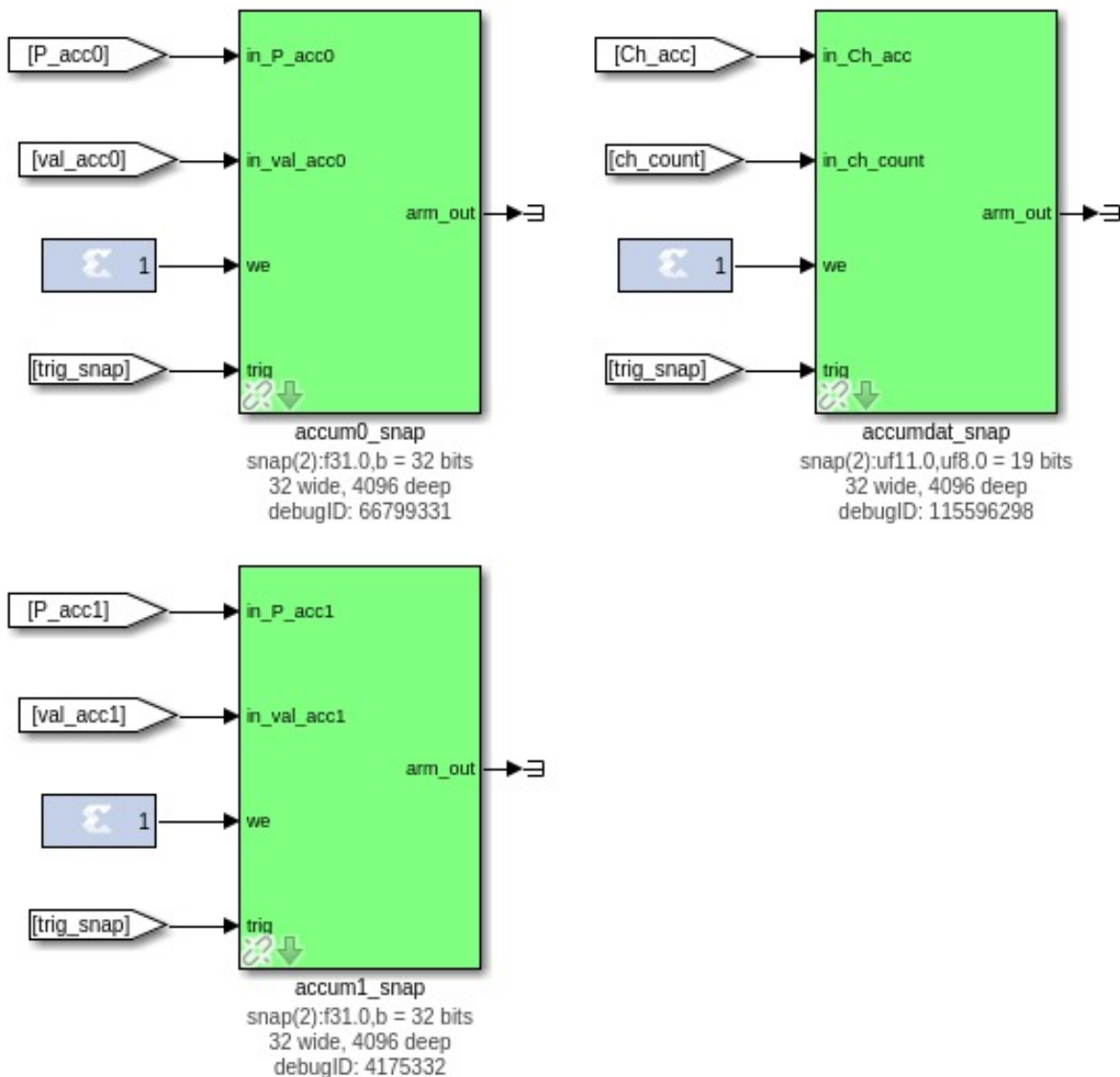


The `simple_bram_vacc` block is used in this design for vector accumulation. If you wanted a really long accumulation (say a few hours), you'd have to use a block such as the `qdr_vacc` or `dram_vacc`. As the name suggests, the `simple_bram_vacc` is simpler so it is fine for this demo spectrometer. The FFT block outputs 256 frequency bins in total. We have two of these `bram_vacc`'s in the design, one for each port on the Red Pitaya. The vector length is set to 256 on both.

PARAMETERS

INPUTS/OUTPUTS

Snap Blocks



The final blocks, `accum0_snap` and `accum1_snap`, capture the data coming from the accumulators, which we will read out the values of using the `tut_spec.py` script.

PARAMETERS

INPUTS/OUTPUTS

Software Registers

There are a few **control registers**, led blinkers, and **snap** block dotted around the design too:

- **reg_cntrl**: Counter reset control. Pulse this high to reset the ADC and ADC counter.
- **acc_len**: Sets the accumulation length. Have a look in `tut_spec.py` for usage.
- **sync_reg**: Synchronizes the FFTs. Pulse this high to start/restart the FFT output.
- **sync_cnt**: Logs the number of syncs into the FFTs.
- **fft_sync_inc**: Logs the number of syncs leaving each FFT.
- **acc_cnt**: Accumulation counter. Keeps track of how many accumulations have been done.
- **gpio_led**: LED flashes while the bitcode is running.

If you've made it to here, congratulations, go and get yourself a cup of tea and a biscuit, then come back for part two, which explains the second part of the tutorial – actually getting the spectrometer running, and having a look at some spectra.

Configuration and Control

Hardware Configuration

The tutorial comes with a pre-compiled fpg file, which is generated from the model you just went through (`tut_spec.fpg`)

Next, you need to set up your Red Pitaya. Power it on, making sure that:

- By default, the Red Pitaya takes RF inputs between -1 and +1 V though it can be configured for higher voltages.
- Connect inputs 0 and 1 to sine wave generators. The frequencies should be between 0 and 50 MHz.

The `tut_spec.py` spectrometer script

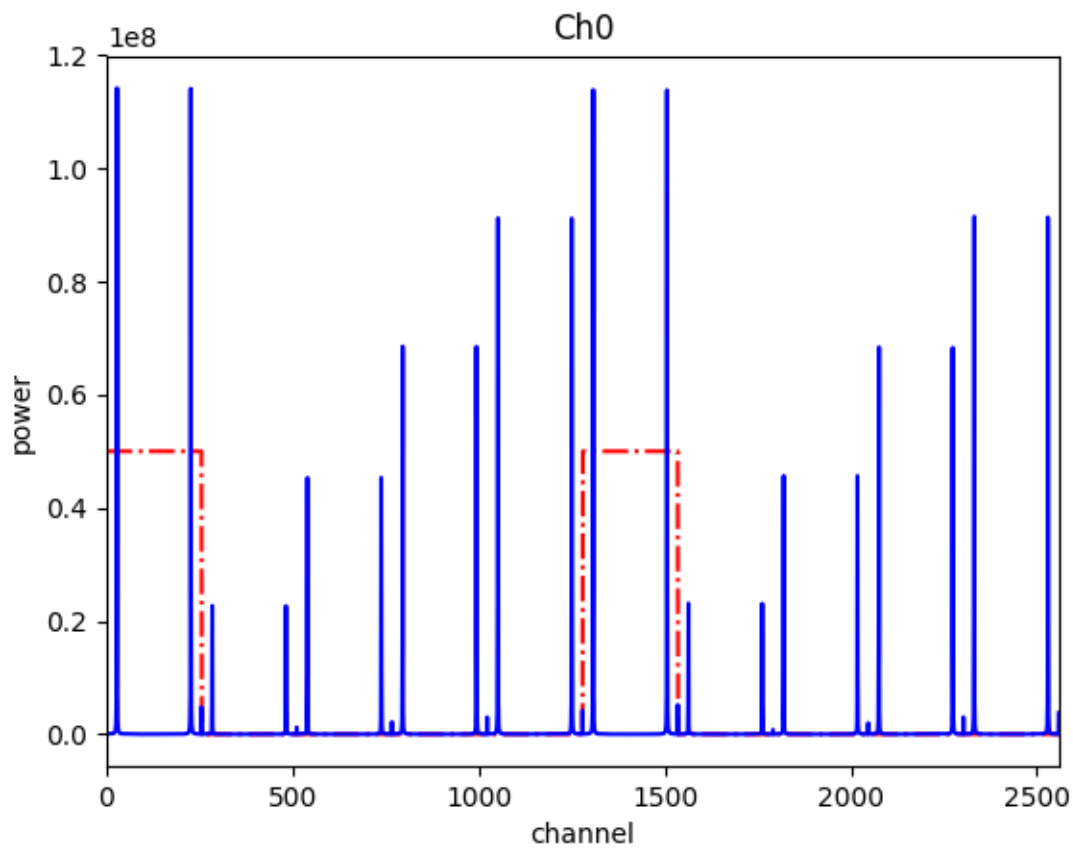
This short script does the following:

- calls the Red-Pitaya
- uploads an fpg file
- activates the reset registers
- pulls data from the snap blocks
- plots the spectra.

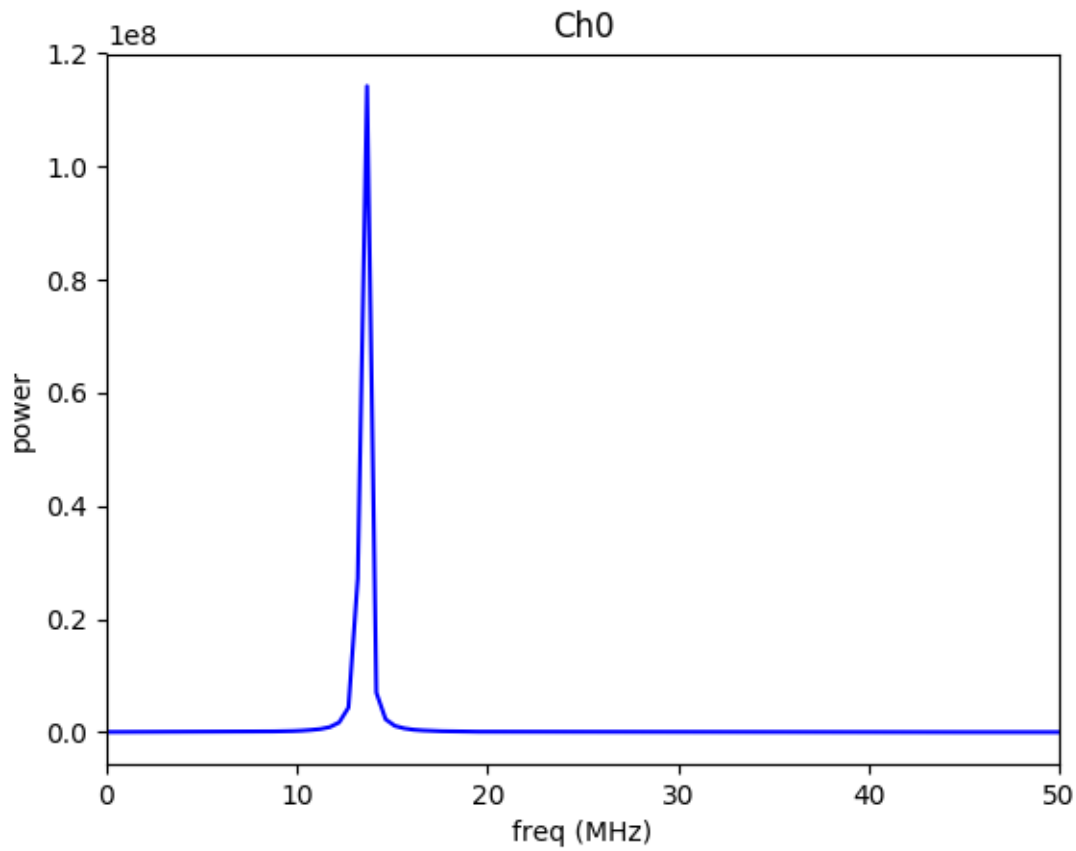
Browse to where the `tut_spec.py` file is in a terminal and at the prompt type

```
python tut_spec.py -f <fpgfile name> -r <Red-Pitaya IP or hostname> -a <accumulation_
↪length>
```

replacing with the IP address of your Red-Pitaya and with your fpg path and filename. You should see a spectrum like this:



In the plot, the blue solid line represents several spectra concatenated one after the next. You see that after each accumulation – in this case 5 – the amplitude of the tones build. After the fifth spectrum, then the accumulation resets and repeats. The data valid flag, shown by a red-dashed line scaled to the range of the plot, shows the indices of the last accumulation. We plot the valid accumulation in frequency space, you see the tone at 14 MHz. Inspect the `tut_spec.py` script to see how this is done.



Conclusion

If you have followed this tutorial faithfully, you should now know:

- What a spectrometer is and what the important parameters for astronomy are.
- Which CASPER blocks you might want to use to make a spectrometer, and how to connect them up in Simulink.
- How to connect to and control a Red Pitaya spectrometer using python scripting.

1.2 ISE

ROACH1/2

1. *Introduction Tutorial*
2. *10GbE Tutorial*
3. *Spectrometer Tutorial*
4. *Correlator Tutorial*

1.2.1 Tutorial 1: Introduction to Simulink

In this tutorial, you will create a simple Simulink design using both standard Xilinx system generator blockset, as well as library blocks specific to CASPER boards (so-called “Yellow Blocks”). At the end of this tutorial, you will know how to generate an fpg file, program it to a CASPER FPGA board, and interact with your running hardware design using `casperfpga` via a Python Interface.

Creating Your Design

Create a New Model

Start MATLAB via executing the `startsg` command, as described [here](#). This ensures that necessary Xilinx and CASPER libraries are loaded into development environment by Simulink. When MATLAB starts up, open Simulink by typing `simulink` on the MATLAB command line. Start a new model, and save it with an appropriate name. **With Simulink, it is very wise to save early and often.**

There are some Matlab limitations you should be aware-of right from the start:

- **Do not use spaces in your filenames** or anywhere in the file path as it will break the toolflow.
- **Do not use capital letters in your filenames** or anywhere in the file path as it will break the toolflow.
- **Beware block paths that exceed 64 characters.** This refers to not only the file path, but also the path to any block within your design.
 - For example, if you save a model file with a name `~/some_really_long_filename.slx`, and have a block called in a submodule the longest block path would be: `some_really_long_filename_submodule_block`.
 - If you use lots of subsystems, this can cause problems.

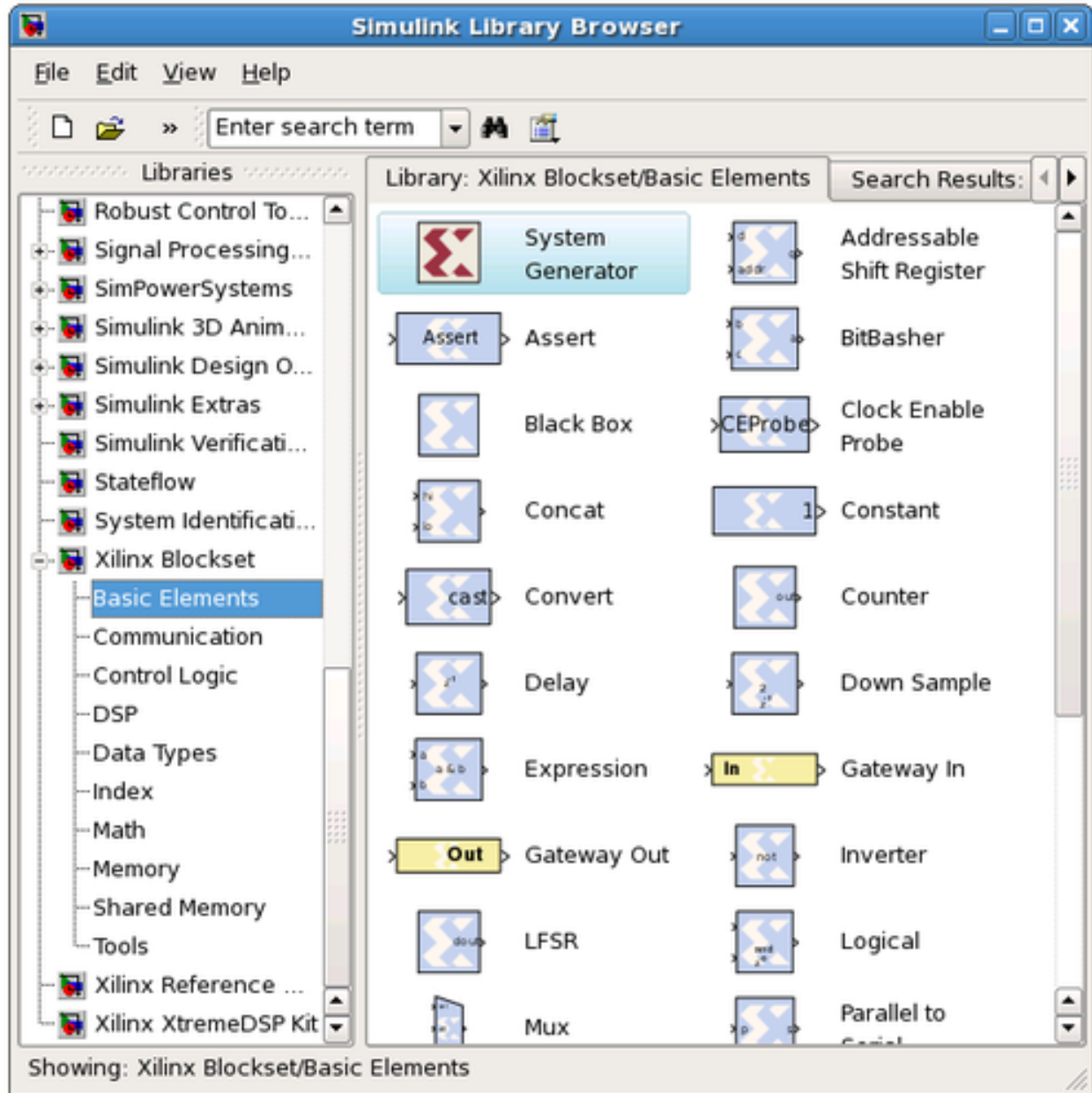
Library organization

There are three libraries which you will use when you design firmware in Simulink.

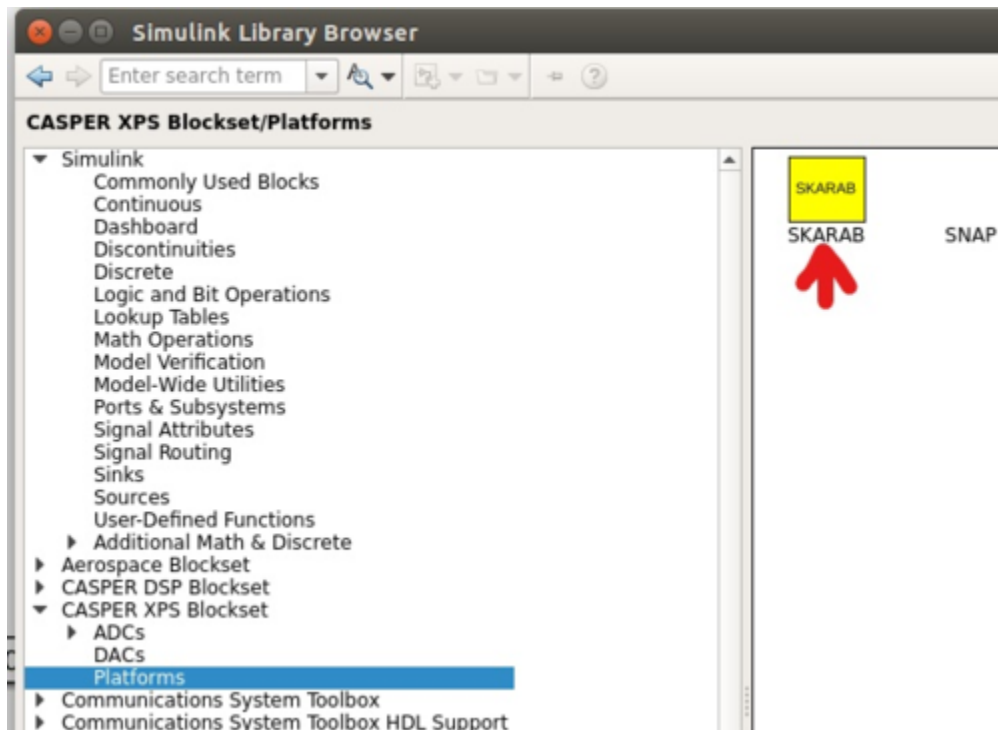
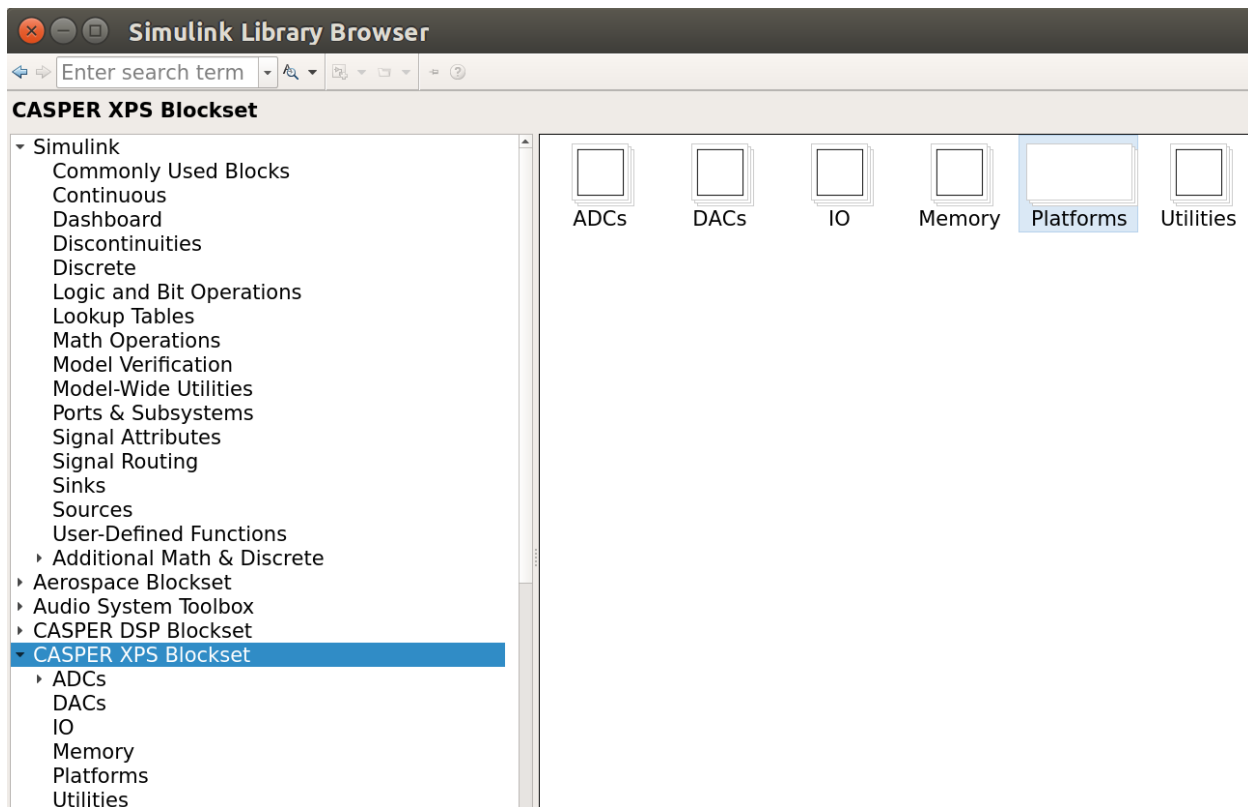
1. The *CASPER XPS Library* contains “Yellow Blocks” – these are blocks which encapsulate interfaces to hardware (ADCs, Memory chips, CPUs, Ethernet ports, etc.)
2. The *CASPER DSP Library* contains (mostly green) blocks which implement DSP functions, like filters, FFTs, etc.
3. The *Xilinx Library* contains blue blocks which provide low-level functionality such as multiplexing, delaying, adding, etc. The Xilinx library also contains the super-special System Generator block, which contains information about the type of FPGA you are targeting.

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 a platform block representing the system you are compiling for. These can be found in the CASPER XPS System Blockset library. For ROACH2, you need an XPS_core_config block.



Double click on the platform block that you just added. The Hardware Platform parameter should match the platform you are compiling for. For ROACH2 you have to set this from a drop-down list, for newer platforms it should automatically be set to the correct board type. Once you have selected a board, you need to choose where it will get its clock. In designs including ADCs you probably want the FPGA clock to be derived from the sampling clock, but for this simple design (which doesn't include an ADC) you should use the platform's on-board clock. To do this, set the

User IP Clock Source to sys_clk. The sys_clk rate is 100 MHz, so you should set this for *User IP Clock Rate* in the block.

The configuration yellow block knows what FPGA corresponds to which platform, and so it will automatically configure the System Generator block which you previously added.

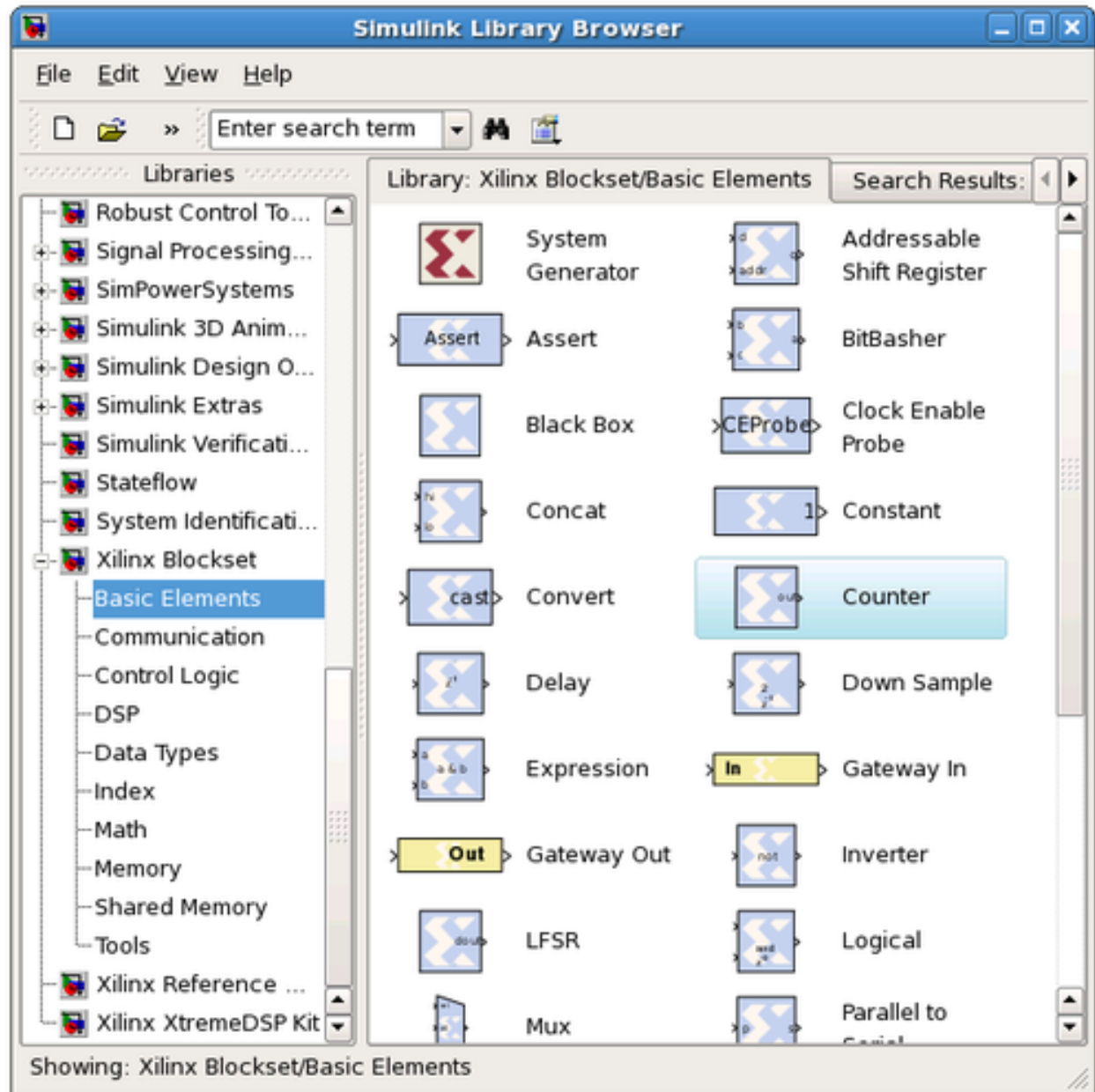
The System Generator and XPS Config blocks are required by all CASPER designs

Flashing LED


To demonstrate the basic use of hardware interfaces, we will make an LED flash. With the FPGA running at ~100MHz (or greater), the most significant bit (MSB) of a 27 bit counter will toggle approximately every 0.67 seconds. We can output this bit to an LED on your board. Most (all?) CASPER platforms have at least four LEDs, with the exact configuration depending on the board. We will make a small circuit connecting the top bit of a 27 bit counter to one of these LEDs. When compiled this will make the LED flash with a 50% duty cycle approximately once a second.

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. This means it will count from 0 to $2^{27} - 1$, and will then wrap back to zero and continue.

 **Counter (Xilinx Counter)** [-] [] [X]

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Advanced** **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value

Step

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits

Binary point

Optional Ports

☐ Provide load port

☐ Provide synchronous reset port

☐ Provide enable port

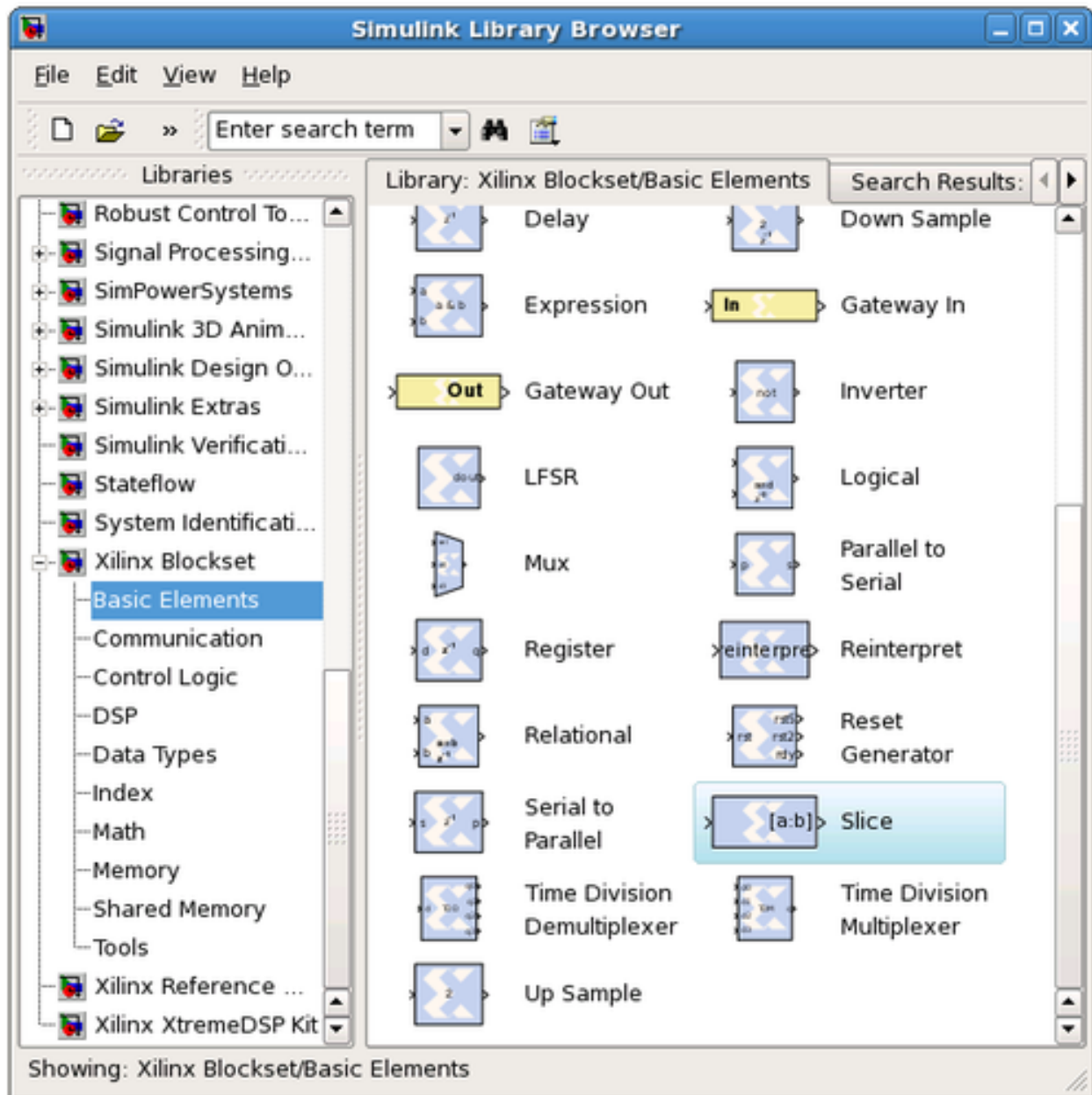
Explicit Sample Period

Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period

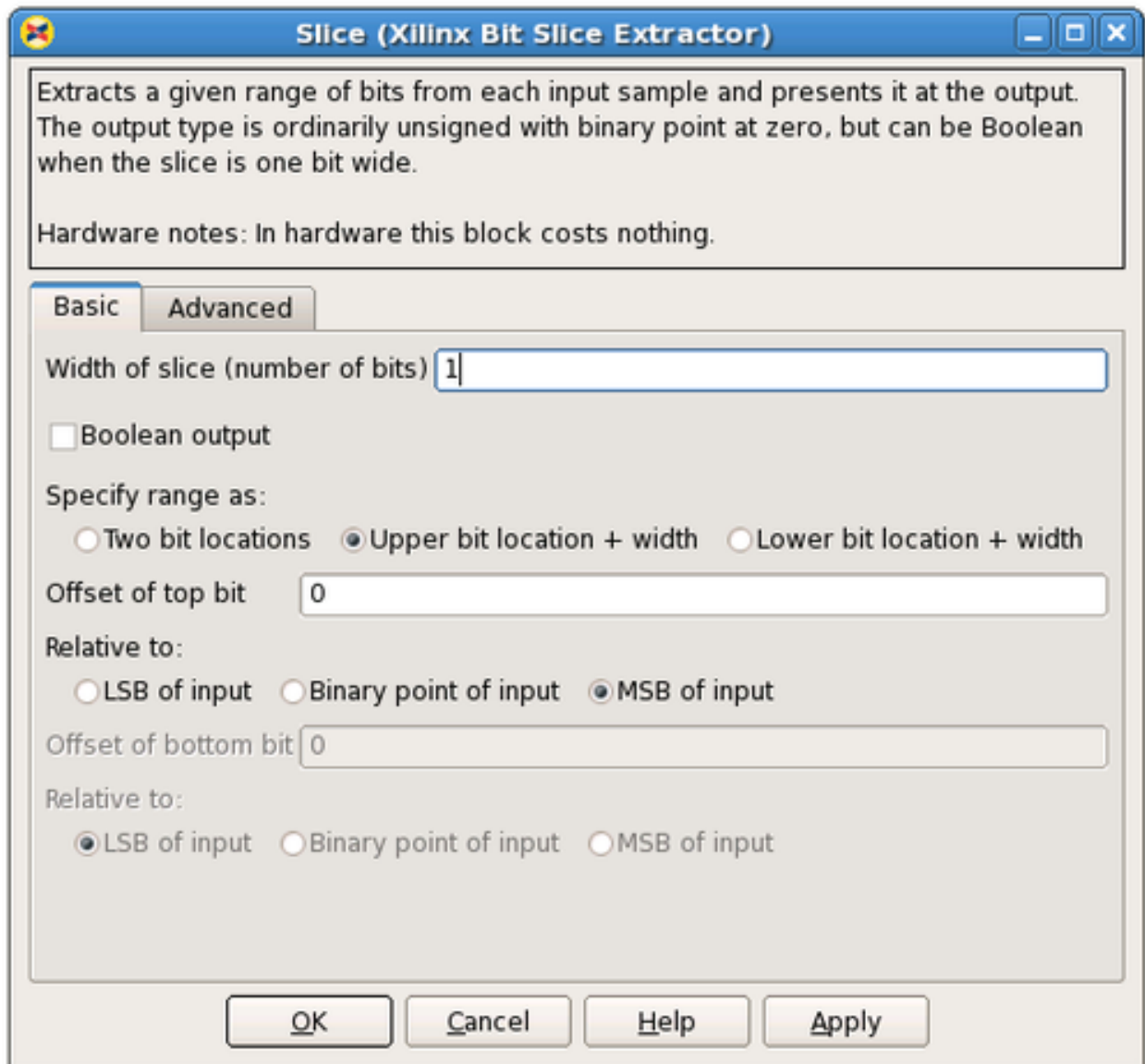
Add a slice block to select the MSB

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



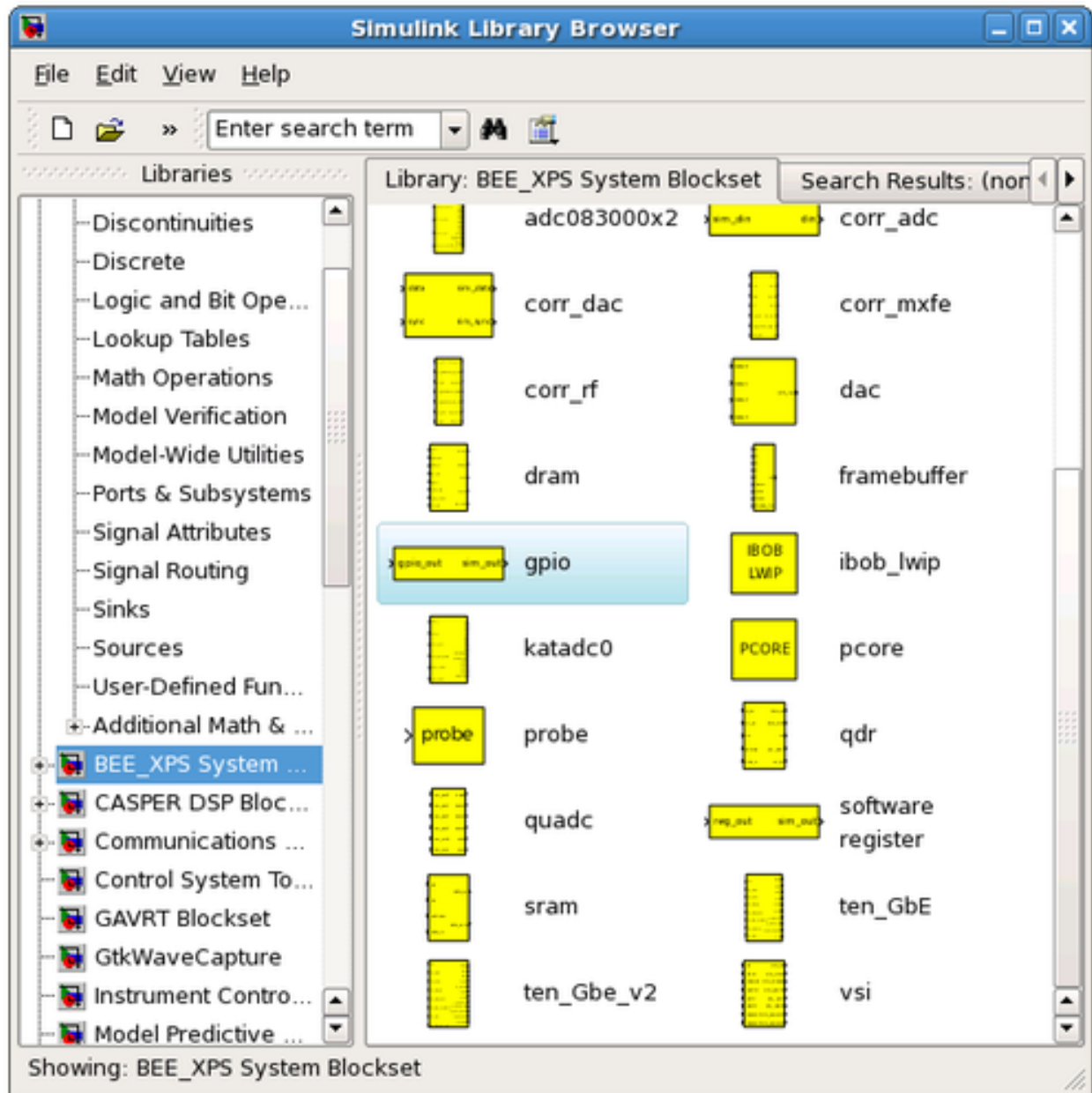
Double-click on the newly-added slice block. There are multiple ways to select which bit(s) you want. In this case, it is simplest to index from the upper end and select the first bit. If you wanted the **least significant bit (LSB)**, you can also index from that position. You can either select the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

Set it for 1 bit wide with offset from top bit at zero. As you might guess, this will take the 27-bit input signal, and output just the top bit.

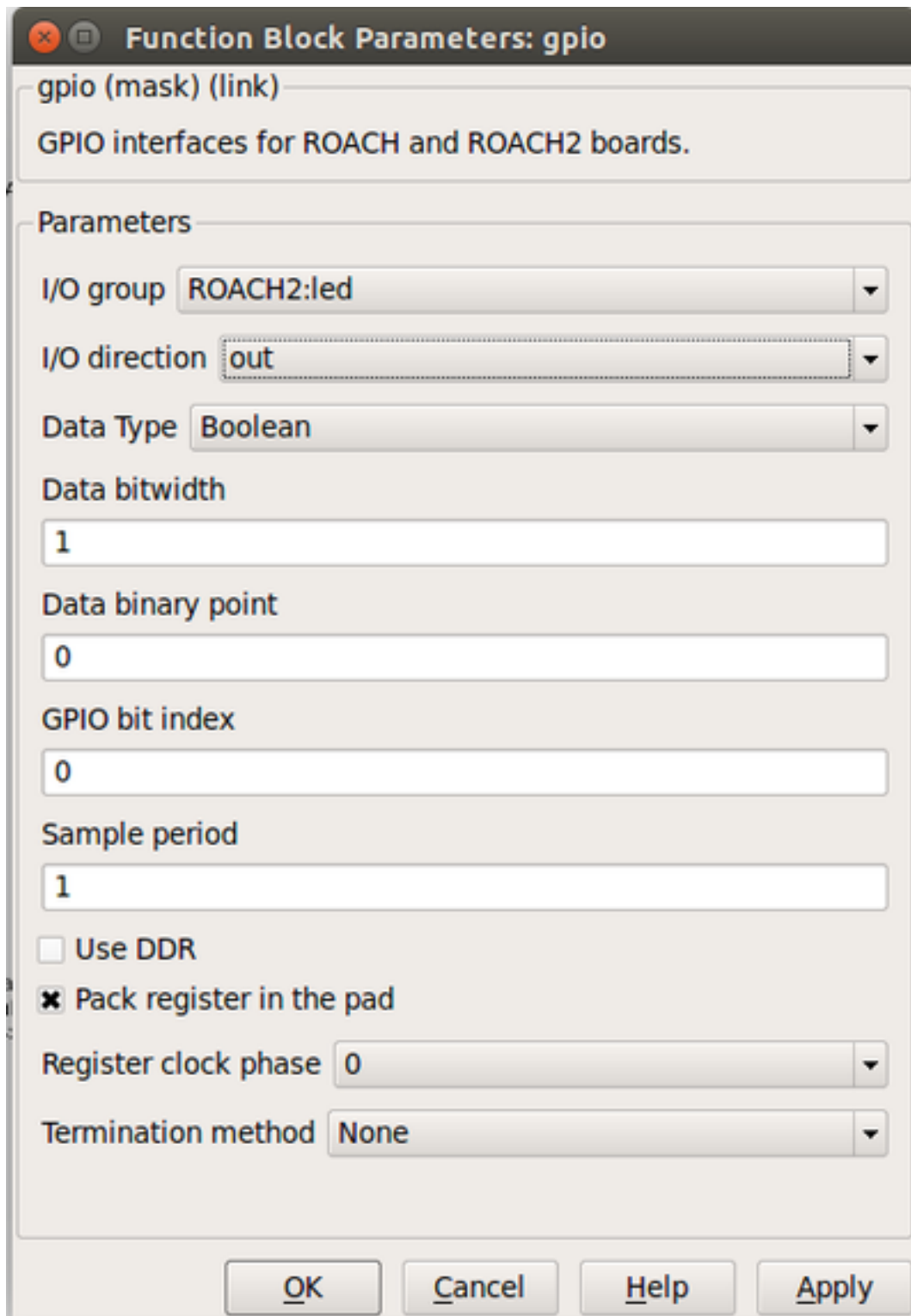


Add a GPIO block

From: CASPER XPS library -> gpio.



In order to send the 1-bit signal you have sliced off to an LED, you need to connect it to the right FPGA output pin. To do this you can use a GPIO (general-purpose input/output) block from the XPS library, this allows you to route a signal from Simulink to a selection of FPGA pins, which are addressed with user-friendly names. Set it to use ROACH2's LED bank as output. Once you've chosen the LED bank, you need to pick *which* LED you want to output to. Set the GPIO bit index to 0 (the first LED) and the data type to Boolean with bitwidth 1. This means your Simulink input is a 1 bit Boolean, and the output is LED0.



The image shows a dialog box titled "Function Block Parameters: gpio". It contains a description "gpio (mask) (link)" and "GPIO interfaces for ROACH and ROACH2 boards." Below this is a "Parameters" section with several fields: "I/O group" is a dropdown menu set to "ROACH2:led"; "I/O direction" is a dropdown menu set to "out"; "Data Type" is a dropdown menu set to "Boolean"; "Data bitwidth" is a text input field containing "1"; "Data binary point" is a text input field containing "0"; "GPIO bit index" is a text input field containing "0"; "Sample period" is a text input field containing "1"; there are two checkboxes, "Use DDR" (unchecked) and "Pack register in the pad" (checked); "Register clock phase" is a dropdown menu set to "0"; and "Termination method" is a dropdown menu set to "None". At the bottom are four buttons: "OK", "Cancel", "Help", and "Apply".

Function Block Parameters: gpio

gpio (mask) (link)

GPIO interfaces for ROACH and ROACH2 boards.

Parameters

I/O group ROACH2:led

I/O direction out

Data Type Boolean

Data bitwidth

1

Data binary point

0

GPIO bit index

0

Sample period

1

☐ Use DDR

☒ Pack register in the pad

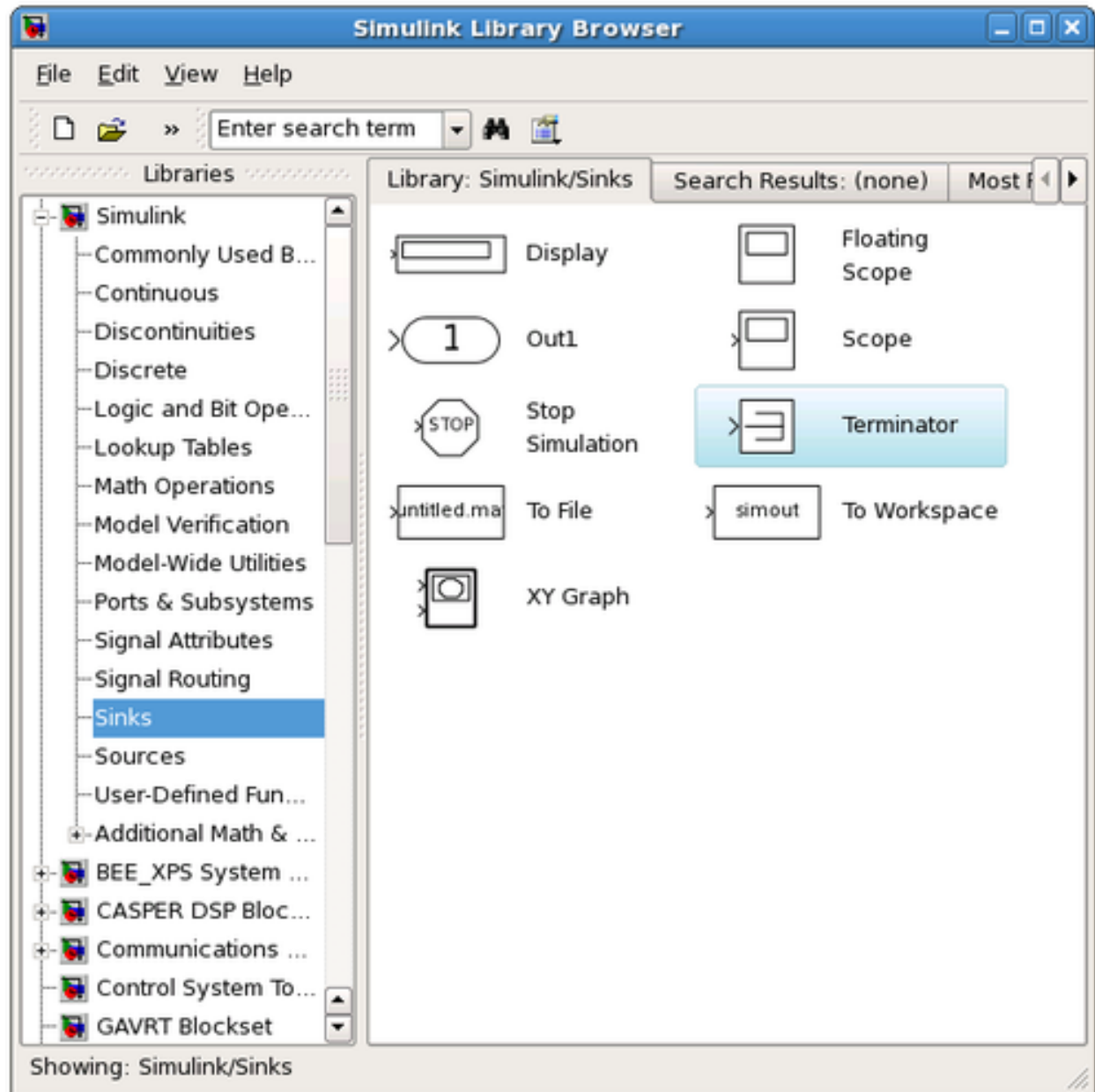
Register clock phase 0

Termination method None

OK Cancel Help Apply

Add a terminator

To prevent warnings (from MATLAB & Simulink) about unconnected outputs, terminate all unused outputs using a *Terminator*:



From: Simulink -> Sinks -> Terminator

You can also use the Matlab function `xlAddTerms`, run in the MATLAB prompt, to automatically terminate your unused outputs.

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.

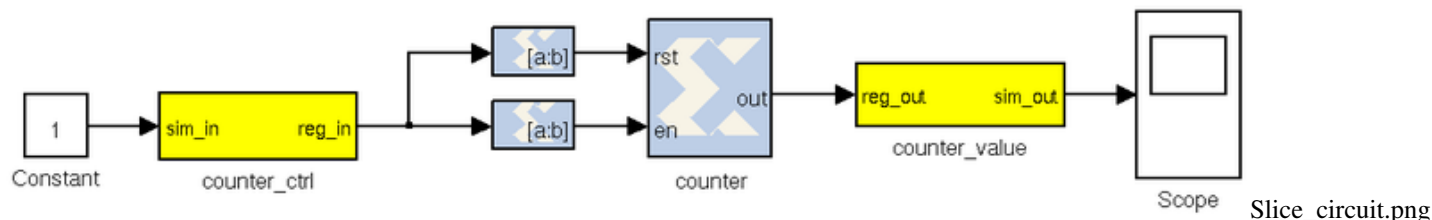
To connect the blocks simply click and drag from the 'output arrow' on one block and drag it to the 'input arrow' of another block. Connect the blocks together: Counter -> Slice -> gpio as showing in digram below.



Remember to save your design often.

Software control

To demonstrate the use of software registers to control the FPGA from a computer, we will add registers so that the counter in our design can be started, stopped, and reset from software. We will also add a register so that we can monitor the counter's current value too. By the end of this section you will create a system that looks like this:

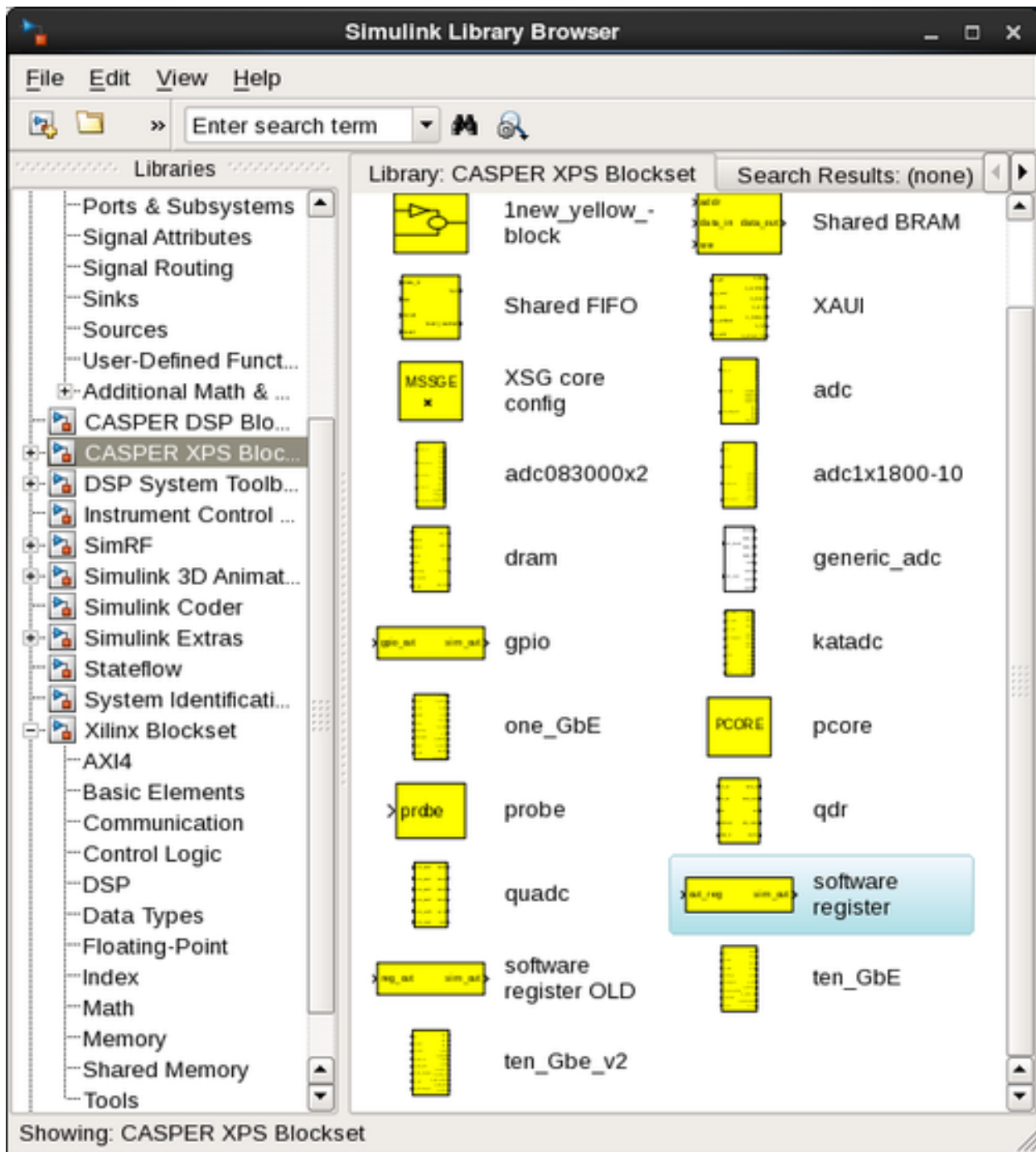


Add the software registers

We need two software registers:



1. To control the counter, and
2. To read its current value.

From the CASPER XPS System Blockset library, drag two software registers into your design.



SW_reg_select2.p

Set the I/O direction to *From Processor* on the first one (counter control) to enable a value to be set from software and sent to your FPGA design. Set it to *To Processor* on the second one (counter value) to enable a value to be sent from the FPGA to software. Set both registers to a bitwidth of 32 bits.

 **Block Parameters: counter_ctrl**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction From Processor

I/O delay

0

Initial Value

0

Sample period

1

Bitfield names [msb...lsb]

reg

Bitfield widths

32

Bitfield binary pts

0



Bitfield types, ufix=0, fix=1, bool=2

0

☒ Provide sim input/output?

1.2. ISE

☐ Print format string?

 **Block Parameters: counter_value**

swreg (mask)

A 32-bit software-accessible register. Can be divided into bitfields of varying widths and types using the fields in the block mask.

Setup

I/O direction To Processor

I/O delay

Initial Value

Sample period

Bitfield names [msb...lsb]

Bitfield widths

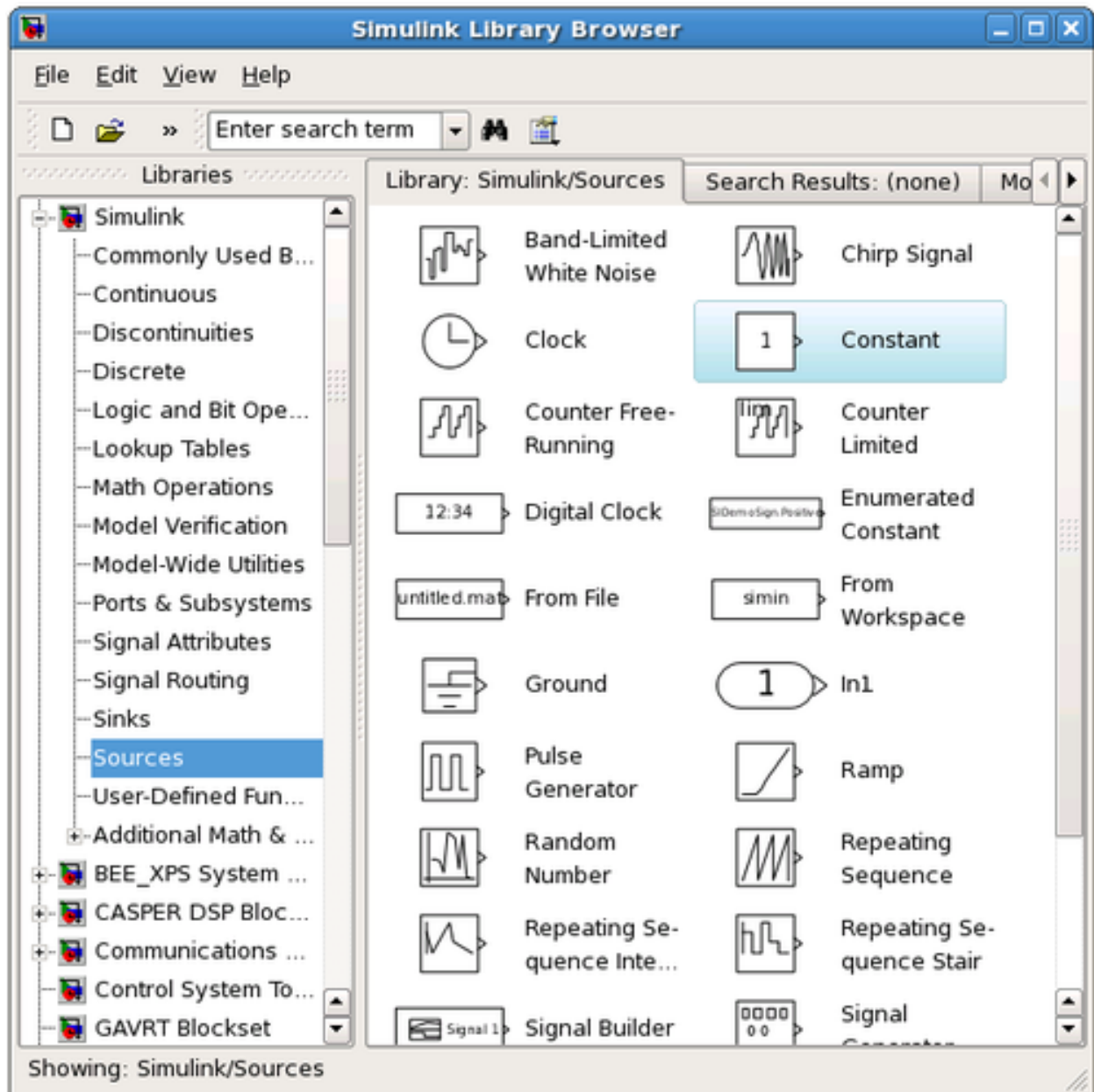
Bitfield binary pts

Bitfield types, ufix=0, fix=1, bool=2

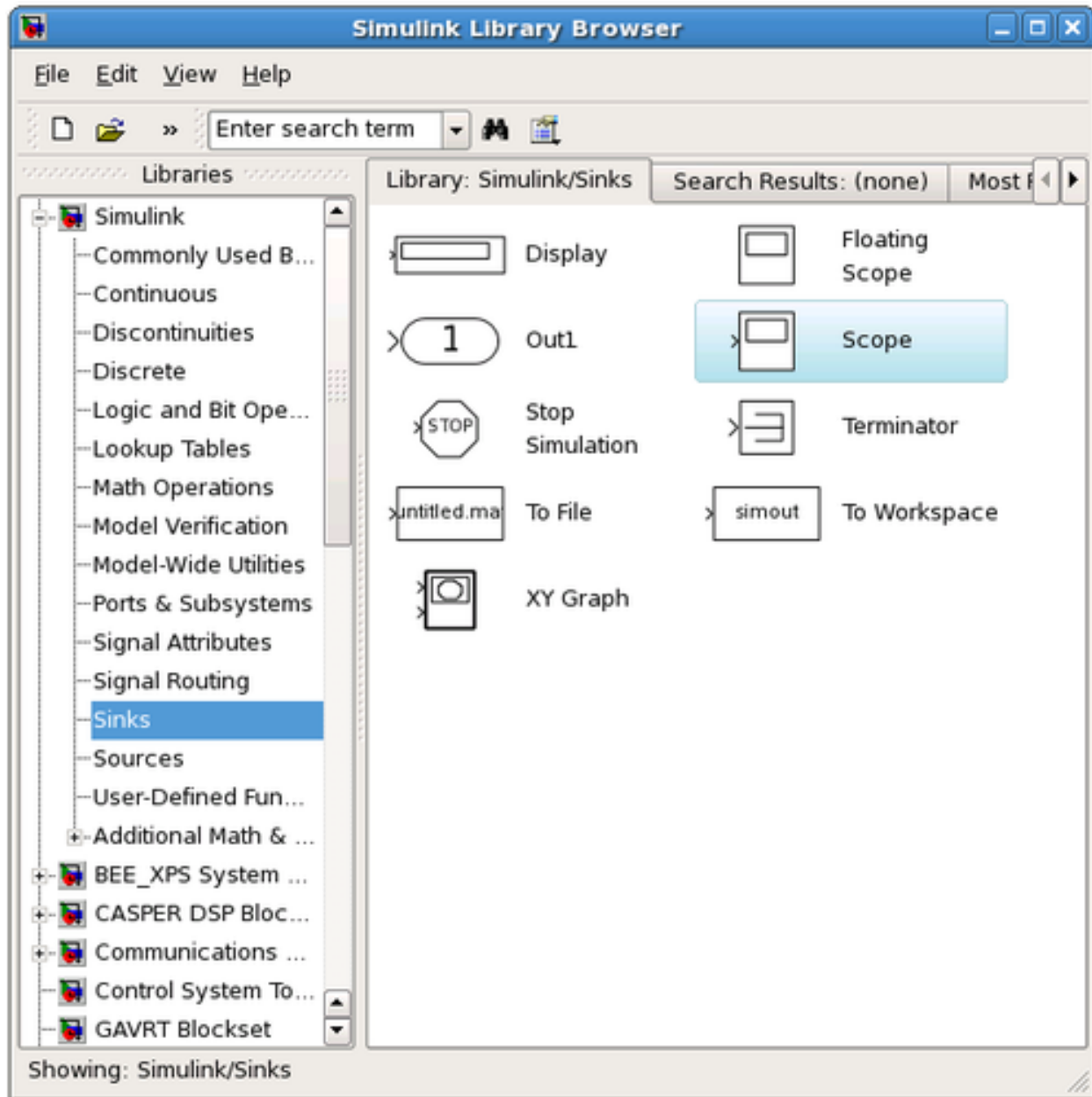
Rename the registers to something sensible, the names you give them here are the names you will use to access them from software. Do not use spaces, slashes and other special characters in these. Perhaps *counter_ctrl* and *counter_value* to represent the control and output registers respectively.

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

For now, set the *sim_reg* port to constant one using a Simulink-type constant. This can be found in *Simulink -> Sources*, and will enable the counter during simulations.



During simulation we can monitor the counter's value using a scope (*Simulink -> Sinks*):



Here is a good point to note that all blocks from the *Simulink* library are usually white in colour, and will not be compiled into hardware. i.e. They are present for simulation only. Xilinx blocks are usually blue in colour with the Xilinx logo, and will be compiled to hardware.

You need to use *gateway* blocks whenever connecting a Simulink-provided block (like a scope or sine-wave generator) to and from a Xilinx block. This will sample and quantize the Simulink signals so that they are compatible with the Xilinx world. Some blocks (like the software register) provide a gateway internally, so you can feed the input of a software register with a Xilinx signal, and monitor its output with a Simulink scope. However, in general, you must manually insert these gateways where appropriate. Simulink will issue warnings for any direct connections between the Simulink and Xilinx domains.

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:

counter (Xilinx Counter)

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Basic **Advanced** **Implementation**

Counter type:
☒ Free running ☐ Count limited

Count to value:

Count direction:
☒ Up ☐ Down ☐ Up/Down

Initial value:

Step:

Output Precision

Output type:
☐ Signed (2's comp) ☒ Unsigned

Number of bits:

Binary point:

Optional Ports

☐ Provide load port

☒ Provide synchronous reset port

☒ Provide enable port

Explicit Sample Period

Sample period source:
☒ Explicit ☐ Inferred from inputs

Explicit period:

OK **Cancel** **Help** **Apply**

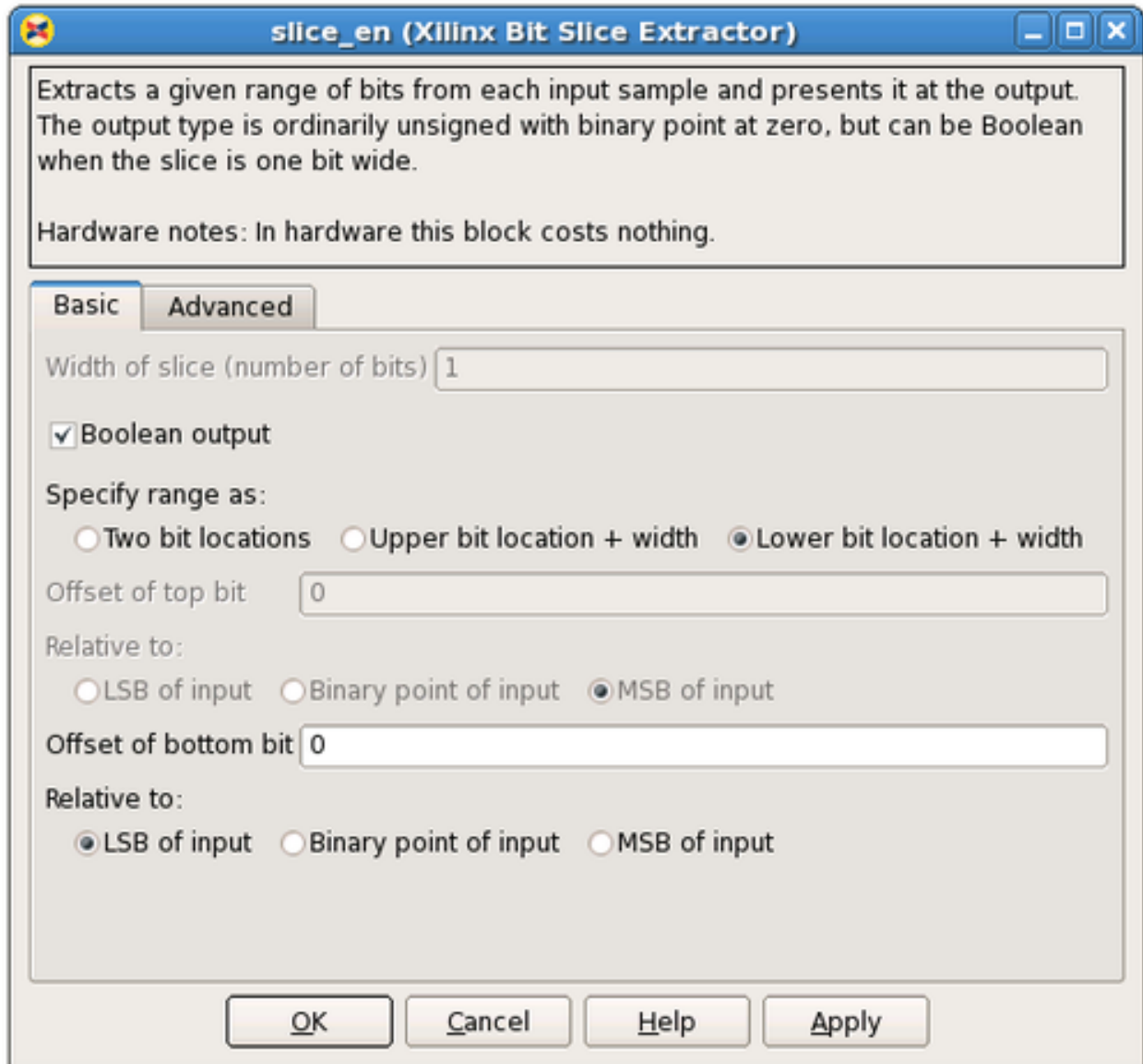
Add the slice blocks

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.

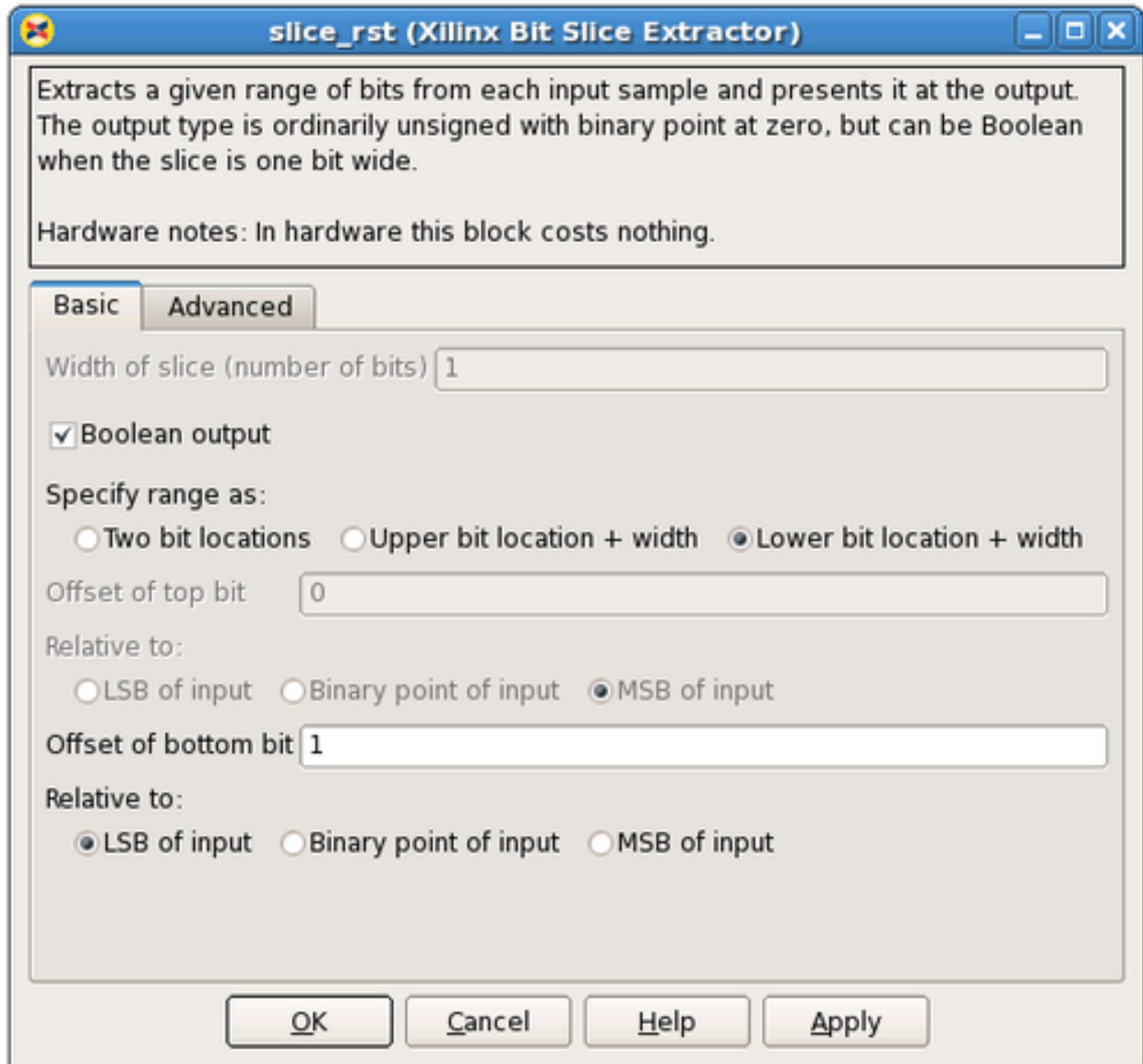
So we'll use a single register and slice out one bit for enabling the counter, and another bit for resetting it. Either copy your existing slice block (copy-paste it or hold ctrl while dragging/dropping it) or add two more from the library.

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:



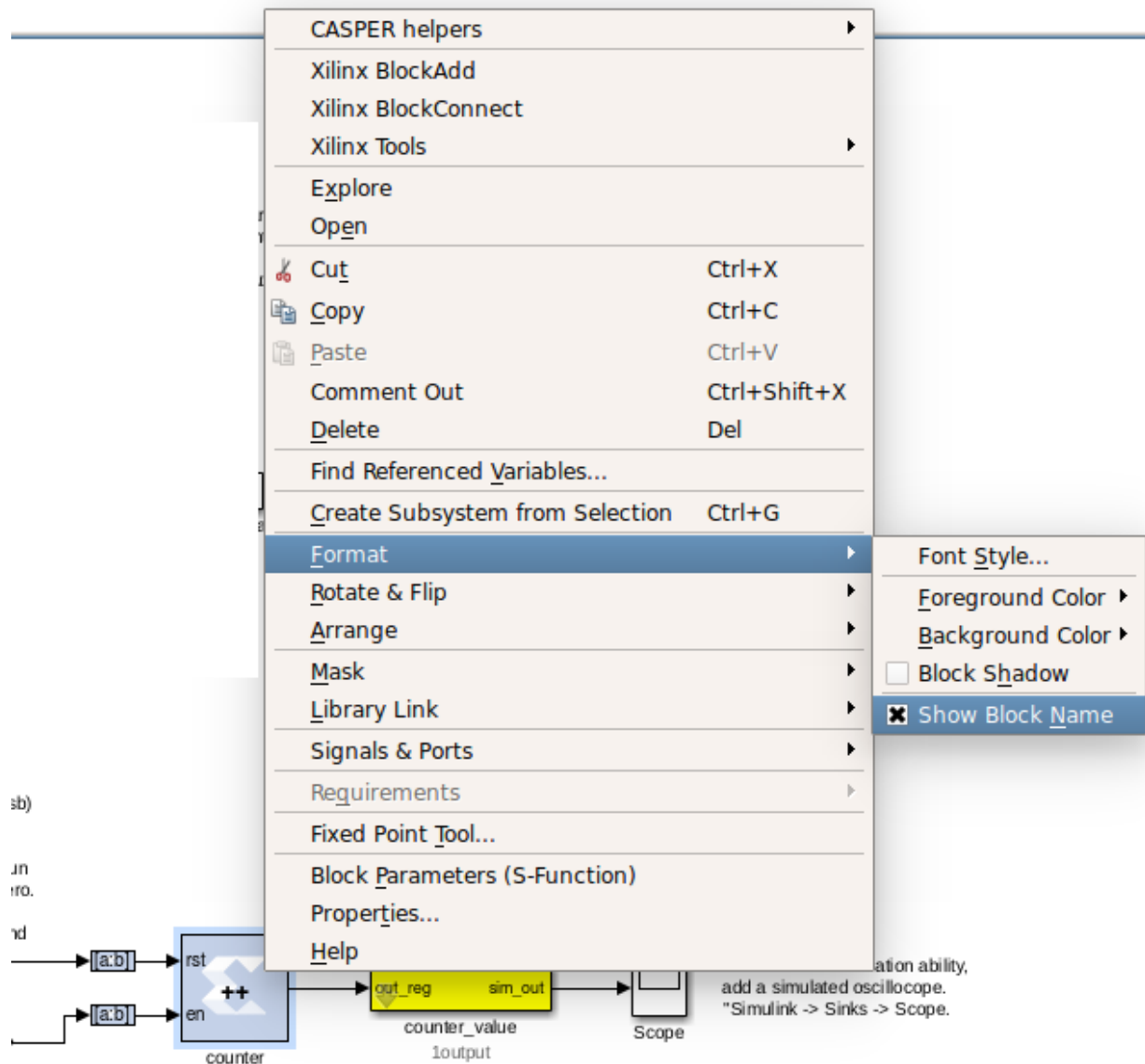
Slice for reset:



Connect 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.

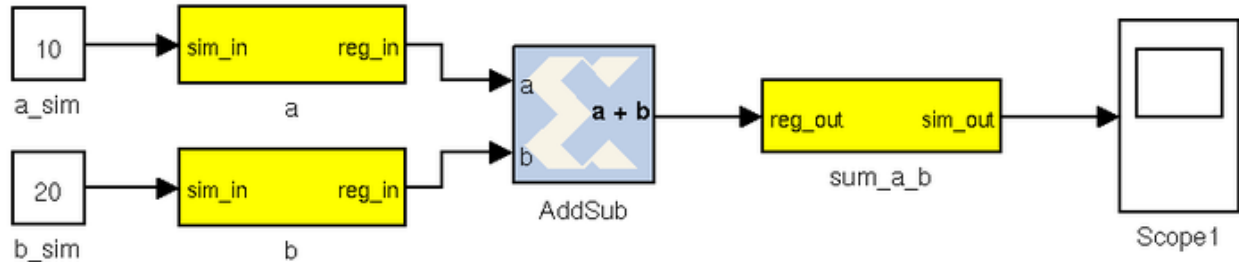
Do so by right-clicking and unchecking Format → Show Block Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



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 $a+b = \text{sum_a_b}$.



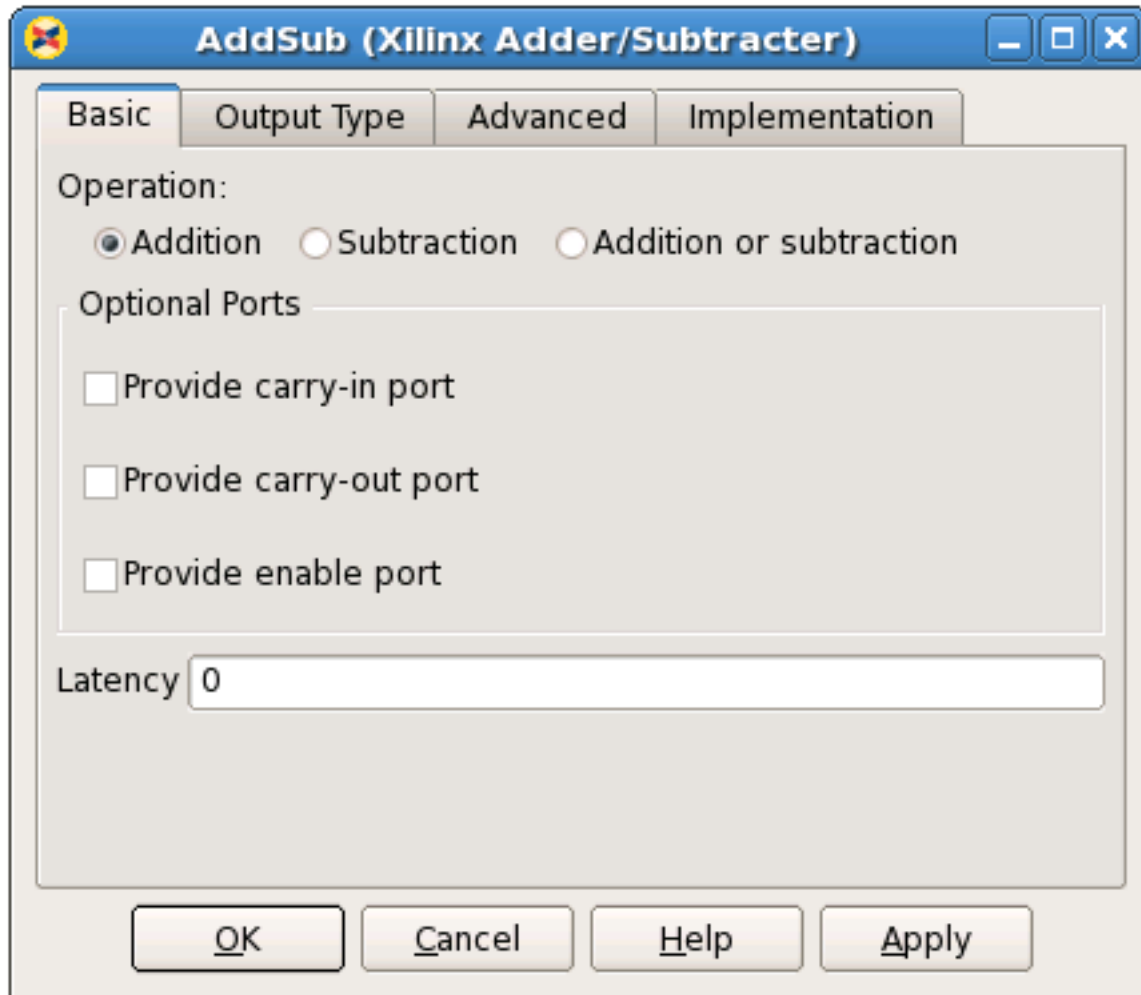
Add the software registers

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.

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.



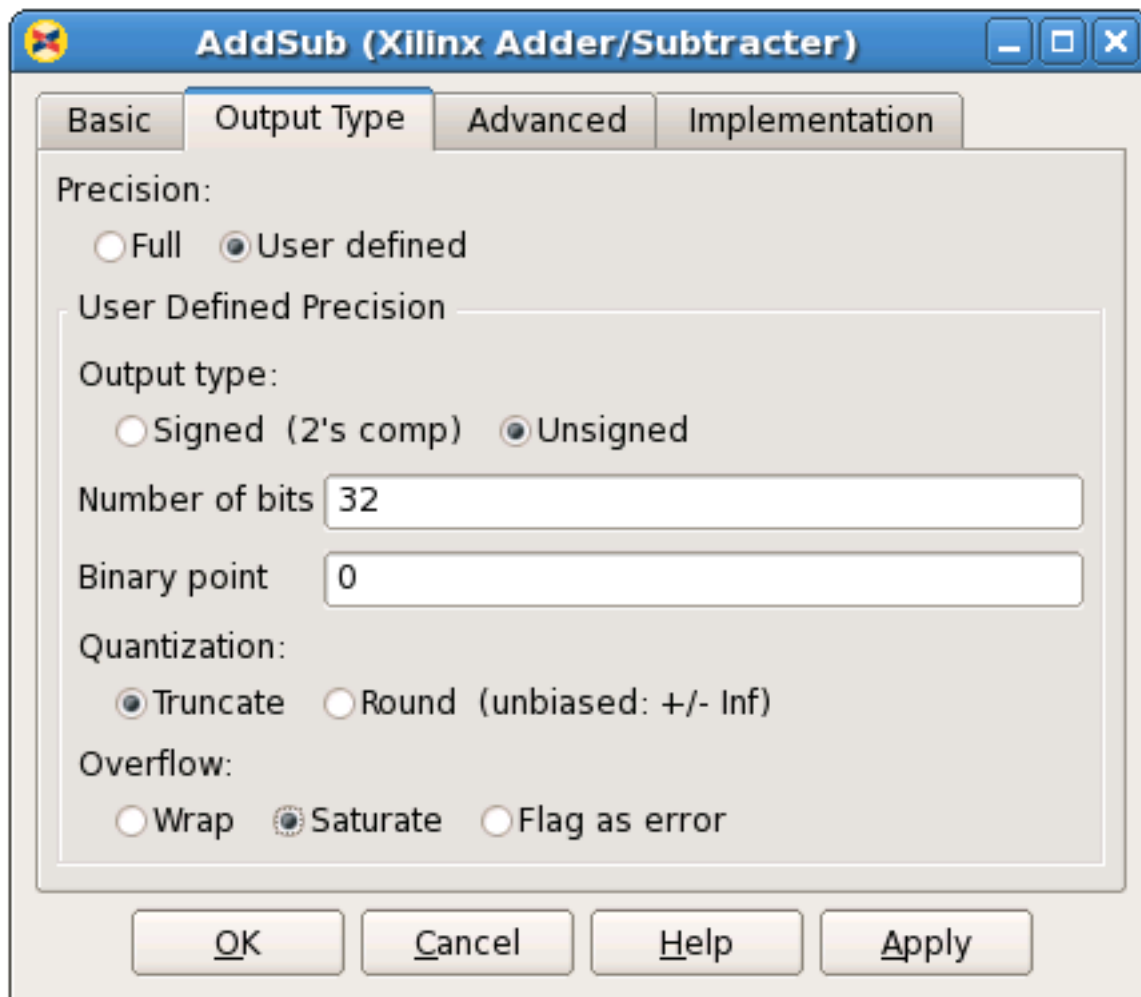
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

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 $2^{32} - 1$.

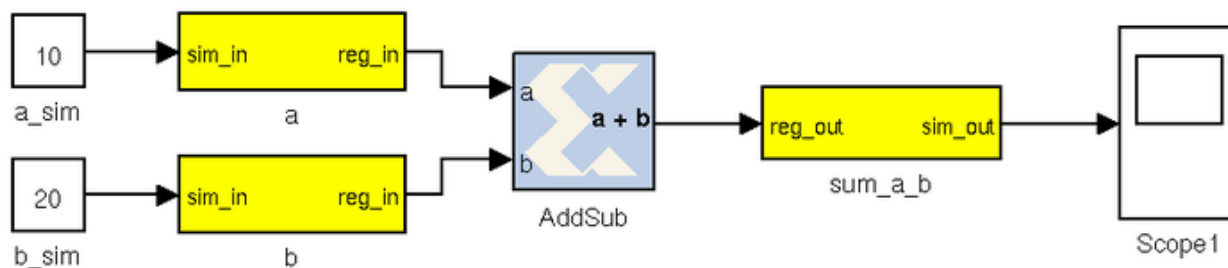


Add the scope and simulation inputs

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.

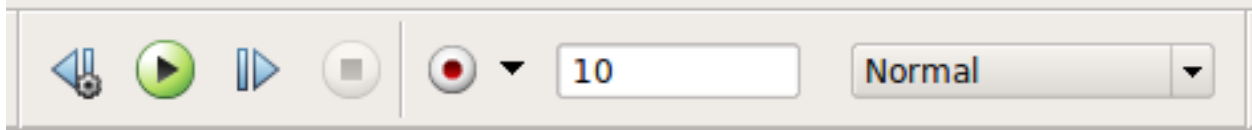
Connect it all together

Like this:



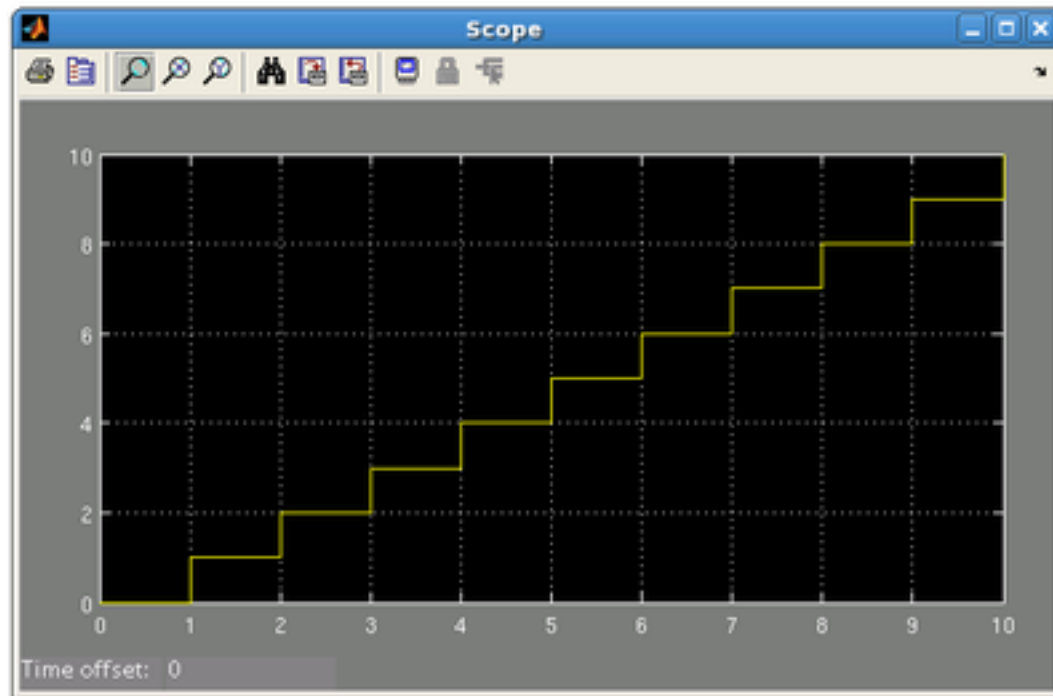
Simulating

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.

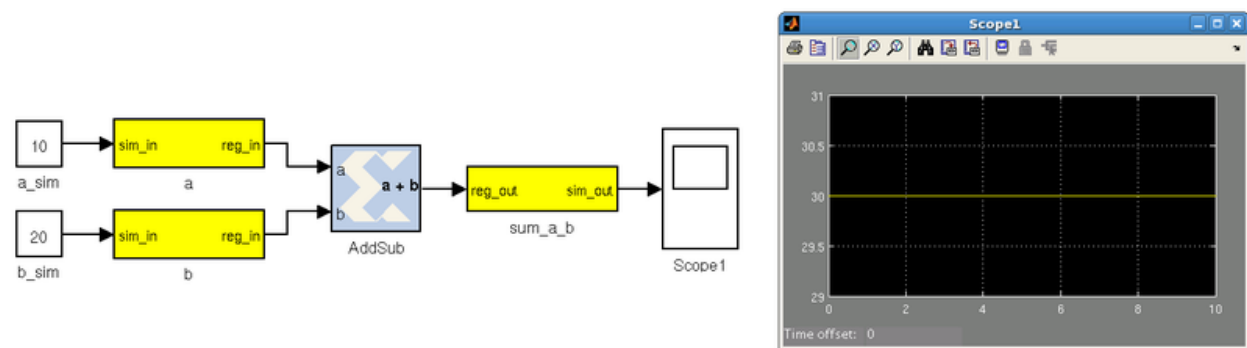


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

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...

Compiling

Essentially, you have constructed three completely separate little instruments.

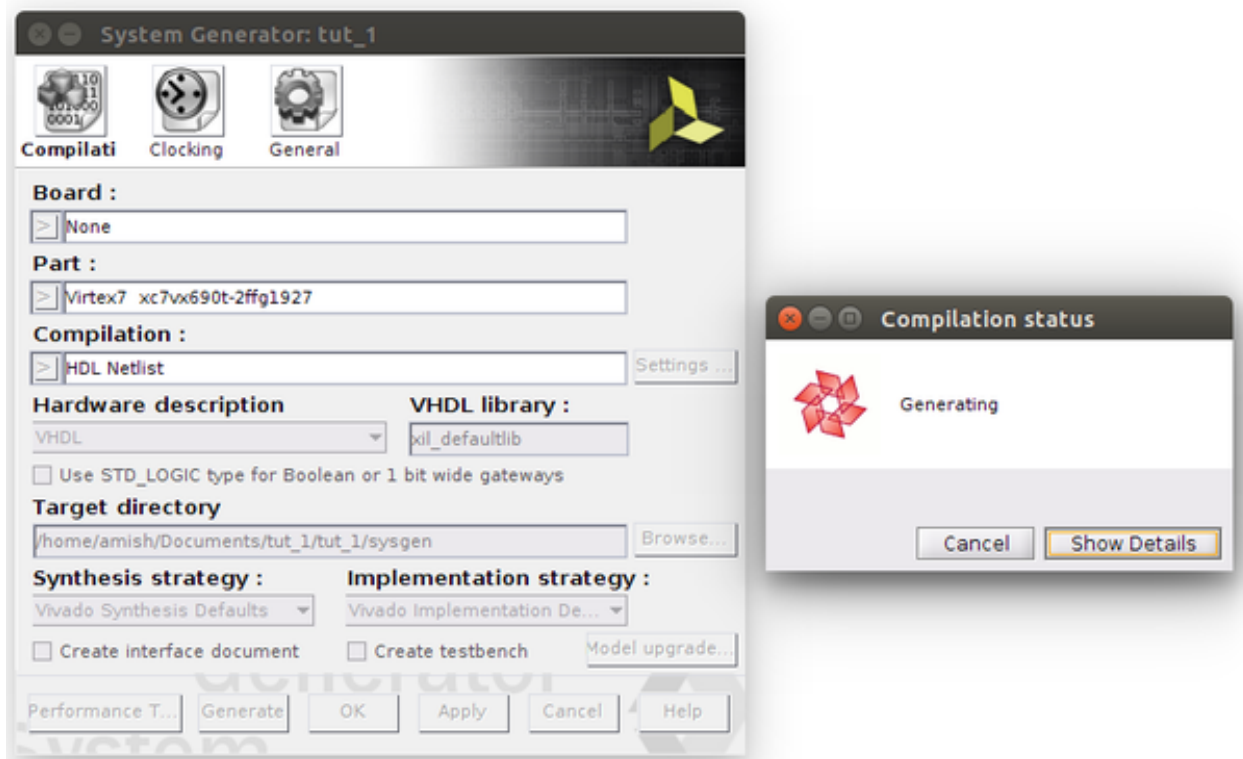
1. You have a flashing LED,
2. A counter which you can start/stop/reset from software, and
3. A simple adder.

These components are all clocked off the same clock source specified in your platform's properties, but they will operate independently.

In order to compile this to an FPGA bitstream, execute the following command in the MATLAB Command Line window. **THIS COMMAND DEPENDS WHICH PLATFORM YOU ARE TARGETING:**

```
>> casper_xps
```

When a GUI pops up, click "Compile!". This will run the complete build process, which consists of two stages. The first involving Xilinx's System Generator, which compiles any Xilinx blocks in your Simulink design to a circuit which can be implemented on your FPGA. While System Generator is running, you should see the following window pop up:



After this, the second stage involves synthesis of your design through Vivado, which goes about turning your design into a physical implementation and figuring out where to put the resulting components and signals on your FPGA. Finally the toolflow will create the final output fpg file that you will use to program your FPGA. This file contains the bitstream (the FPGA configuration information) as well as meta-data describing what registers and other yellow blocks are in your design. This file will be created in the 'bit_files' directory. **Note: Compile time is approximately 15-20 minutes.**


```

amish@amish-desktop:~/Documents/tut_1$ cd tut_1/
myproj/ outputs/ sysgen/
amish@amish-desktop:~/Documents/tut_1$ cd tut_1/outputs/
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$ ls -lt
total 8336
-rw-rw-r-- 1 amish amish 1249412 Jul 25 14:18 tut_1_2017-7-25_1355.fpg
-rw-rw-r-- 1 amish amish 7280615 Jul 25 14:18 tut_1_2017-7-25_1355.bof
amish@amish-desktop:~/Documents/tut_1/tut_1/outputs$ █

```

Programming the FPGA

Reconfiguration of CASPER FPGA boards is achieved using the casperfpga python library, created by the SA-SKA group.

Getting the required packages

These are pre-installed on the server in the workshop and you do not need to do any further configuration. However, should you want to run these tutorials on your own machines, you should download the latest casperfpga libraries from [here](#).

Copy your .fpg file to your Server

As per the previous figure, navigate to the outputs folder and (secure)copy this across to a test folder on the workshop server. Instructions to do this are available [here](#)

Connecting to the board

SSH into the server that the ROACH board is connected to and navigate to the folder in which your .fpg file is stored.

Start interactive python by running:

```
$ ipython
```

Now import the fpga control library. This will automatically pull-in the KATCP library and any other required communications libraries.

```
import casperfpga
```

To connect to the board we create a CasperFpga instance; let's call it fpga. The CasperFpga constructor requires just one argument: the IP hostname or address of your FPGA board.

```
fpga = casperfpga.CasperFpga('<roach2 hostname or ip_address>')
```

The first thing we do is program the FPGA with the .fpg file which your compile generated.

```
fpga.upload_to_ram_and_program('<your_fpgfile.fpg>')
```

Should the execution of this command return true, you can safely assume the FPGA is now configured with your design. You should see the LED on your board flashing. Go check! All the available/configured registers can be displayed using:

```
fpga.listdev()
```

The adder and counter can be controlled by **writing to** and **reading from** registers added in the design using:

```
fpga.write_int('a',10)
fpga.write_int('b',20)
fpga.read_int('sum_a_b')
```

With any luck, the sum returned by the FPGA should be correct.

You can also try writing to the counter control registers in your design. You should find that with appropriate manipulation of the control register, you can make the counter start, stop, and return to zero.

```
fpga.write_int('counter_ctrl', 1)
fpga.read_uint('counter_value')
```

Conclusion

This concludes the first CASPER Tutorial. You have learned how to construct a simple Simulink design, program an FPGA board and interact with it with Python using `casperfpga`. Congratulations!

While the design you made might not be very complicated, you now have the basic skills required to build more complex designs which are covered in later tutorials.

1.2.2 Tutorial 2: 10GbE Interface

Introduction

In this tutorial, you will create a simple Simulink design which uses the ROACH2's (or ROACH1's) 10GbE ports to send data at high speeds to another port. This could just as easily be another FPGA board or a computer with a 10GbE network interface card. In addition, we will learn to control the design remotely, using a supplied Python library for KATCP.

In this tutorial, a counter will be transmitted through one SFP+ port and back into another. This will allow a test of the communications link. This test can be used to test the link between boards and the effect of different cable lengths on communication robustness.

Background

ROACH2 boards have four SFP+ ports on a single 10GbE Mezzanine Card. The Ethernet interface is driven by an on-board 156.25MHz crystal oscillator. This clock is then multiplied up on the FPGA by a factor of 66. Thus, the speed on the wire is actually $156.25\text{MHz} \times 66 = 10.3125\text{ Gbps}$. However, 10GbE over single-lane SFP+ connectors uses 64b/66b encoding, which means that for every 66 bits sent, 64 bits are actually transmitted. This is to ensure proper clocking, since the receiver recovers and locks-on to the transmitter's clock and requires edges in the data. Imagine transmitting a string of 0xFF or 0b11111111... which would otherwise generate a DC level on the line, now an extra two bits are introduced which includes a zero bit which the receiver can use to recover the clock and byte endings. See [here](#) for more information.

For this reason, we actually get 10Gbps usable data rate. CASPER's 10GbE Simulink core sends and receives UDP over IPv4 packets. These IP packets are wrapped in Ethernet frames. Each Ethernet frame requires a 38 byte header, IPv4 requires another 20 bytes and UDP a further 16. So, for each packet of data you send, you will incur a cost of at least 74 bytes. I say at least, because the core will zero-pad some headers to be on a 64-bit boundary. You will thus

never achieve 10Gbps of usable throughput, though you can get close. It pays to send larger packets if you are trying to get higher throughputs.

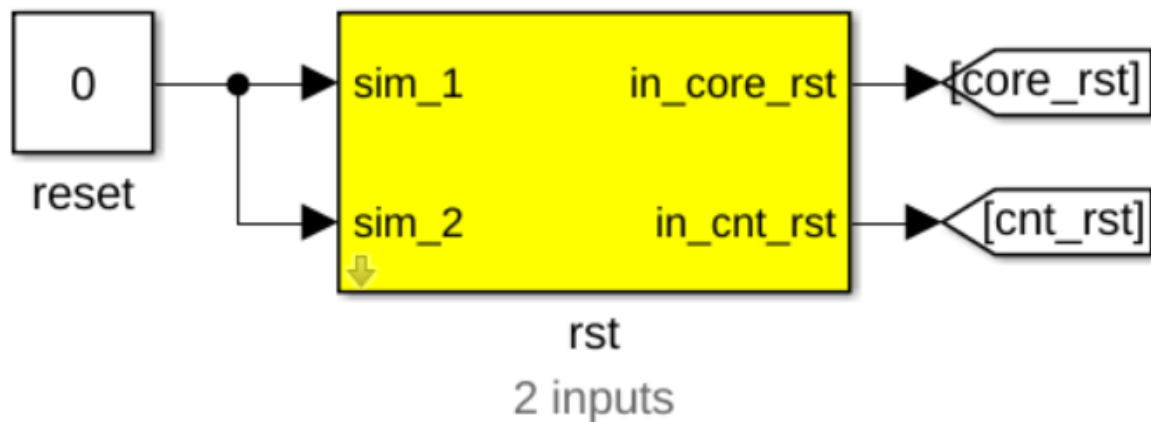
The maximum payload length of the CASPER 10GbE core is 8192 bytes (implemented in BRAM) plus another 512 (implemented in distributed RAM) which is useful for an application header. These ports (and hence part of the 10 GbE cores) run at 156.25MHz, while the interface to your design runs at the FPGA clock rate (`sys_clk`, `adcX_clk` etc). The interface is asynchronous, and buffers are required at the clock boundary. For this reason, even if you send data between two ROACH boards which are running off the same hard-wired clock, there will be jitter in the data. A second consideration is how often you clock values into the core when you try to send data. If your FPGA is running faster than the core, and you try and clock data in on every clock cycle, the buffers will eventually overflow. Likewise for receiving, if you send too much data to a board and cannot clock it out of the receive buffer fast enough, the receive buffers will overflow and you will lose data. In our design, we are clocking the FPGA at 100 MHz, with the cores running at 156.25MHz. We can thus clock data into the TX buffer continuously without worrying about overflows.

Create a new model

Start Matlab and open Simulink (either by typing ‘`simulink`’ on the Matlab command line, or by clicking on the Simulink icon in the taskbar). A template is provided for Tut2 with a pre-created packet generator in the `tutorials_devel` git repository. Get a copy of this template and save it. Make sure the `XSG_config_block` is configured for ROACH2 (or ROACH1, if that’s what you’re using). Specify a clock frequency of 100 MHz and the clock source “`sys_clock`”.

Add reset logic

A very important piece of logic to consider when designing your system is how, when and what happens during reset. In this example we shall control our resets via a software register. We shall have two independent resets, one for the 10GbE cores which shall be used initially, and one to reset the user logic which may be used more often to restart the user part of the system. Construct reset circuitry as shown below.



Add a software register

Use a software register yellow block from the CASPER XPS System Blockset for the `rst` block. Rename it to `rst`.

It used to be that every register you inserted had to be natively 32-bits, and you were responsible for slicing these 32 bits into different signals if you want to control multiple flags. The latest block can implicitly break the 32-bit registers out into separate names signals, so we’ll use that. The downside is there are a bunch of settings to configure

– you need to set up the names and data types of your register subfields. The settings you need are “NOTE: due to missing MATLAB licenses on the 2017 conference servers, you can’t use do this for ROACH2.” For ROACH2 take the default block, configure the I/O direction to be From Processor and then manually slice the bottom bit of the output to make the cnt_rst signal, and bit 1 to make the core_rst signal. You can check the example ROACH2 design to see how to do this.

Add Goto blocks

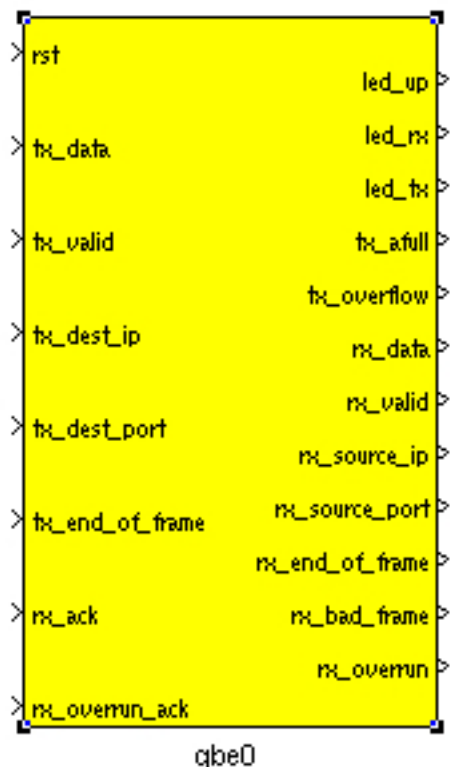
Add two Goto blocks from Simulink->Signal Routing. Configure them to have the tags as shown (core_rst and cnt_rst). These tags will be used by associated From (also found in Simulink->Signal Routing) blocks in other parts of the design. These help to reduce clutter in your design and are useful for control signals that are routed to many destinations. They should not be used a lot for data signals as it reduces the ease with which data flow can be seen through the system.

Add 10GbE and associated registers for data transmission

We will now add the 10GbE block to transmit a counter at a programmable rate.

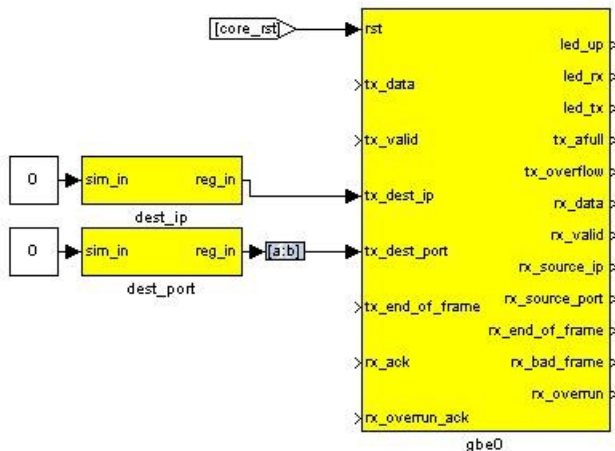
Add a 10GbE block for data transmission

Add a ten_GbE yellow block from the CASPER XPS System Blockset. It will be used to transmit data and we shall add another later to receive data. Rename it gbe0. Double click on the block to configure it and set it to be associated with SFP+ port 0. If your application can guarantee that it will be able to use received data straight away (as our application can), shallow receive buffers can be used to save resources. This optimisation is not necessary in this case as we will use a small fraction of resources in the FPGA.



Add registers to provide the target IP address and port number

Add two yellow-block software registers to provide the destination IP address and port number for transmission with the data. Name one `dest_ip` and the other `dest_port`. The registers should be configured to receive their values from the processor. Connect them to the appropriate inputs of the `gbe0 10GbE` block as shown. A Slice block is required to use the lower 16 bits of data from the `dest_port` register. Constant blocks from Simulink->Sources with 0 values are attached to the simulation inputs of the software registers. The destination port and IP address are not important in this system as it is a loopback example. Add a From block from Simulink->Signal Routing and set the tag to use `core_rst`, this enables one to reset the block.



Create a subsystem to generate a counter to transmit as data

We will now implement logic to generate a counter to transmit as data. This is already included in the Template for Tut 2. Some details are provided here for completeness.

Construct a subsystem for data generation logic

It is often useful to group related functionality and hide the details. This reduces drawing space and complexity of the logic on the screen, making it easier to understand what is happening. Simulink allows the creation of Subsystems to accomplish this.

These can be copied to places where the same functionality is required or even placed in a library for use in other projects and by other people. To create a subsystem, one can highlight the logical elements to be encapsulated, then right-click and choose Create Subsystem from the list of options. You can also simply add a Subsystem block from Simulink->Ports & Subsystems.

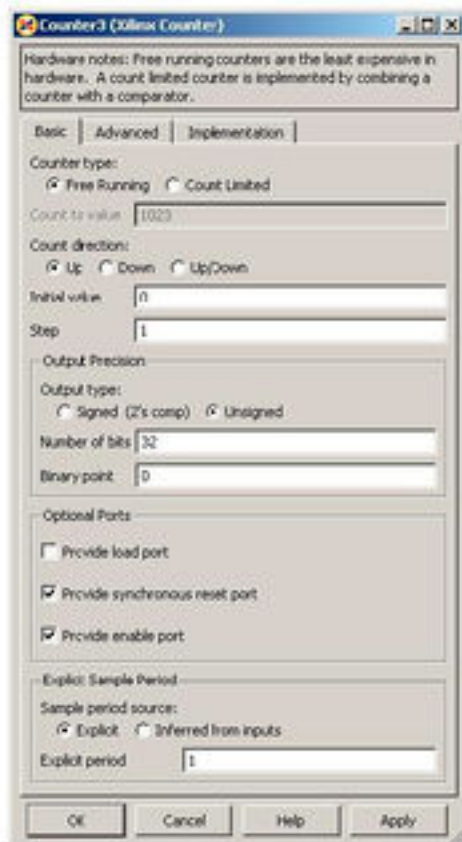
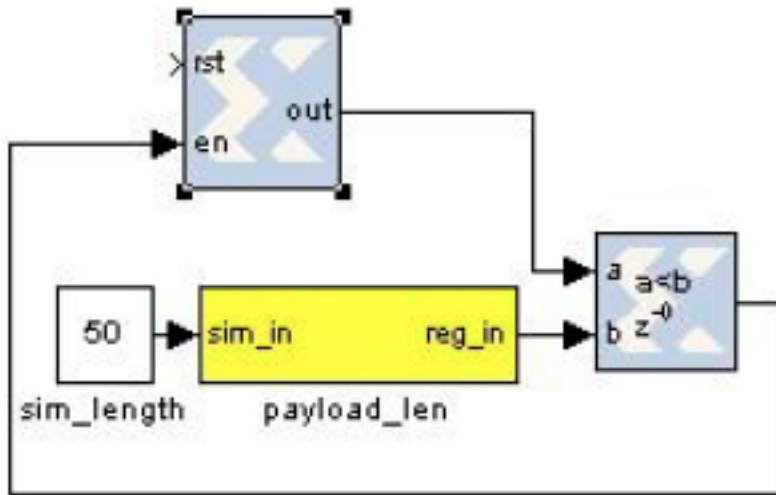
Subsystems inherit variables from their parent system. Simulink allows one to create a variable whose scope is only a particular subsystem. To do this, right-click on a subsystem and choose the Create Mask option. The mask created for that particular subsystem allows one to add parameters that appear when you double-click on the icon associated with the subsystem.

The mask also allows you to associate an initialisation script with a particular subsystem. This script is called every time a mask parameter is modified and the Apply button clicked. It is especially useful if the internal structure of a subsystem must change based on mask parameters. Most of the interesting blocks in the CASPER library use these initialisation scripts.

Drop a subsystem block into your design and rename it `pkt_sim`. Then double-click on it to add logic.

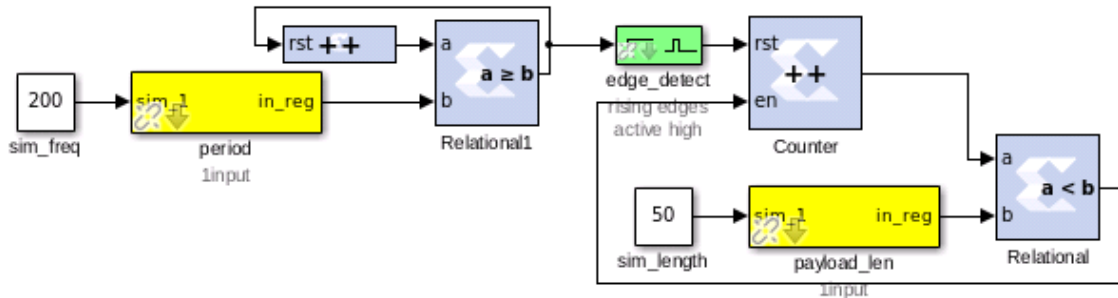
Add a counter to generate a certain amount of data

Add a Counter block from Xilinx Blockset->Basic Elements and configure it to be unsigned, free-running, 32-bits, incrementing by 1 as shown. Add a Relational block, software register and Constant block as shown. In simulation this circuit will generate a counter from 0 to 49 and then stop counting. This will allow us to generate 50 data elements before stopping.



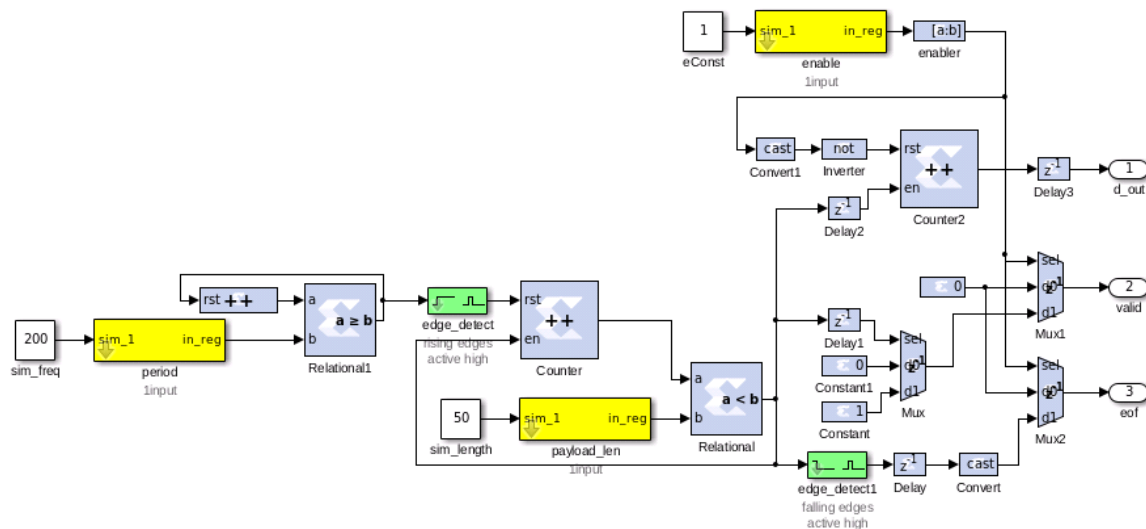
Add a counter to limit the data rate

As mentioned earlier in this tutorial, it is impossible to supply data to the 10GbE transmission block at the full clock rate. This would mean transmitting a 64-bit word at 200MHz, and the 10GbE standard only supports up to 156.25MHz data transmission. We thus want to generate data in bursts such that the transmission FIFOs do not overflow. We thus add circuitry to limit the data rate as shown below. The logic that we have added on the left generates a reset at a fixed period determined by the software register. This will trigger the generation of a new packet of data as before. In simulation this allows us to limit the data rate to $50/200 * 200\text{MHz} = 50\text{MHz}$. Using these values in actual hardware would limit the data rate to $(50/(8/10 * 156.25)) = 4\text{Gbps}$.



Finalise logic including counter to be used as data

We will now finalise the data generation logic as shown below. To save time, use the existing logic provided with the tutorial. Counter1 in the illustration generates the actual data to be transmitted and the enable register allows this data stream to the transmitting 10GbE core to be turned off and on. Logic linked to the eof output port provides an indication to the 10GbE core that the final data word for the frame is being sent. This will trigger the core to begin transmission of the frame of data using the IP address and port number specified.



Receive blocks and logic

The receive logic is composed of another 10GbE yellow block with the transmission interface inputs all tied to 0 as no transmission is to be done, however Simulink requires all inputs to be connected. Connecting them to 0 should ensure that during synthesis the transmission logic for this 10GbE block is removed. Double click on the block to configure it and set it to be associated with SFP+ port 1.

Buffers to capture received and transmitted data

The casperfpga Python package contains all kinds of methods to interact with your 10GbE cores. For example, grabbing packets from the TX and RX stream, or counting the number of packets sent and received are all supported, as long as you turn on the appropriate functionality in the 10GbE yellow block. The settings we'll use are –

Block Parameters: gbe0

ten_GbE_v2 (mask)

This block sends and receives UDP frames (packets). It accepts a 64 bit wide data stream with user-determined frame breaks. The data stream is wrapped in a UDP frame for transmission. Incoming UDP packets are unwrapped and the data presented as a 64 bit wide stream.

Core Debug counters

Port 0

☒ Shallow RX Fifo (Beware overruns!)

☐ Enable Large TX Frames (8k+512)

☒ ----- Show Implementation Parameters -----

Pre-emphasis 3

Differential swing 800

☒ Enable fabric on startup

Fabric MAC Address
hex2dec('123456780000')

Fabric IP Address
 $192 \cdot (2^{24}) + 168 \cdot (2^{16}) + 5 \cdot (2^8) + 20 \cdot (2^0)$

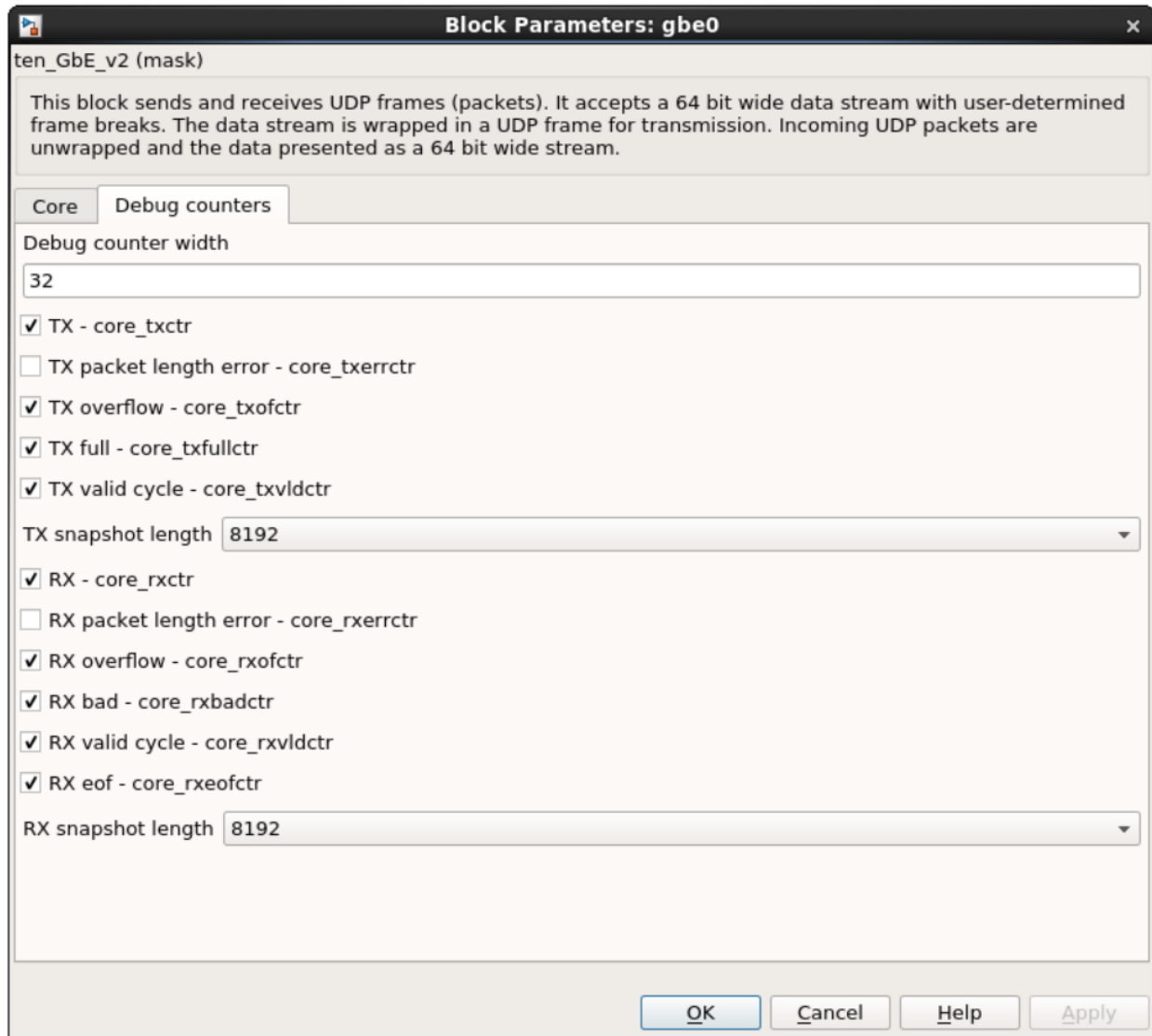
Fabric UDP Port
10000

Fabric Gateway
1

☒ Enable CPU RX

☒ Enable CPU TX

OK Cancel Help Apply



You can see how to use these functions in the software that accompanies this tutorial.

LEDs and status registers

You can also sprinkle around other registers or LEDs to monitor status of core parameters, or give visual feedback that the design is doing something sane. Check out the reference model for some examples of potentially useful monitoring circuitry.

Compilation

Compiling this design takes approximately 20 to 30 minutes. A pre-compiled binary (.fpg file) is made available to save time.

Programming and interacting with the FPGA

A pre-written python script, “roach2_tut_tge.py” is provided. This script programs the fpga with your complied design (.fpg file) configures the 10GbE Ports and initiates data transfer. The script is run using:

```
./roach2_tut_tge.py <ROACH_IP_ADDRESS>
```

If everything goes as expected, you should see a whole bunch of lines running across your screen as the code sets up the IP/MAC parameters of the 10GbE cores and checks their status, and that the data the cores are sending and receiving are consistent. Have a look at this code to see how one uses the more advanced (i.e. more complex that `read_int`, and `write_int`) methods `casperfpga` makes available. Documentation for `casperfpga` is still a work in progress(!) but the basic idea is that when you instantiate a `CasperFpga`, the software intelligently builds python objects into this instance, based on what you put in your design. For example, your Ethernet cores should show up as objects `CasperFpga.gbes.<simulink_block_name>` (or `CasperFpga.gbes['simulink_block_name']`) which have useful methods like “`setup`”, which sets the core’s IP/MAC address, or “`print_10gbe_core_details`” which will print out useful status information, like the current state of the core’s ARP cache. iPython and tab-complete are your friend here, there are lots of handy methods to discover. (I’m still discovering them now :))

The control software should be(!) well-commented, to explain what’s going on behind the scene as the software interacts with your FPGA design.

Conclusion

This concludes Tutorial 2. You have learned how to utilize the 10GbE ports on a ROACH to send and receive UDP packets. You also learned how to further use the Python to program the FPGA and control it remotely using some of the OOP goodies available in `casperfpga`.

1.2.3 Tutorial 3: Wideband Spectrometer

Introduction

A spectrometer is something that takes a signal in the time domain and converts it to the frequency domain. In digital systems, this is generally achieved by utilising the FFT (Fast Fourier Transform) algorithm. However, with a little bit more effort, the signal to noise performance can be increased greatly by using a Polyphase Filter Bank (PFB) based approach.

When designing a spectrometer for astronomical applications, it’s important to consider the science case behind it. For example, pulsar timing searches will need a spectrometer which can dump spectra on short timescales, so the rate of change of the spectra can be observed. In contrast, a deep field HI survey will accumulate multiple spectra to increase the signal to noise ratio. It’s also important to note that “bigger isn’t always better”; the higher your spectral and time resolution are, the more data your computer (and scientist on the other end) will have to deal with. For now, let’s skip the science case and familiarize ourselves with an example spectrometer.

Setup

This tutorial comes with a completed model file, a compiled bitstream, ready for execution on ROACH, as well as a Python script to configure the ROACH and make plots. [Here](#)

Spectrometer Basics

When designing a spectrometer there are a few main parameters of note:

- **Bandwidth:** The width of your frequency spectrum, in Hz. This depends on the sampling rate; for complex sampled data this is equivalent to:

$$BW = \text{sampling rate} = \frac{1}{\text{sampling period}}$$

In contrast, for real or Nyquist sampled data the rate is half this:

$$BW = \frac{\text{sampling rate}}{2} = \frac{1}{2 \times \text{sampling period}}$$

as two samples are required to reconstruct a given waveform .

- **Frequency resolution:** The frequency resolution of a spectrometer, Δf , is given by

$$\Delta f = \frac{BW}{\text{no. channels}},$$

and is the width of each frequency bin. Correspondingly, Δf is a measure of how precise you can measure a frequency.

- **Time resolution:** Time resolution is simply the spectral dump rate of your instrument. We generally accumulate multiple spectra to average out noise; the more accumulations we do, the lower the time resolution. For looking at short timescale events, such as pulsar bursts, higher time resolution is necessary; conversely, if we want to look at a weak HI signal, a long accumulation time is required, so time resolution is less important.

Simulink / CASPER Toolflow

Simulink Design Overview

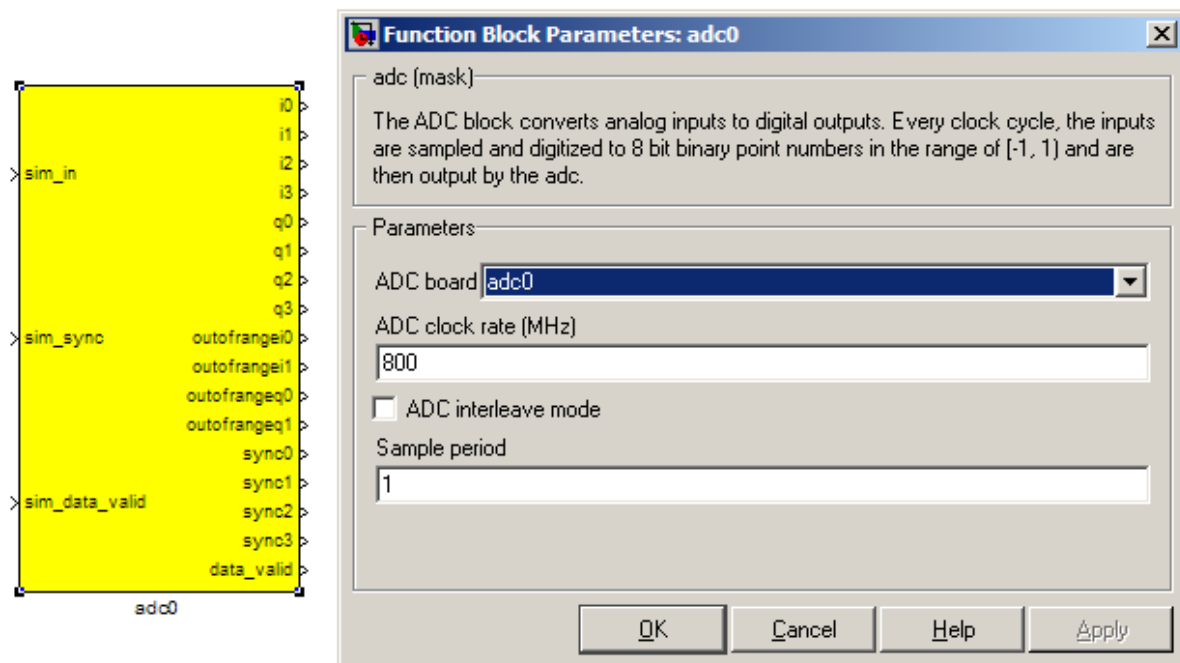
If you're reading this, then you've already managed to find all the tutorial files. Jason has gone to great effort to create an easy to follow simulink model that compiles and works. By now, I presume you can open the model file and have a vague idea of what's happening. The best way to understand fully is to follow the arrows, go through what each block is doing and make sure you know why each step is done. To help you through, there's some "blockumentation" in the appendix, which should (hopefully) answer all questions you may have. A brief rundown before you get down and dirty:

- The all important Xilinx token is placed to allow System Generator to be called to compile the design.
- In the MSSGE block, the hardware type is set to "ROACH:sx95t" and clock rate is specified as 200MHz.
- The input signal is digitised by the ADC, resulting in four parallel time samples of 8 bits each clock cycle. The ADC runs at 800MHz, which gives a 400MHz nyquist sampled spectrum. The output range is a signed number in the range -1 to +1 (ie 7 bits after the decimal point). This is expressed as fix_8_7.
- The four parallel time samples pass through the pfb_fir_real and fft_wideband_real blocks, which together constitute a polyphase filter bank. We've selected 212=4096 points, so we'll have a 211=2048 channel filter bank.
- You may notice Xilinx delay blocks dotted all over the design. It's common practice to add these into the design as it makes it easier to fit the design into the logic of the FPGA. It consumes more resources, but eases signal timing-induced placement restrictions.

- The real and imaginary (sine and cosine value) components of the FFT are plugged into power blocks, to convert from complex values to real power values by squaring. They are also scaled by a gain factor before being quantised...
- These power values are then requantized by the quant0 block, from 36.34 bits to 6.5 bits, in preparation for accumulation. This is done to limit bit growth.
- The requantized signals then enter the vector accumulators, vacc0 and vacc1, which are simple_bram_vacc 32 bit vector accumulators. Accumulation length is controlled by the acc_cntrl block.
- The accumulated signal is then fed into software registers, odd and even.

Without further ado, open up the model file and start clicking on things, referring the blockumentation as you go.

adc



The first step to creating a frequency spectrum is to digitize the signal. This is done with an ADC – an Analogue to Digital Converter. In Simulink, the ADC daughter board is represented by a yellow block. Work through the “iADC tutorial” if you’re not familiar with the iADC card.

The ADC block converts analog inputs to digital outputs. Every clock cycle, the inputs are sampled and digitized to 8 bit binary point numbers in the range of -1 to 1 and are then output by the ADC. This is achieved through the use of two’s-compliment representation with the binary point placed after the seven least significant bits. This means we can represent numbers from -128/128 through to 127/128 including the number 0. Simulink represents such numbers with a fix_8_7 moniker.

ADCs often internally bias themselves to halfway between 0 and -1. This means that you’d typically see the output of an ADC toggling between zero and -1 when there’s no input. It also means that unless otherwise calibrated, an ADC will have a negative DC offset.

The ADC has to be clocked to four times that of the FPGA clock. In this design the ADC is clocked to 800MHz, so the ROACH will be clocked to 200MHz. This gives us a bandwidth of 400MHz, as Nyquist sampling requires two samples (or more) each second.

INPUTS

OUTPUTS

The ADC outputs two main signals: *i* and *q*, which correspond to the coaxial inputs of the ADC board. In this tutorial, we'll only be using input *i*. As the ADC runs at 4x the FPGA rate, there are four parallel time sampled outputs: *i0*, *i1*, *i2* and *i3*. As mentioned before, these outputs are 8.7 bit.

pfb_fir_real



Function Block Parameters: pfb_fir_real

pfb_fir_real (mask)

Fold adders into DSPs: Causes adders to be absorbed into DSP blocks (supported in Virtex5)

Adder implementation: Cores using Fabric or DSP48 or behavioral HDL

Parameters

Size of PFB: ($2^?$ pnts)

12

Total Number of Taps:

4

Windowing Function: hamming

Number of Simultaneous Inputs: ($2^?$)

2

Make Biplex

0

Input Bitwidth:

8

Output Bitwidth:

18

Coefficient Bitwidth:

OK Cancel Help Apply

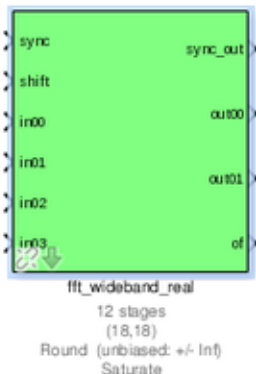
There are two main blocks required for a polyphase filter bank. The first is the `pfb_fir_real` block, which divides the signal into parallel 'taps' then applies finite impulse response filters (FIR). The output of this block is still a time-domain signal. When combined with the `FFT_wideband_real` block, this constitutes a polyphase filterbank.

INPUTS/OUTPUTS

As the ADC has four parallel time sampled outputs: i0, i1, i2 and i3, we need four parallel inputs for this PFB implementation.

PARAMETERS

fft_wideband_real



Function Block Parameters: `fft_wideband_real`

`fft_wideband_real` (mask)
A real-sampled wideband FFT.

Basic Latency Implementation

Number simultaneous streams
1

Size of FFT: (2[?] pts)
12

Number of Simultaneous Inputs: (2[?])
2

Input Bit Width
18

Input binary point
17

Coefficient Bit Width
18

☒ Unscramble output (ie, put channels in canonical order)

☐ Asynchronous operation

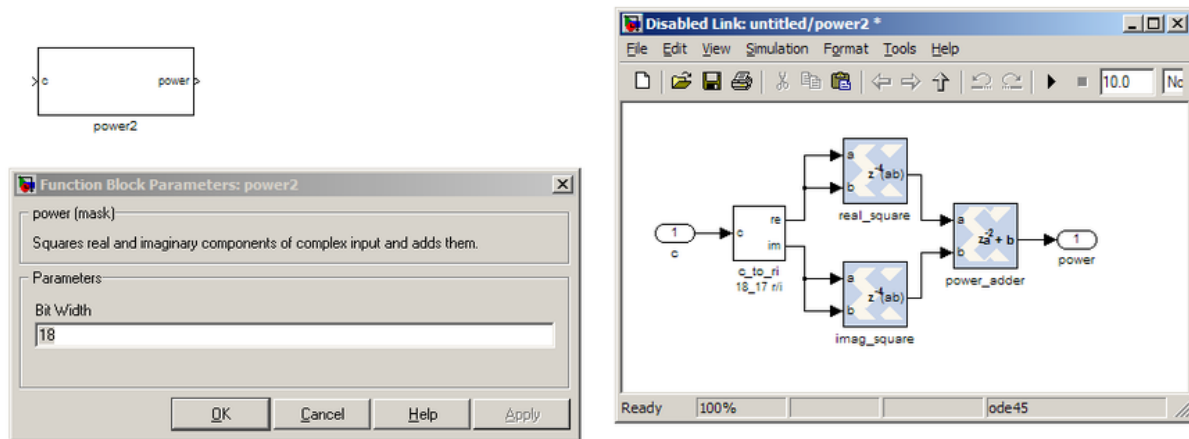
OK Cancel Help Apply

The `FFT_wideband_real` block is the most important part of the design to understand. The cool green of the FFT block hides the complex and confusing FFT butterfly biplex algorithms that are under the hood. You do need to have a working knowledge of it though, so I recommend reading Chapter 8 and Chapter 12 of Smith's free online DSP guide at (<http://www.dspguide.com/>). Parts of the documentation below are taken from the [[Block_Documentation | block documentation]] by Aaron Parsons and Andrew Martens.

INPUTS/OUTPUTS

PARAMETERS

power

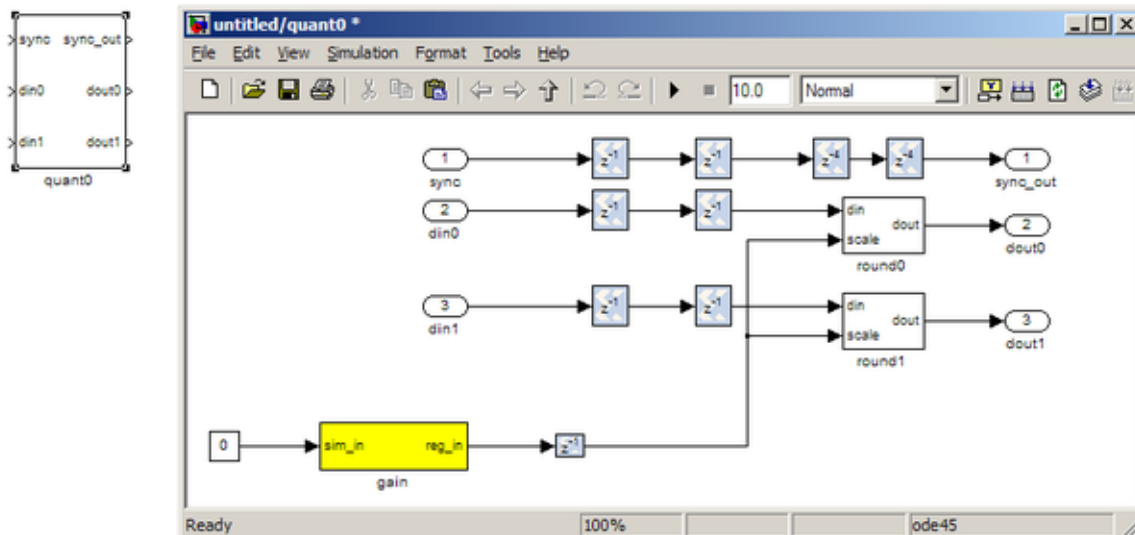


The power block computes the power of a complex number. The power block typically has a latency of 5 and will compute the power of its input by taking the sum of the squares of its real and imaginary components. The power block is written by Aaron Parsons and online documentation is by Ben Blackman. In our design, there are two power blocks, which compute the power of the odd and even outputs of the FFT. The output of the block is 36.34 bits; the next stage of the design re-quantizes this down to a lower bitrate.

INPUTS/OUTPUTS

PARAMETERS

quant



The quant0 was written by Jason Manley for this tutorial and is not part of the CASPER blockset. The block re-quantizes from 36.34 bits to 6.5 unsigned bits, in preparation for accumulation by the 32 bit bram_vacc block. This block also adds gain control, via a software register. The tut3.py script sets this gain control. You would not need to re-quantize if you used a larger vacc block, such as the 64bit one, but it's illustrative to see a simple example of

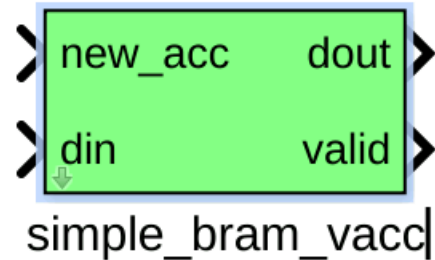
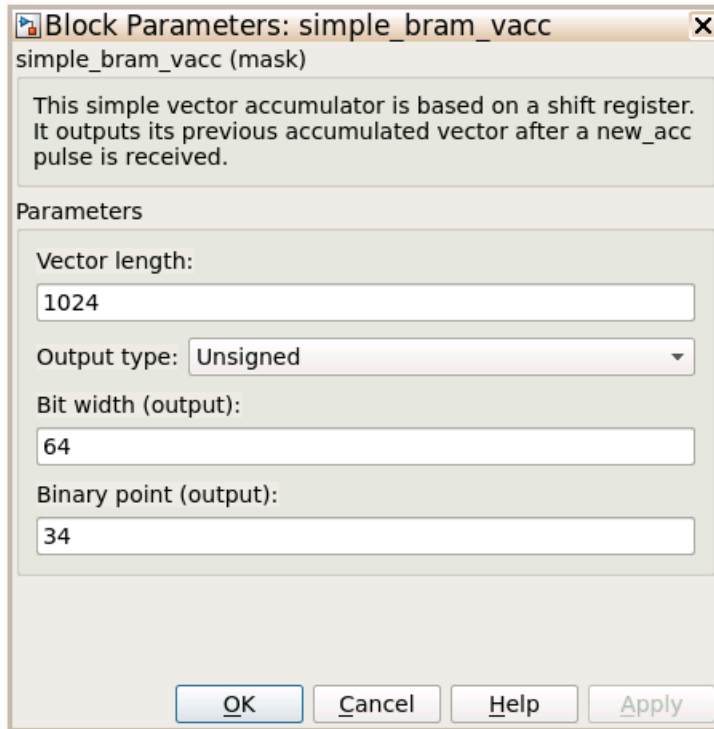
re-quantization, so it's in the design anyway. Note that the sync_out port is connected to a block, acc_cntrl, which provides accumulation control.

INPUTS/OUTPUTS

PARAMETERS

None.

simple_bram_vacc



The simple_bram_vacc block is used in this design for vector accumulation. Vector growth is approximately 28 bits each second, so if you wanted a really long accumulation (say a few hours), you'd have to use a block such as the qdr_vacc or dram_vacc. As the name suggests, the simple_bram_vacc is simpler so it is fine for this demo spectrometer. The FFT block outputs 1024 cosine values (odd) and 1024 sine values, making 2048 values in total. We have two of these bram vacc's in the design, one for the odd and one for the even frequency channels. The vector length is thus set to 1024 on both.

PARAMETERS

INPUTS/OUTPUTS

Even and Odd BRAMs



Block Parameters: even

Subsystem (mask)

Parameters

Output Data Type: Unsigned

Address width: 10

Data Width: 64

☒ Register Primitive Output

☒ Register Core Output

Optimization: Minimum_Area

Data Binary Point: 0

Initial values (simulation only): [0:2^10-1]

Sample rate: 1

OK Cancel Help Apply

The final blocks, odd and even, are shared BRAMs, which we will read out the values of using the tut3.py script.

PARAMETERS

INPUTS/OUTPUTS

Software Registers

There are a few **control registers**, led blinkers, and **snap** block dotted around the design too:

- **cnt_rst**: Counter reset control. Pulse this high to reset all counters back to zero.
- **acc_len**: Sets the accumulation length. Have a look in tut3.py for usage.
- **sync_cnt**: Sync pulse counter. Counts the number of sync pulses issued. Can be used to figure out board uptime and confirm that your design is being clocked correctly.
- **acc_cnt**: Accumulation counter. Keeps track of how many accumulations have been done.

- **led0_sync**: Back on topic: the led0_sync light flashes each time a sync pulse is generated. It lets you know your ROACH is alive.
- **led1_new_acc**: This lights up led1 each time a new accumulation is triggered.
- **led2_acc_clip**: This lights up led2 whenever clipping is detected.

There are also some [snap](#) blocks, which capture data from the FPGA fabric and makes it accessible to the Power PC. This tutorial doesn't go into these blocks (in its current revision, at least), but if you have the inclination, have a look at their [documentation](#). In this design, the snap blocks are placed such that they can give useful debugging information. You can probe these through [KATCP](#), as done in [Tutorial 1](#), if interested. If you've made it to here, congratulations, go and get yourself a cup of tea and a biscuit, then come back for part two, which explains the second part of the tutorial – actually getting the spectrometer running, and having a look at some spectra.

Configuration and Control

Hardware Configuration

The tutorial comes with a pre-compiled bof file, which is generated from the model you just went through (tut3.bof). Copy this over to your ROACH bofiles directory, chmod it to a+x as in the other tutorials, then load up your ROACH. You don't need to telnet in to the ROACH; all communication and configuration will be done by the python control script called tut3.py.

Next, you need to set up your ROACH. Switch it on, making sure that:

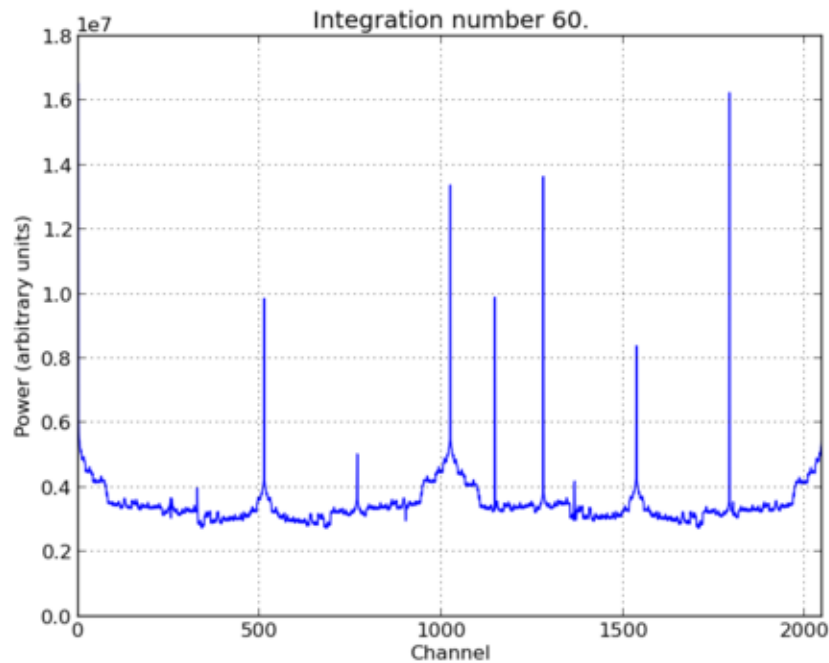
- You have your ADC in ZDOK0, which is the one nearest to the power supply.
- You have your clock source connected to clk_i on the ADC, which is the second on the right. It should be generating an 800MHz sine wave with 0dBm power.

The tut3.py spectrometer script

Once you've got that done, it's time to run the script. First, check that you've connected the ADC to ZDOK0, and that the clock source is connected to clk_i of the ADC. Now, if you're in linux, browse to where the tut3.py file is in a terminal and at the prompt type

```
./tut3.py <roach IP or hostname> -b <boffile name>
```

replacing with the IP address of your ROACH and with your boffile. You should see a spectrum like this:



In the plot, there should be a fixed DC offset spike; and if you're putting in a tone, you should also see a spike at the correct input frequency. If you'd like to take a closer look, click the icon that is below your plot and third from the right, then select a section you'd like to zoom in to. The digital gain (-g option) is set to maximum (0xffff_ffff) by default to observe the ADC noise floor. Reduce the gain (decrease the value (for a -10dBm input 0x100)) when you are feeding the ADC with a tone, as not to saturate the spectrum.

Now you've seen the python script running, let's go under the hood and have a look at how the FPGA is programmed and how data is interrogated. To stop the python script running, go back to the terminal and press ctrl + c a few times.

iPython walkthrough

The tut3.py script has quite a few lines of code, which you might find daunting at first. Fear not though, it's all pretty easy. To whet your whistle, let's start off by operating the spectrometer through iPython. Open up a terminal and type:

```
ipython
```

and press enter. You'll be transported into the magical world of iPython, where we can do our scripting line by line, similar to MATLAB. Our first command will be to import the python packages we're going to use:

```
import corr,time,numpy,struct,sys,logging,pylab
```

Next, we set a few variables:

```
katcp_port = 7147
roach = 'enter IP address or hostname here'
timeout = 10
```

Which we can then use in FpgaClient() such that we can connect to the ROACH and issue commands to the FPGA:

```
fpga = corr.katcp_wrapper.FpgaClient(roach,katcp_port, timeout)
```

We now have an fpga object to play around with. To check if you managed to connect to your ROACH, type:

```
fpga.is_connected()
```

Let's set the bitstream running using the progdev() command:

```
fpga.progdev('tut3.bof')
```

Now we need to configure the accumulation length and gain by writing values to their registers. For two seconds and maximum gain: accumulation length, $2 \cdot (2^{28}) / 2048$, or just under 2 seconds:

```
fpga.write_int('acc_len', 2 * (2**28) / 2048)
fpga.write_int('gain', 0xffffffff)
```

Finally, we reset the counters:

```
fpga.write_int('cnt_rst', 1)
fpga.write_int('cnt_rst', 0)
```

To read out the integration number, we use fpga.read_uint():

```
acc_n = fpga.read_uint('acc_cnt')
```

Do this a few times, waiting a few seconds in between. You should be able to see this slowly rising. Now we're ready to plot a spectrum. We want to grab the even and odd registers of our PFB:

```
a_0=struct.unpack('>1024i', fpga.read('even', 1024*4, 0))
a_1=struct.unpack('>1024i', fpga.read('odd', 1024*4, 0))
```

These need to be interleaved, so we can plot the spectrum. We can use a for loop to do this:

```
interleave_a=[]

for i in range(1024):
    interleave_a.append(a_0[i])
    interleave_a.append(a_1[i])
```

This gives us a 2048 channel spectrum. Finally, we can plot the spectrum using pyLab:

```
pylab.figure(num=1, figsize=(10, 10))
pylab.plot(interleave_a)
pylab.title('Integration number %i.' % acc_n)
pylab.ylabel('Power (arbitrary units)')
pylab.grid()
pylab.xlabel('Channel')
pylab.xlim(0, 2048)
pylab.show()
```

Voila! You have successfully controlled the ROACH spectrometer using python, and plotted a spectrum. Bravo! You should now have enough of an idea of what's going on to tackle the python script. Type exit() to quit ipython. tut3.py notes ==

Now you're ready to have a closer look at the tut3.py script. Open it with your favorite editor. Again, line by line is the only way to fully understand it, but to give you a head start, here's a few notes:

Connecting to the ROACH

To make a connection to the ROACH, we need to know what port to connect to, and the IP address or hostname of our ROACH. The connection is made on line 96:

```
fpga = corr.katcp_wrapper.FpgaClient(...)
```

The `katcp_port` variable is set on line 16, and the `roach` variable is passed to the script at the terminal (remember that you typed `python tut3.py roachname`). We can check if the connection worked by using `fpga.is_connected()`, which returns true or false:

```
if fpga.is_connected():
```

The next step is to get the right bitstream programmed onto the FPGA fabric. The bitstream is set on line 15:

```
bitstream = 'tut3.bof'
```

Then the `progdev` command is issued on line 108:

```
fpga.progdev(bitstream)
```

Passing variables to the script

Starting from line 64, you'll see the following code:

```
from optparse import OptionParser

p = OptionParser()
p.set_usage('tut3.py <ROACH_HOSTNAME_or_IP> [options]')
p.set_description(__doc__)

p.add_option('-l', '--acc_len', dest='acc_len',
             type='int', default=2*(2**28)/2048,
             help='Set the number of vectors to accumulate between dumps. default is 2*(2^28)/2048,
             ↳ or just under 2 seconds.')

p.add_option('-g', '--gain', dest='gain',
             type='int', default=0xffffffff,
             help='Set the digital gain (6bit quantisation scalar). Default is 0xffffffff (max),
             ↳ good for wideband noise. Set lower for CW tones.')

p.add_option('-s', '--skip', dest='skip', action='store_true',
             help='Skip reprogramming the FPGA and configuring EQ.')

opts, args = p.parse_args(sys.argv[1:])

if args==[]:
    print 'Please specify a ROACH board. Run with the -h flag to see all_
    ↳ options.\nExiting.'
    exit()
else:
    roach = args[0]
```

What this code does is set up some default parameters which we can pass to the script from the command line. If the flags aren't present, it will default to the values set here.

Conclusion

If you have followed this tutorial faithfully, you should now know:

- What a spectrometer is and what the important parameters for astronomy are.
- Which CASPER blocks you might want to use to make a spectrometer, and how to connect them up in Simulink.

- How to connect to and control a ROACH spectrometer using python scripting.

In the following tutorials, you will learn to build a correlator, and a polyphase filtering spectrometer using an FPGA in conjunction with a Graphics Processing Unit (GPU).

1.2.4 Tutorial 4: Wideband Pocket Correlator

Introduction

In this tutorial, you will create a simple Simulink design which uses the iADC board on ROACH and the CASPER DSP blockset to process a wideband (400MHz) signal, channelize it and output the visibilities through ROACH's PPC.

By this stage, it is expected that you have completed [tutorial 1](#) and [tutorial 2](#) and are reasonably comfortable with Simulink and basic Python. We will focus here on higher-level design concepts, and will provide you with low-level detail preimplemented.

Background

Some of this design is similar to that of the previous tutorial, the Wideband Spectrometer. So completion of [tutorial 3](#) is recommended.

Interferometry

In order to improve sensitivity and resolution, telescopes require a large collection area. Instead of using a single, large dish which is expensive to construct and complicated to maneuver, modern radio telescopes use interferometric arrays of smaller dishes (or other antennas). Interferometric arrays allow high resolution to be obtained, whilst still only requiring small individual collecting elements.

Correlation

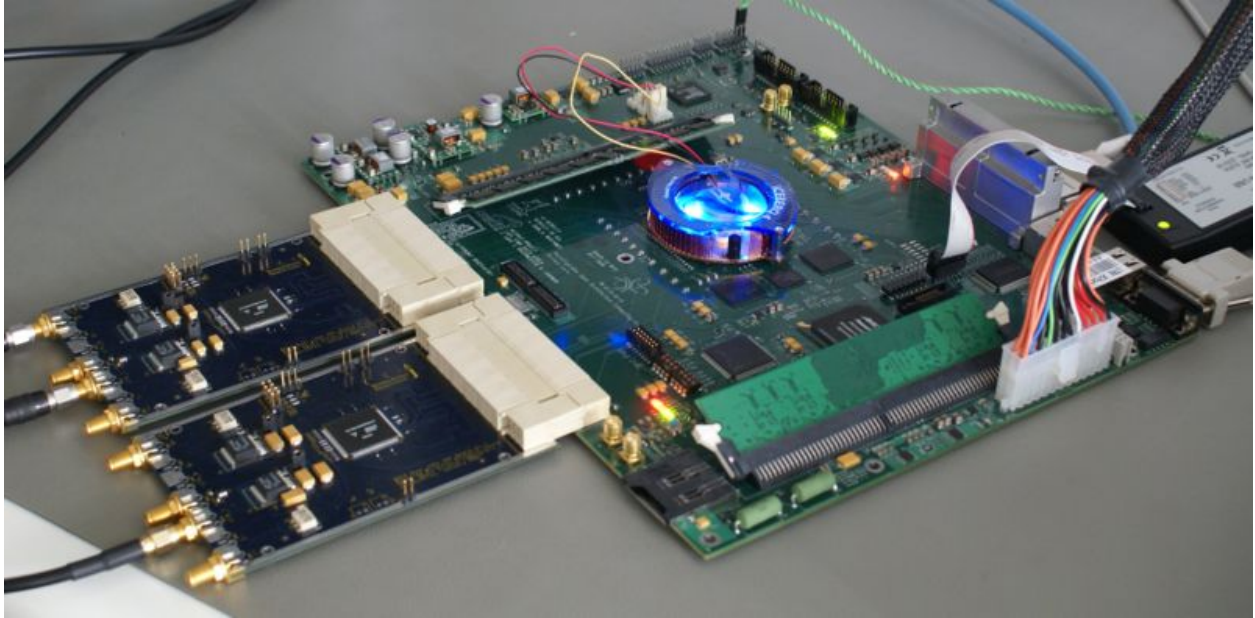
Interferometric arrays require the relative phases of antennas' signals to be measured. These can then be used to construct an image of the sky. This process is called correlation and involves multiplying signals from all possible antenna pairings in an array. For example, if we have 3 antennas, A, B and C, we need to perform correlation across each pair, AB, AC and BC. We also need to do auto-correlations, which will give us the power in each signal. ie AA, BB, CC. We will see this implemented later. The complexity of this calculation scales with the number of antennas squared. Furthermore, it is a difficult signal routing problem since every antenna must be able to exchange data with every other antenna.

Polarization

Dish type receivers are typically dual polarized (horizontal and vertical feeds). Each polarization is fed into separate ADC inputs. When correlating these antennae, we differentiate between full Stokes correlation or a half Stokes method. A full Stokes correlator does cross correlation between the different polarizations (ie for a given two antennas, A and B, it multiplies the horizontal feed from A with the vertical feed from B and vice-versa). A half stokes correlator only correlates like polarizations with each other, thereby halving the compute requirements.

The Correlator

The correlator we will be designing is a 4 input correlator as shown below. It uses 2 inputs from each of two ADCs. It can be thought of as a 2-input full Stokes correlator or as a four input single polarization correlator.



Setup

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

Creating Your Design

Create a new model

Having started Matlab, open Simulink (either by typing `simulink` on the Matlab command line, or by clicking the Simulink icon in the taskbar). Create a new model and add the Xilinx System Generator and XSG core config blocks as before in Tutorial 1.

System Generator and XSG Blocks

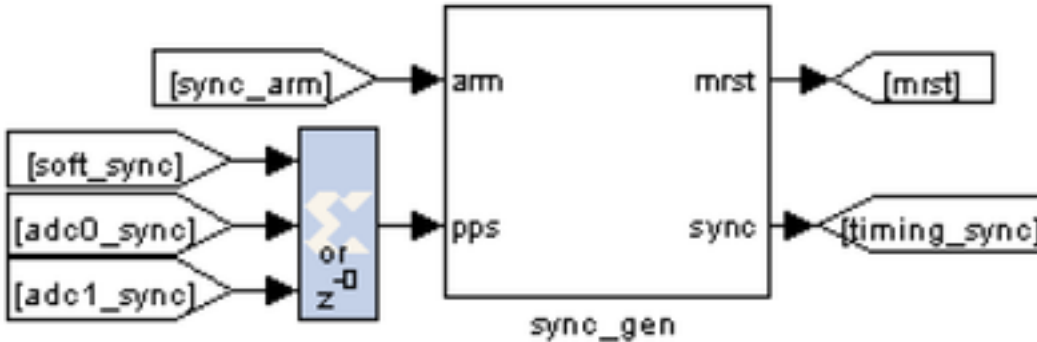


By now you should have used these blocks a number of times. Pull the System Generator block into your design from the Xilinx Blockset menu under Basic Elements. The settings can be left on default.

The XSG block can be found under the CASPER XPS System Blockset. Set the Hardware platform to ROACH:sx95t, the Clock Source to adc0_clk and the rest of the configuration as the default.

Make sure you have an ADC plugged into ZDOK0 to supply the FPGA's clock!

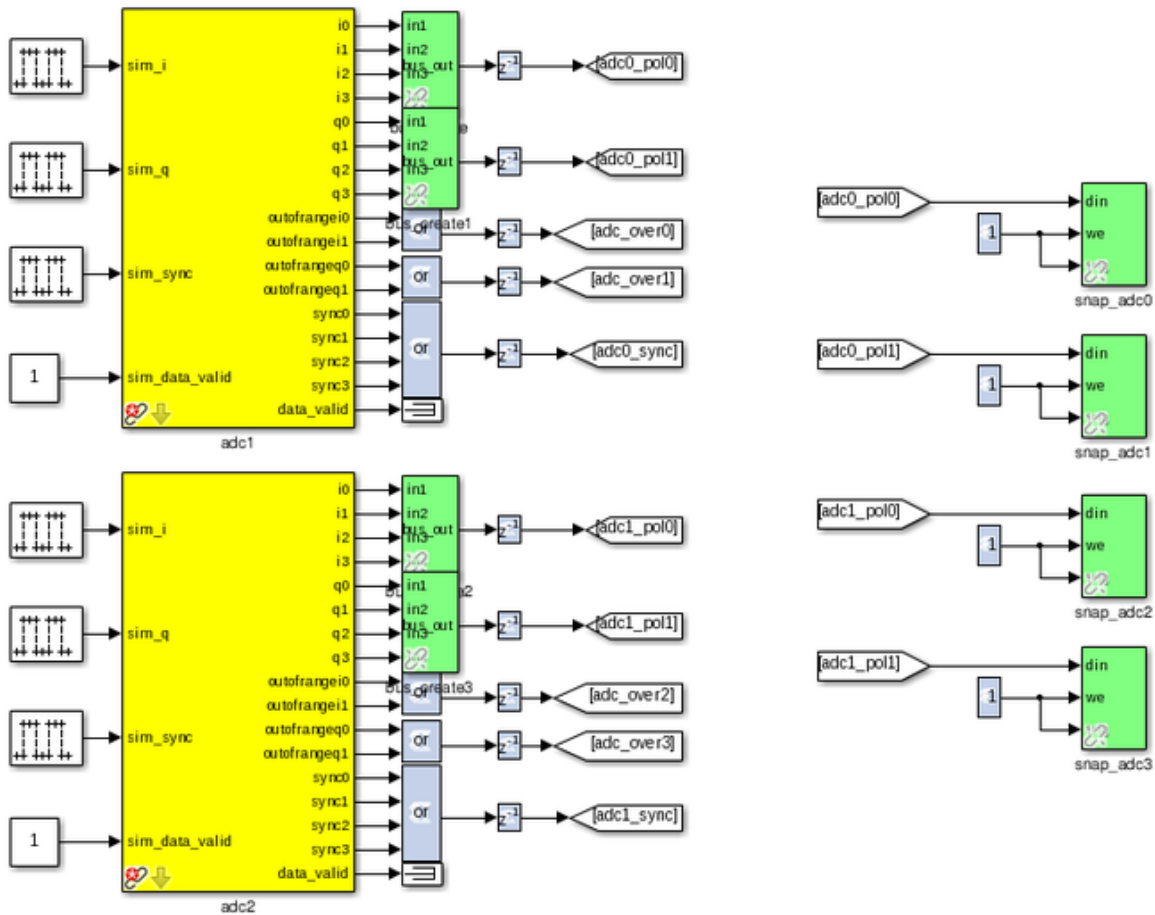
Sync Generator



The Sync Generator puts out a sync pulse which is used to synchronize the blocks in the design. See the CASPER memo on sync pulse generation for a detailed explanation.

This sync generator is able to synchronize with an external trigger input. Typically we connect this to a GPS's 1pps output to allow the system to reset on a second boundary after a software arm and thus know precisely the time at which an accumulation was started. To do this you can input the 1pps signal into either ADCs' sync input. The sync pulse allows data to be tagged with a precise timestamp. It also allows multiple ROACH boards to be synchronized, which is useful if large numbers of antenna signals are being correlated.

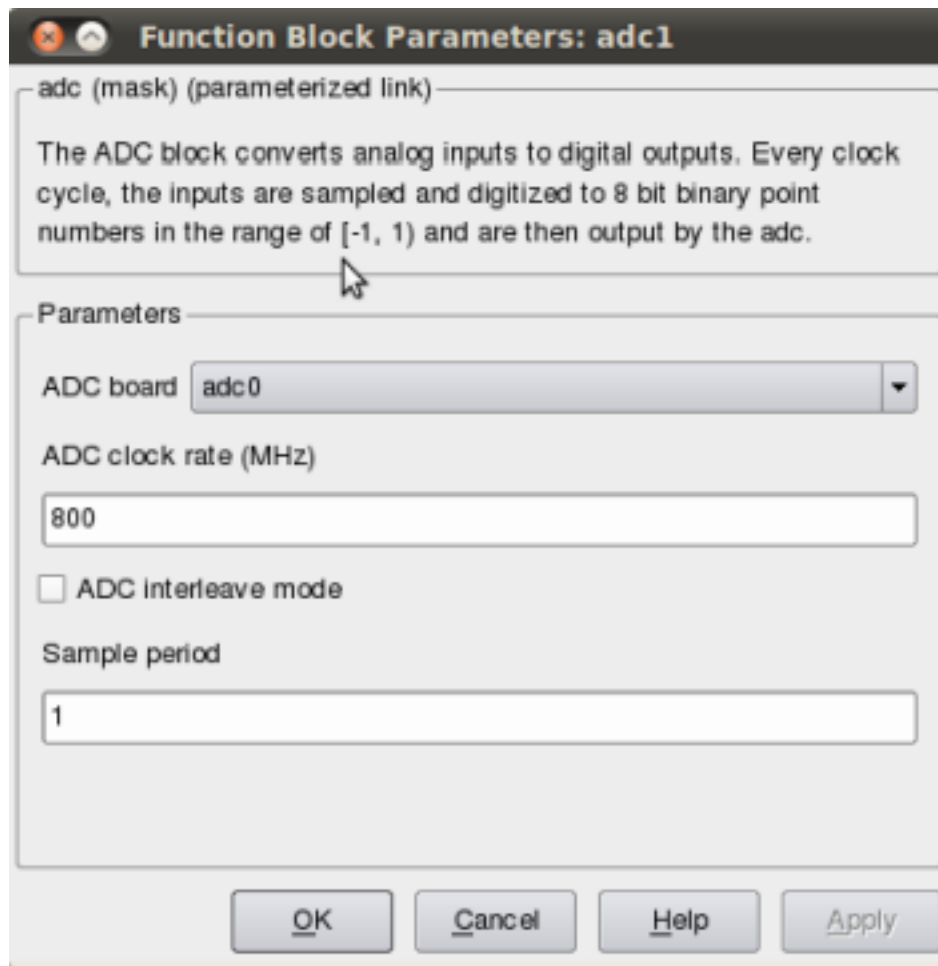
ADCs



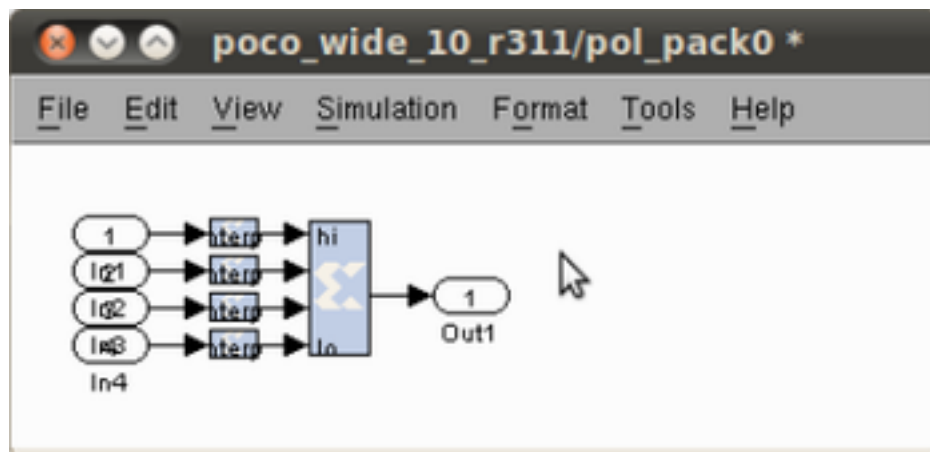
Connection of the ADCs is as in tutorial 3 except for the sync outputs. Here we OR all four outputs together to produce a sync pulse sampled at one quarter the rate of the ADC's sample clock. This is the simplest way of dealing with the four sync samples supplied to the FPGA on every clock cycle, but means that our system can only be synchronized to within 3 ADC sample clocks.

Logic is also provided to generate a sync manually via a software input. This allows the design to be used even in the absence of a 1 pps signal. However, in this case, the time the sync pulse occurs depends on the latency of software issuing the sync command and the FPGA signal triggering. This introduces some uncertainty in the timestamps associated with the correlator outputs. We will not use the 1pps in this tutorial although the design has the facility to do this hardware sync'ing.

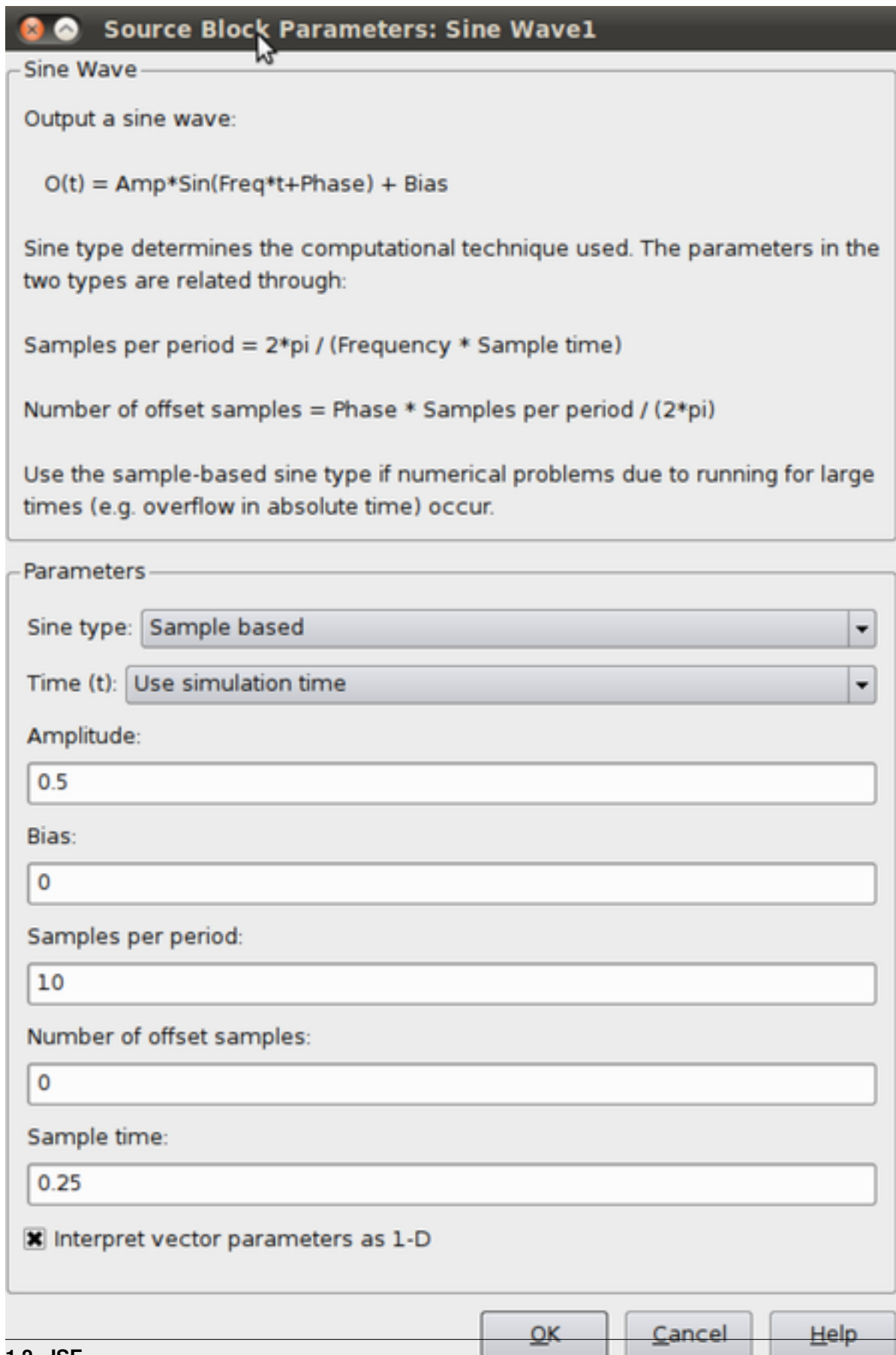
Set up the ADCs as follows and change the second ADC board's mask parameter to `adc1...`



Throughout this design, we use CASPER's `bus_create` and `bus_expand` blocks to simplify routing and make the design easier to follow.



For the purposes of simulation, it can be useful to put simulation input signals into the ADCs. These blocks are pulse generators in the case of sync inputs and any analogue source for the RF inputs (noise, CW tones etc).



Source Block Parameters: Sine Wave1

Sine Wave

Output a sine wave:

$$O(t) = \text{Amp} * \sin(\text{Freq} * t + \text{Phase}) + \text{Bias}$$

Sine type determines the computational technique used. The parameters in the two types are related through:

$$\text{Samples per period} = 2 * \pi / (\text{Frequency} * \text{Sample time})$$
$$\text{Number of offset samples} = \text{Phase} * \text{Samples per period} / (2 * \pi)$$

Use the sample-based sine type if numerical problems due to running for large times (e.g. overflow in absolute time) occur.

Parameters

Sine type:

Time (t):

Amplitude:

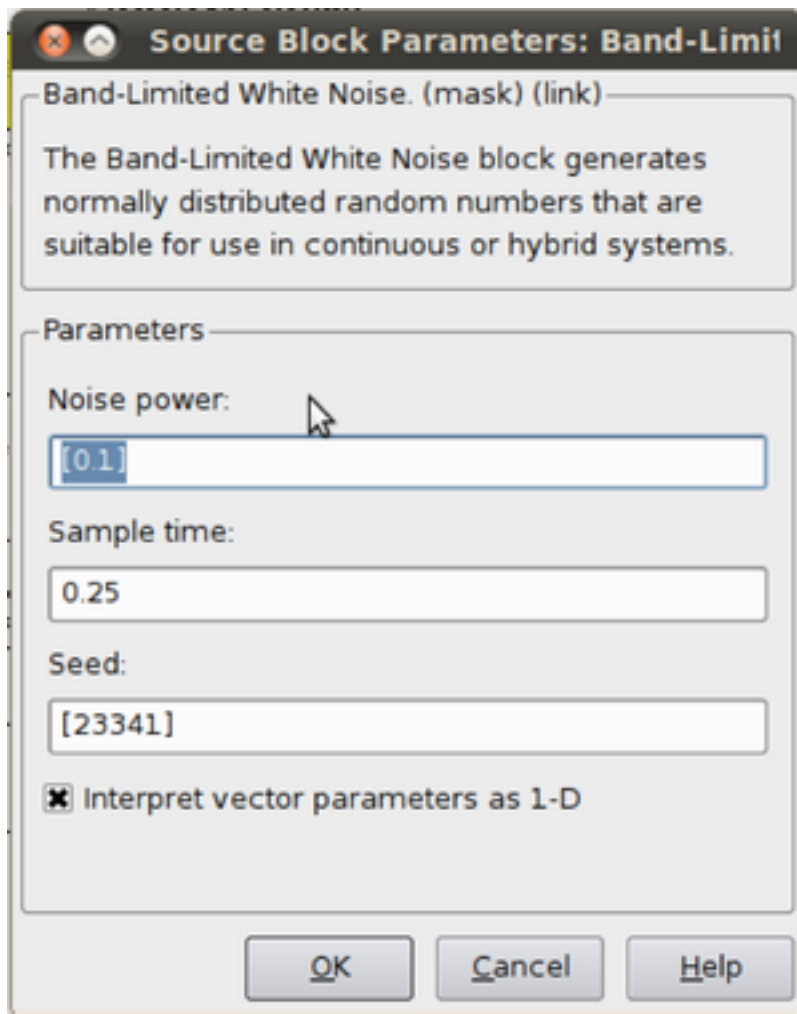
Bias:

Samples per period:

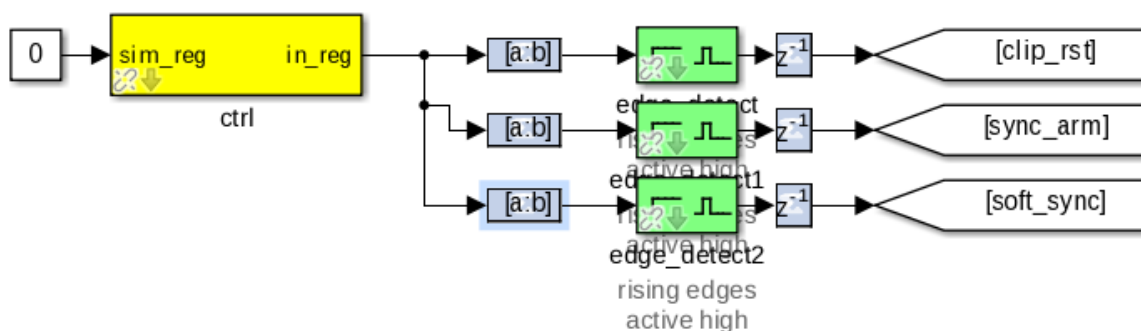
Number of offset samples:

Sample time:

☒ Interpret vector parameters as 1-D

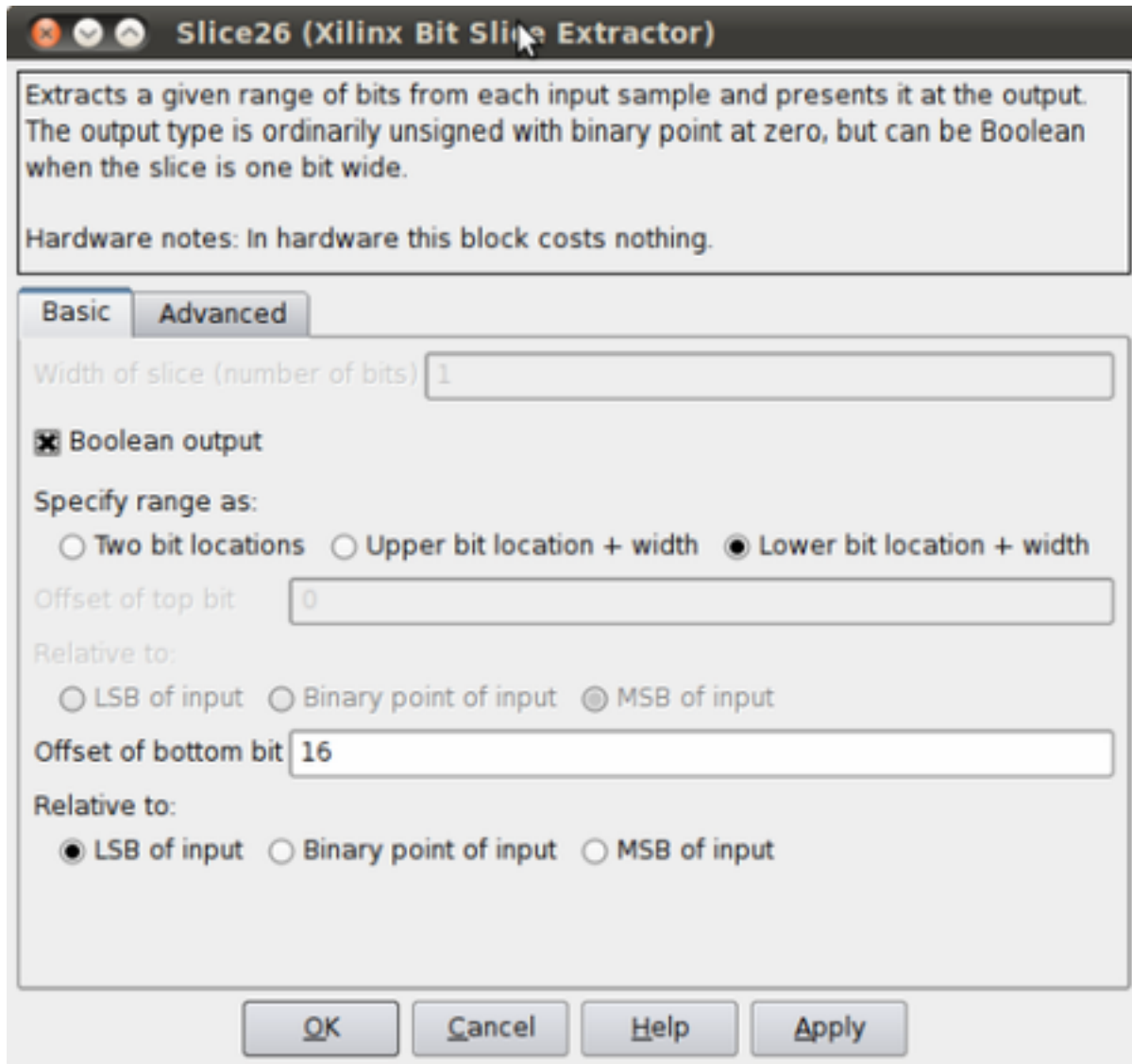


Control Register



This part of the Simulink design sets up a software register which can be configured in software to control the correlator. Set the yellow software register's IO direction as from processor. You can find it in the CASPER_XPS System blockset. The constant block input to this register is used only for simulation.

The output of the software register goes to three slice blocks, which will pull out the individual parameters for use with configuration. The first slice block (top) is setup as follows:



The slice block can be found under the Xilinx Blockset → Control Logic. The only change with the subsequent slice blocks is the Offset of the bottom bit. They are, from top to bottom, respectively, 16, 17 & 18.

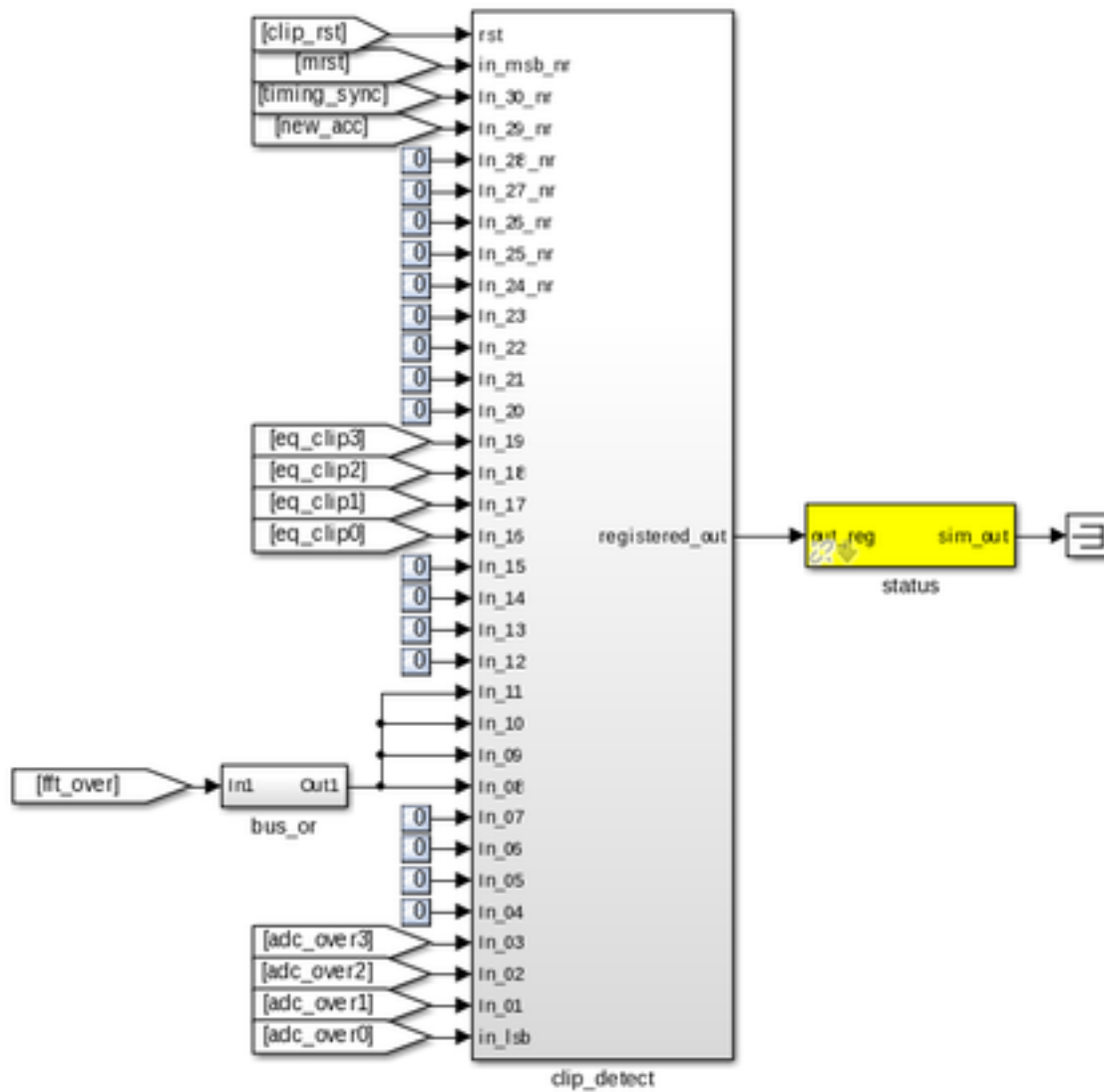
After each slice block we put an edge_detect block, this outputs true if a boolean input signal is true this clock and was false last clock. Found under CASPER DSP Blockset → Misc.

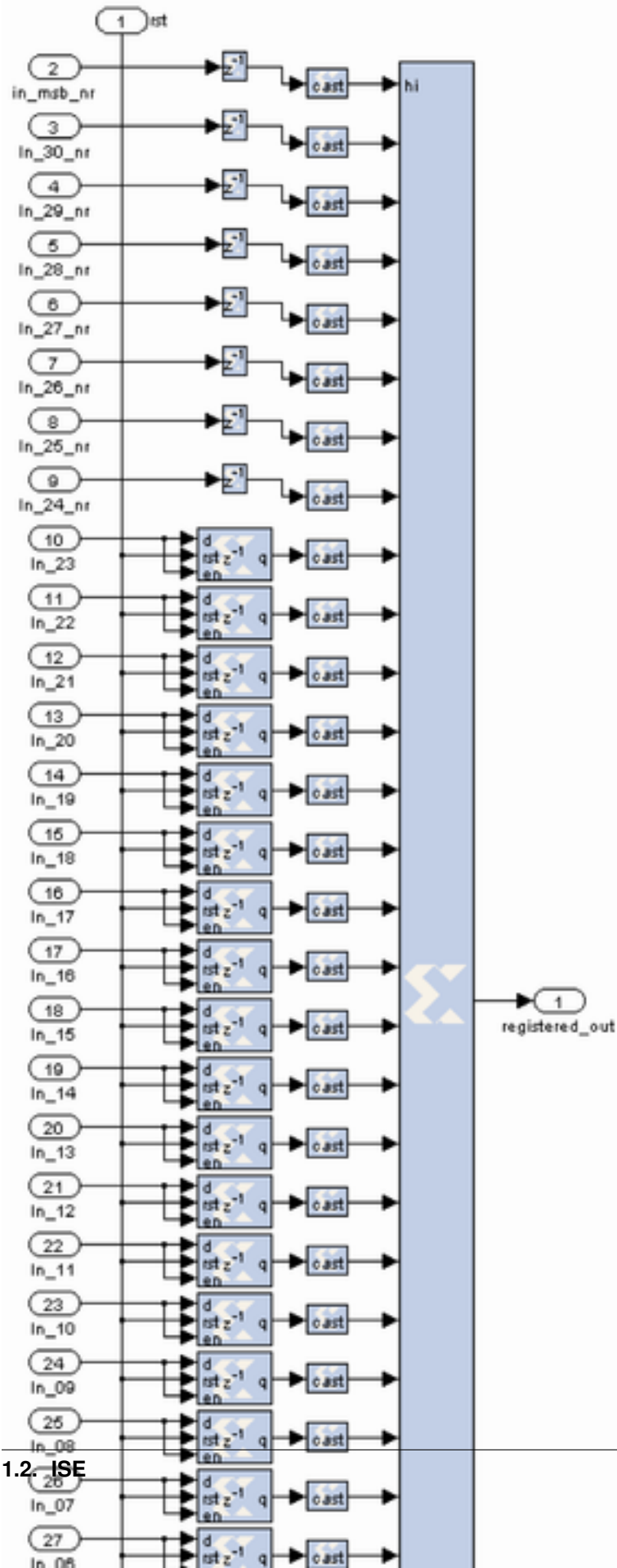
Next are the delay blocks. They can be left with their default settings and can be found under Xilinx Blockset → Common.

The Goto and From blocks can be found under Simulink → Signal Routing. Label them as in the block diagram above.

Clip Detect and status reporting

To detect and report signal saturation (clipping) to software, we will create a subsystem with latching inputs.





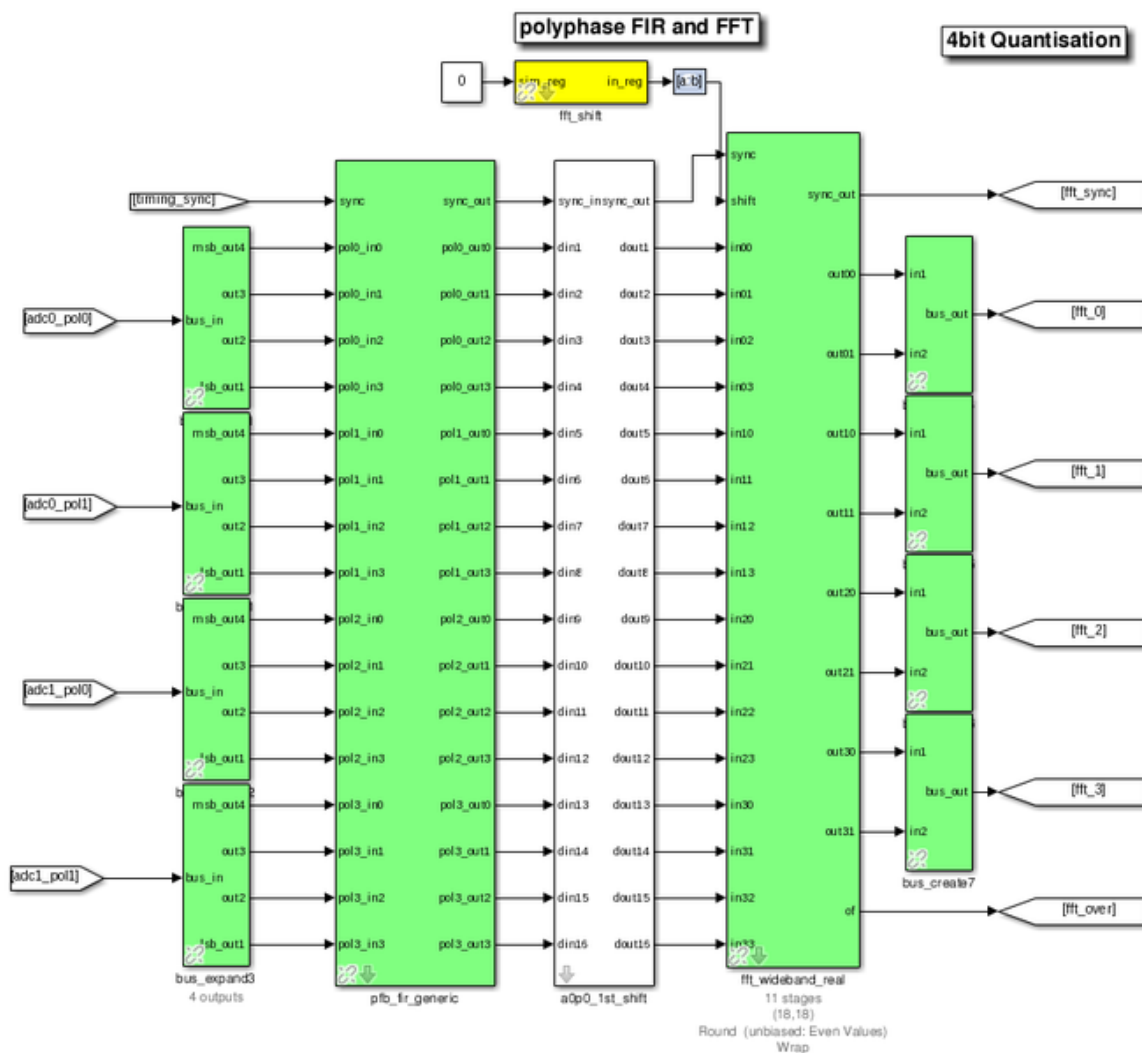
The internals of this subsystem (right) consist of delay blocks, registers and cast blocks.

The delays (inputs 2 - 9) can be kept as default. Cast blocks are required as only unsigned integers can be concatenated. Set their parameters to Unsigned, 1 bit, 0 binary points Truncated Quantization, Wrapped Overflow and 0 Latency.

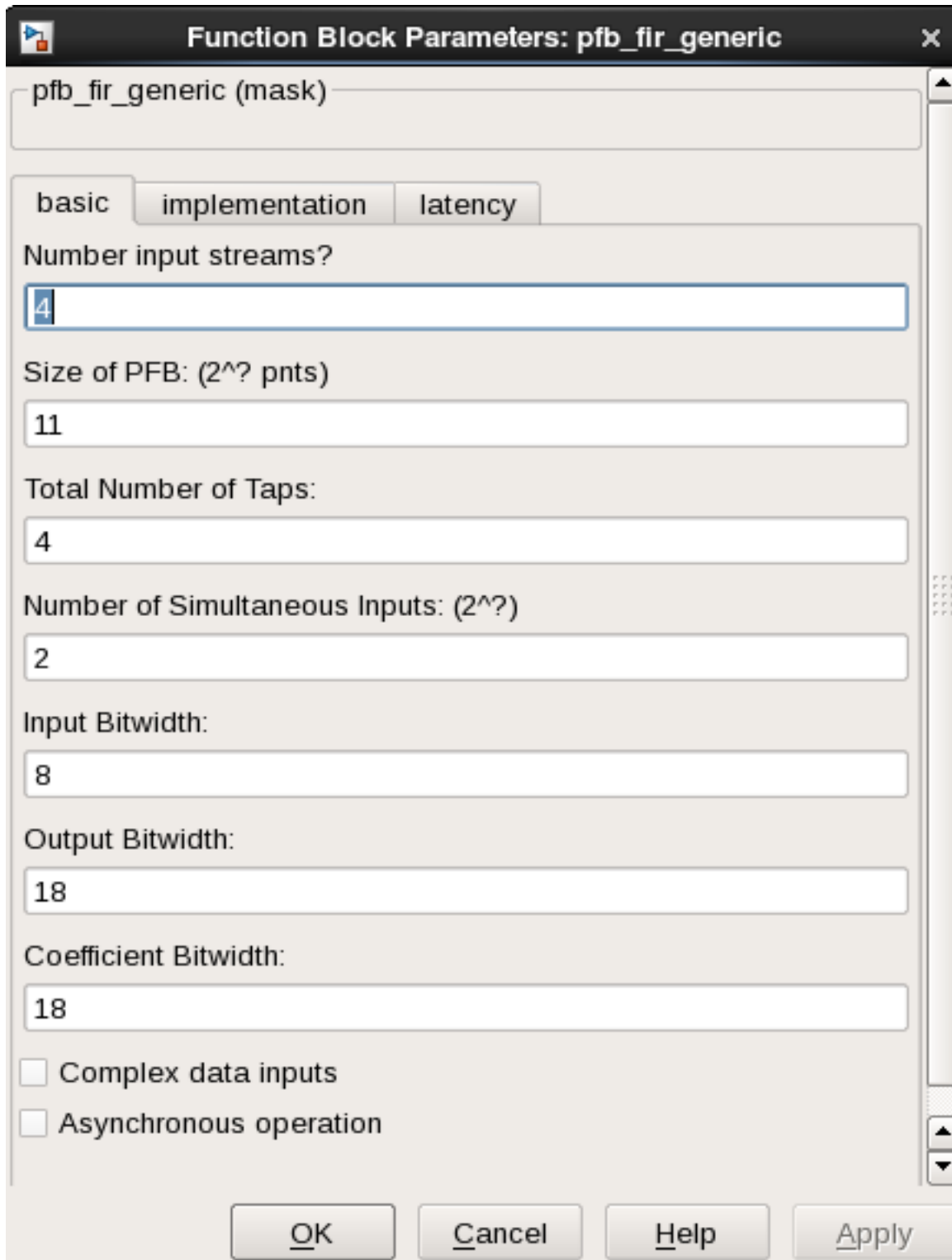
The Registers (inputs 10 - 33) must be set up with an initial value of 0 and with enable and reset ports enabled. The status register on the output of the clip detect is set to processor in with unsigned data type and 0 binary point with a sample period of 1.

PFBs, FFTs and Quantisers

The PFB FIR, FFT and the Quantizer are the heart of this design, there is one set of each for the 4 outputs of the ADCs. However, in order to save resources associated with control logic and PFB and FFT coefficient storage, the four independent filters are combined into a single simlink block. This is configured to process four independent data streams by setting the “number of inputs” parameter on the PFB_FIR and FFT blocks to 4.



Configure the PFB_FIR_generic blocks as shown below:



The image shows a dialog box titled "Function Block Parameters: pfb_fir_generic". It has a tabbed interface with three tabs: "basic", "implementation", and "latency". The "basic" tab is selected. The dialog contains several input fields and two checkboxes. The inputs are: "Number input streams?" with a value of 4, "Size of PFB: (2^? pnts)" with a value of 11, "Total Number of Taps:" with a value of 4, "Number of Simultaneous Inputs: (2^?)" with a value of 2, "Input Bitwidth:" with a value of 8, "Output Bitwidth:" with a value of 18, and "Coefficient Bitwidth:" with a value of 18. The checkboxes are "Complex data inputs" and "Asynchronous operation", both of which are unchecked. At the bottom of the dialog are four buttons: "OK", "Cancel", "Help", and "Apply".

Function Block Parameters: pfb_fir_generic

pfb_fir_generic (mask)

basic implementation latency

Number input streams?

4

Size of PFB: (2^? pnts)

11

Total Number of Taps:

4

Number of Simultaneous Inputs: (2^?)

2

Input Bitwidth:

8

Output Bitwidth:

18

Coefficient Bitwidth:

18

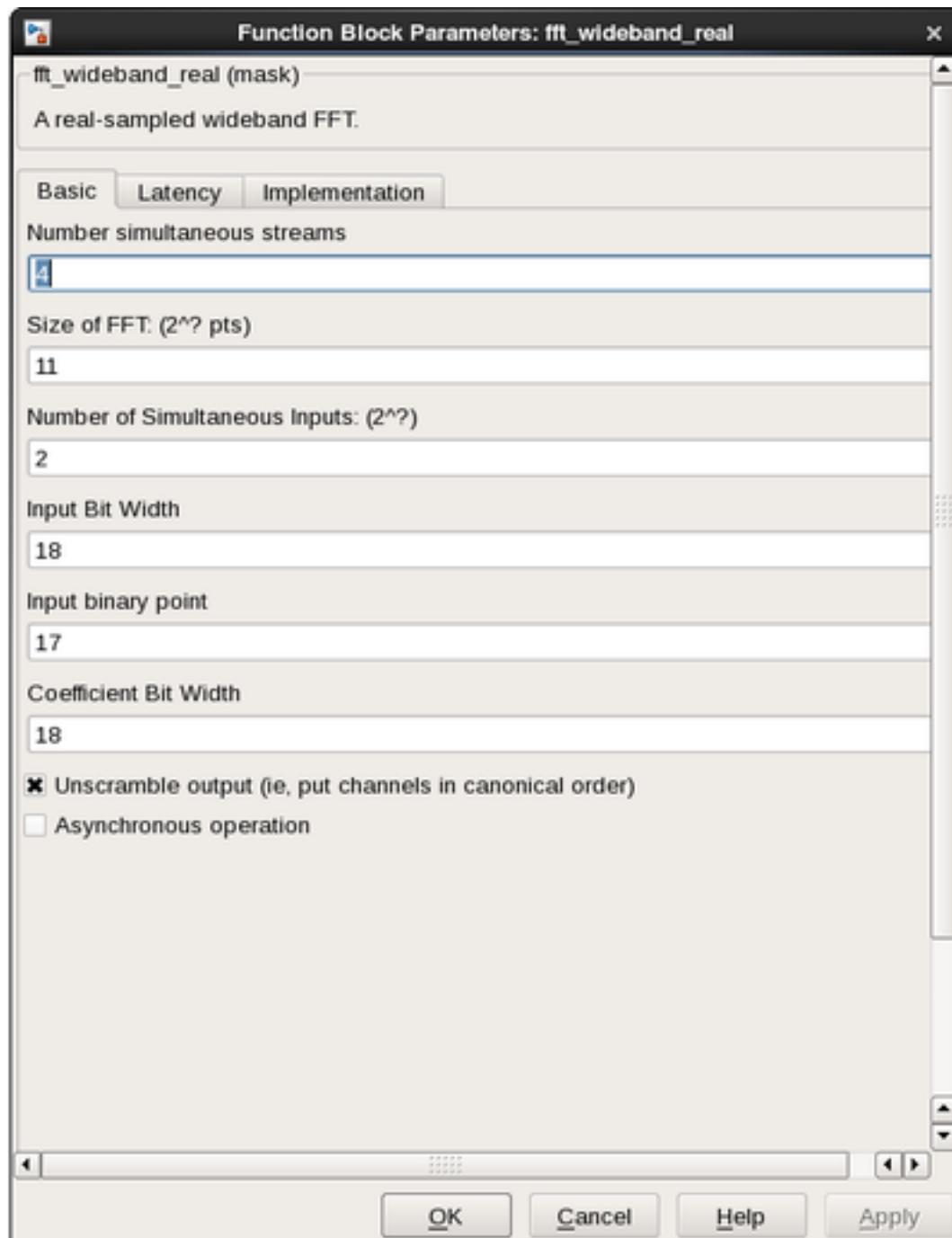
☐ Complex data inputs

☐ Asynchronous operation

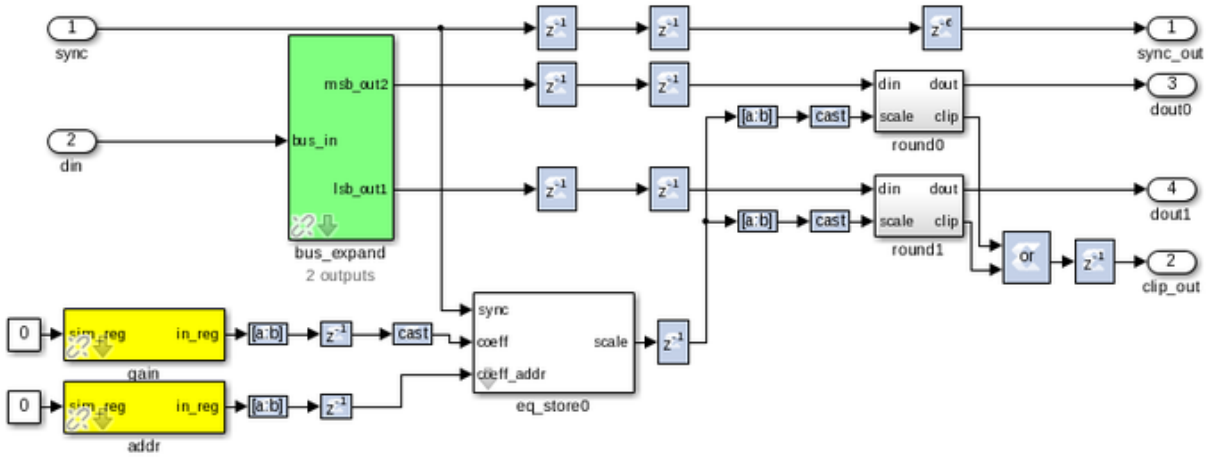
OK Cancel Help Apply

There is potential to overflow the first FFT stage if the input is periodic or signal levels are high as shifting inside the FFT is only performed after each butterfly stage calculation. For this reason, we recommend casting any inputs up to 18 bits with the binary point at position 17 (thus keeping the range of values -1 to 1), and then ownshifting by 1 bit to place the signal in one less than the most significant bits.

The `fft_wideband_real` block should be configured as follows:

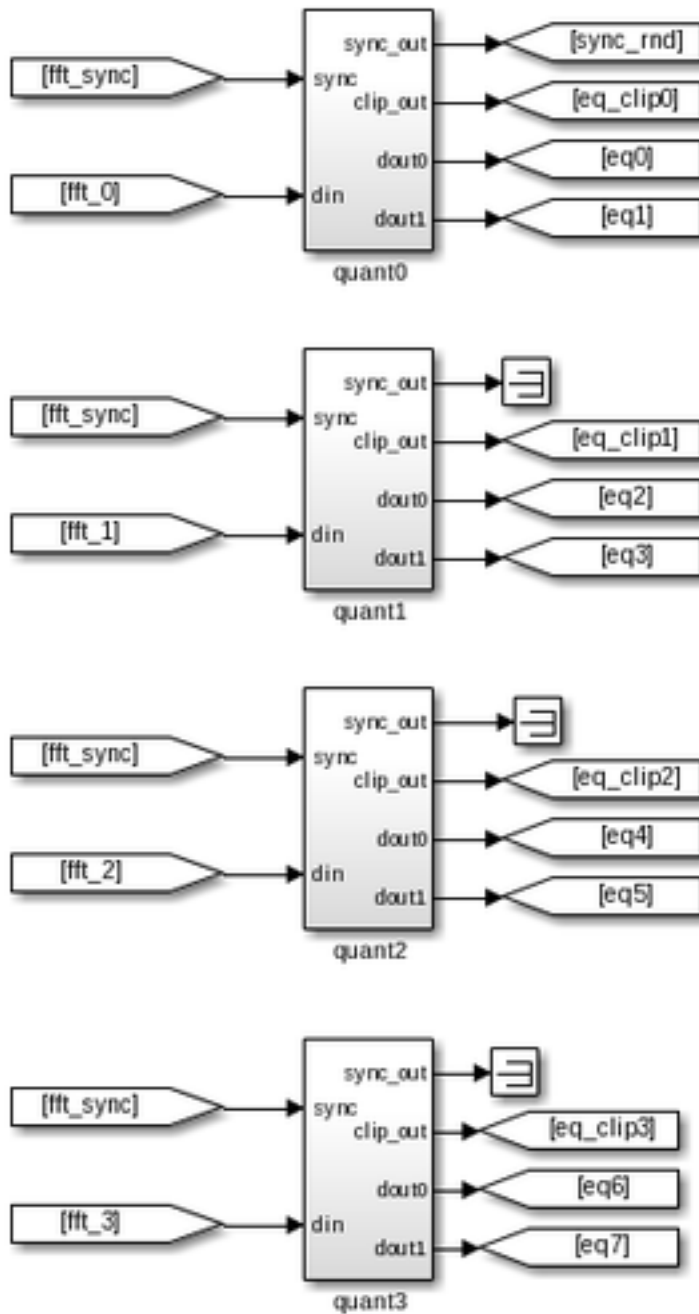


The Quantizer Subsystem is designed as seen below. The quantizer removes the bit growth that was introduced in the PFB and FFT. We can do this because we do not need the full dynamic range.



The top level view of the Quantizer Subsystem is as seen below.

4bit Quantisation



LEDs

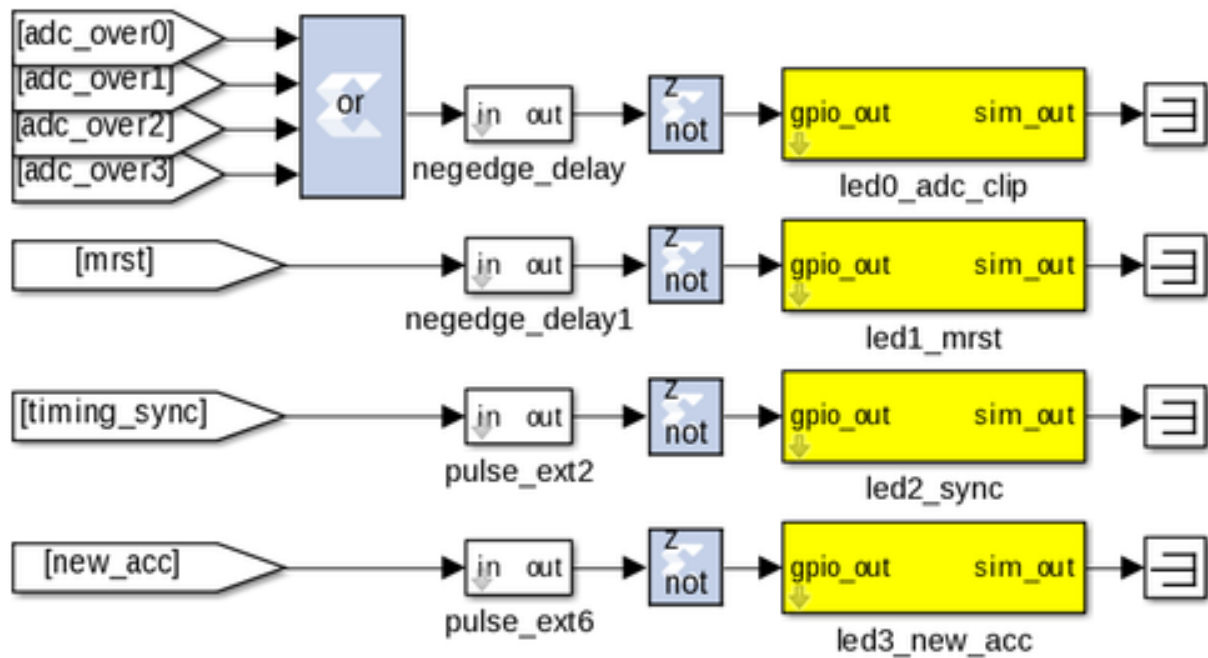
The following sections are more periphery to the design and will only be touched on. By now you should be comfortable putting the blocks together and be able to figure out many of the values and parameters. Also feel free to consult the reference design which sits in the tutorial 4 project directory or ask any questions of the tutorial helpers.

As a debug and monitoring output we can wire up the LEDs to certain signals. We light an LED with every sync pulse. This is a sort of heartbeat showing that the design is clocking and the FPGA is running.

We light an error LED in case any ADC overflows and another if the system is reset. The fourth LED gives a visual indication of when an accumulation is complete.

ROACH's LEDs are negative logic, so when the input to the yellow block is high, the LED is off. Since this is the opposite of what you'd normally expect, we invert the logic signals with a NOT gate.

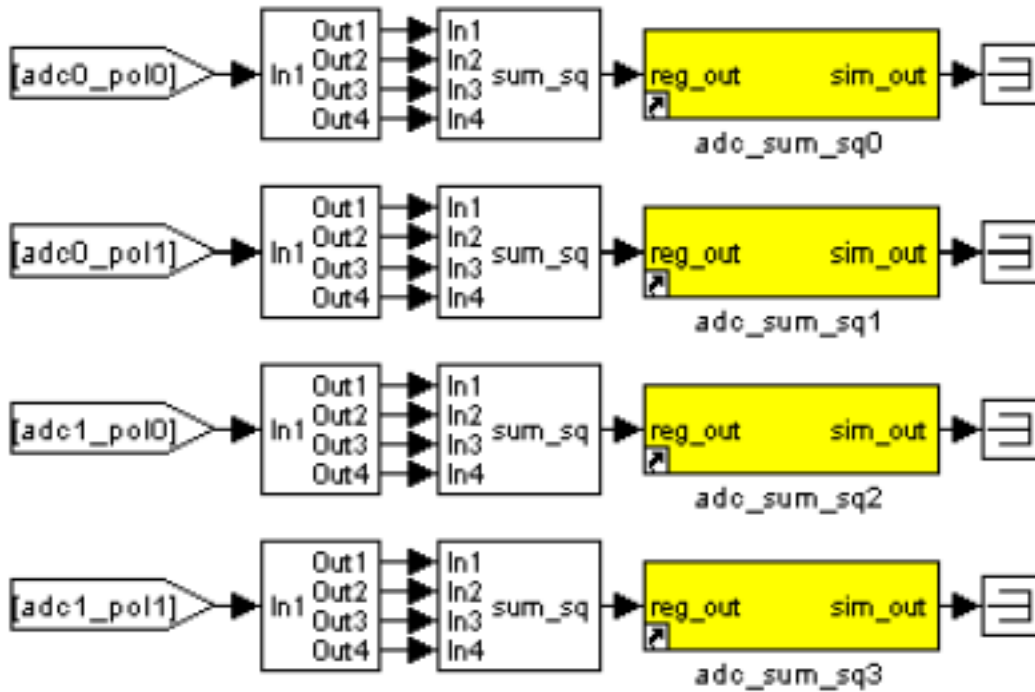
Since the signals might be too short to light up an LED and for us to actually see it (consider the case where a single ADC sample overflows; $1/800\text{MHz}$ is 1.25 nS – much too short for the human eye to see) we add a negedge delay block which delays the negative edge of a block, thereby extending the positive pulse. A length of 2^{23} gives about a 10ms pulse.



Recall that ROACH's LEDs are inverted. Add "not" blocks so things make sense.

ADC RMS

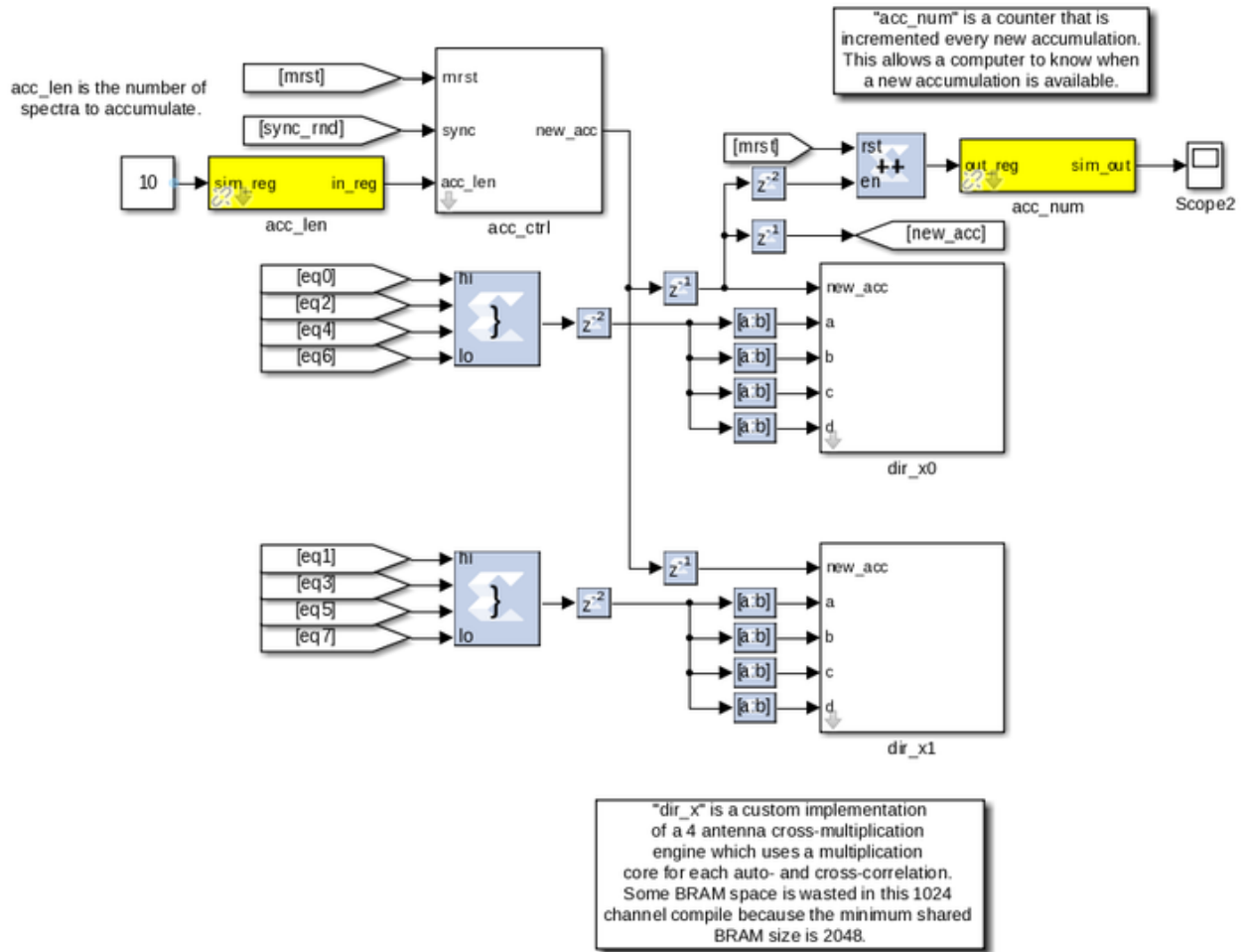
These blocks calculate the RMS values of the ADCs' input signals. We subsample the input stream by a factor of four and do a pseudo random selection of the parallel inputs to prevent false reporting of repetitive signals. This subsampled stream is squared and accumulated for 2^{16} samples.



The MAC operation

The multiply and accumulate is performed in the `dir_x` (direct-x) blocks, so named because different antenna signal pairs are multiplied directly, in parallel (as opposed to the packetized correlators' X engines which process serially).

Two sets are used, one for the even channels and another for the odd channels. Accumulation for each antenna pair takes place in BRAM using the same simple vector accumulator used in `tut3`.



CONTROL:

The design starts by itself when the FPGA is programmed. The only control register inputs are for resetting counters and optionally sync'ing to external signal.

Sync LED provides a "heartbeat" signal to instantly see if your design is clocked sensibly.

New accumulation LED gives a visual indication of data rates and dump times.

Accumulation counter provides simple mechanism for checking if a new spectrum output is available. (poll and compare to last value)

Software

The python scripts are located in the tut4 tutorial directory. We first need to run `poco_init.py` to program the FPGA and configure the design. Then we can run either the auto or the cross correlations plotting scripts (`plot_poco_auto.py` and `plot_poco_cross.py`).

`poco_init.py`

```
print('Connecting to server %s on port %i...'%(roach,katcp_port)),
      fpga = corr.katcp_wrapper.FpgaClient(roach, katcp_port,
      timeout=10, logger=logger)
```

(continues on next page)

(continued from previous page)

```

time.sleep(1)
if fpga.is_connected():
    print 'ok\n'
else:
    print 'ERROR connecting to server %s on port %i.\n'%
(roach,katcp_port)
    exit_fail()
    print '-----'
    print 'Programming FPGA...',
    if not opts.skip:
        fpga.progdev(boffile)
        print 'done'
    else:
        print 'Skipped.'
    print 'Configuring fft_shift...',
    fpga.write_int('fft_shift', (2**32)-1)
    print 'done'
    print 'Configuring accumulation period...',
    fpga.write_int('acc_len', opts.acc_len)
    print 'done'
    print 'Resetting board, software triggering and resetting error
counters...',
    fpga.write_int('ctrl', 1<<17) #arm
    fpga.write_int('ctrl', 1<<18) #software trigger
    fpga.write_int('ctrl', 0)
    fpga.write_int('ctrl', 1<<18) #issue a second trigger
    print 'done'

```

In previous tutorials you will probably have seen very similar code to the code above. This initiates the katcp wrapper named fpga which manages the interface between the software and the hardware. fpga.progdev programs the boffile onto the FPGA and fpga.write_int writes to a register.

poco_adc_amplitudes.py

This script outputs in the amplitudes (or power) of each signal as well as the bits used. It updates itself ever second or so.

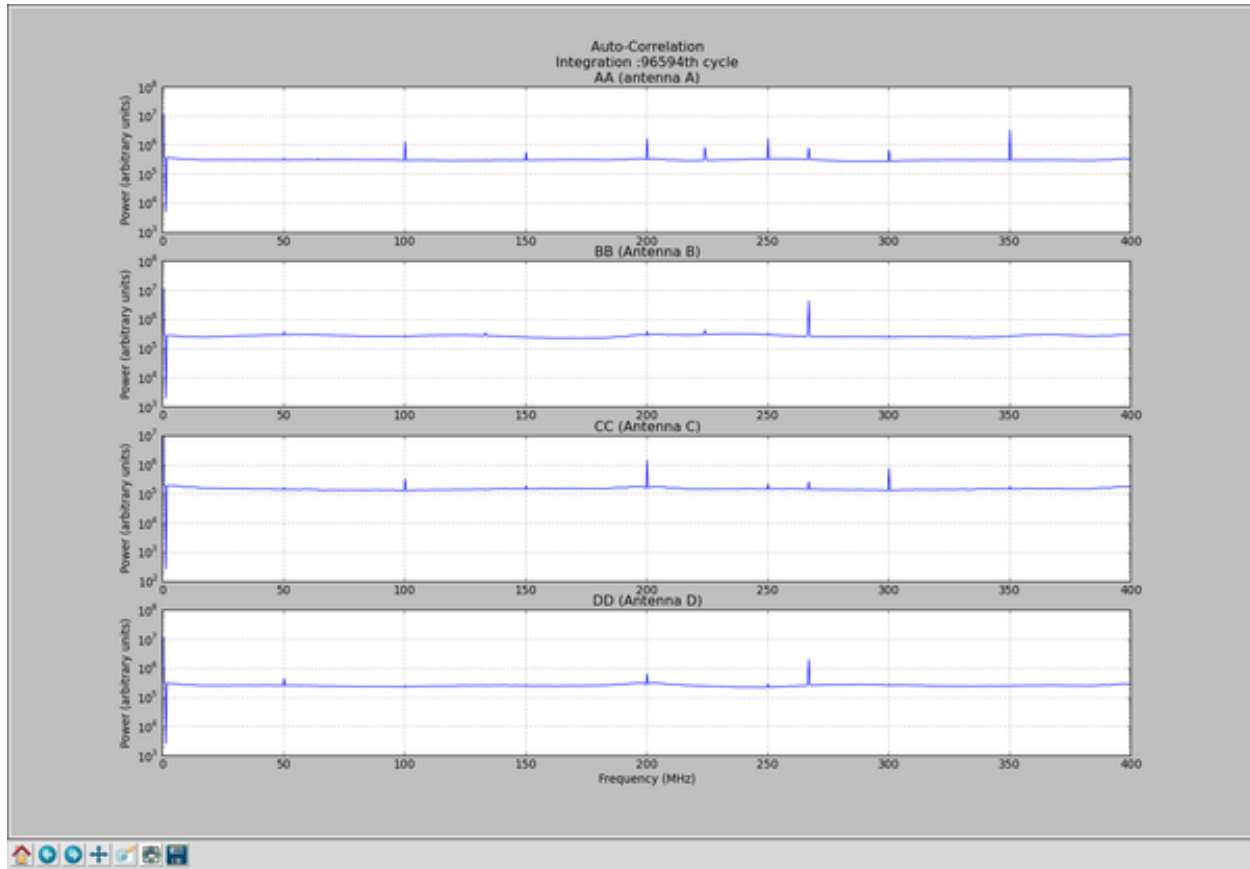
```

ADC amplitudes
-----
ADC0 input I: 0.006 (0.51 bits used)
ADC0 input Q: 0.004 (0.19 bits used)
ADC1 input I: 0.005 (0.45 bits used)
ADC1 input Q: 0.004 (0.19 bits used)
-----

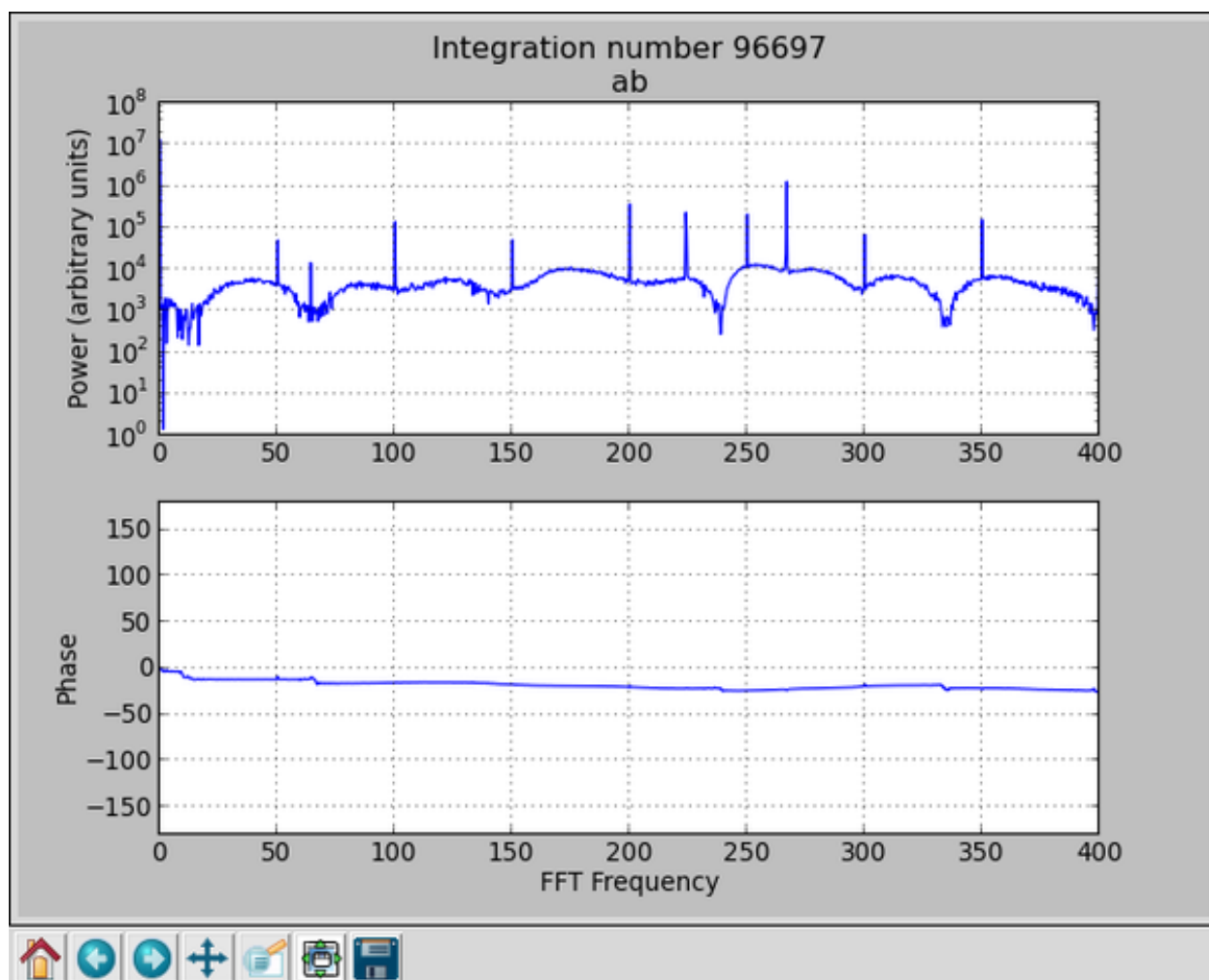
```

poco_plot_auto.py

This script grabs auto-correlations from the brams and plots them. Since there are 4 inputs, 2 for each ADC there are 4 plots. Some plots will be random if there is no noise source or tone being inputted into ADC. Ie plots 3 and 4.



poco_plot_cross.py This script grabs cross-correlations from the brams and plots them. This plot shows the cross-correlation of AB.



2.1 OS

It is recommended to use Ubuntu 14.04. 16.04 has also been known to work, although the setup process can be a bit of a headache. Ubuntu 16.04 LTS will work with SKARAB and Red Pitaya.

2.2 Matlab and Xilinx

To use the tutorials you will need to install the versions of Matlab and the Xilinx tools particular to the hardware you plan to use. See the installation matrix below.

Hardware	Matlab Version	Xilinx Version
ROACH1/2	2013b	ISE 14.7
SKARAB	2018a	Vivado 2018.2
SNAP	2016b	Vivado 2016.4
Red Pitaya	2018a	Vivado 2018.2

2.3 Modifications to be run after installs

ROACH1/2

Xilinx removed support for several hardware pcores we use for ROACH1/2 from ISE 14. So the current solution is to add the following pcores from the Xilinx 11 install to your *XPS_ROACH_BASE/pcores* folder or to your 14 install directory at *Xilinx/14.7/ISE_DS/EDK/hw/XilinxProcessorIPLib/pcore*.

OPB pcores

- bram_if_cntlr_v1_00_a
- bram_if_cntlr_v1_00_b

- ipif_common_v1_00_c
- opb_arbiter_v1_02_e
- opb_bram_if_cntlr_v1_00_a
- opb_ipif_v3_00_a
- opb_opb_lite_v1_00_a
- opb_v20_v1_10_c
- proc_common_v1_00_a

All installs

The syntax in the Xilinx Perl scripts is not supported under the Ubuntu default shell Dash. Change the symbolic link sh -> dash to sh -> bash:

```
cd /bin/  
sudo rm sh  
sudo ln -s bash sh
```

Point gmake to make by creating the symbolic link gmake -> make:

```
cd /usr/bin/  
sudo ln -s make gmake
```

If you are not getting any blocks in Simulink (Only seen in CentOS) change the permissions on /tmp/LibraryBrowser to a+rwX:

```
chmod a+rwX /tmp/LibraryBrowser
```