2010 Workshop

Tutorial 1: Introduction to Simulink

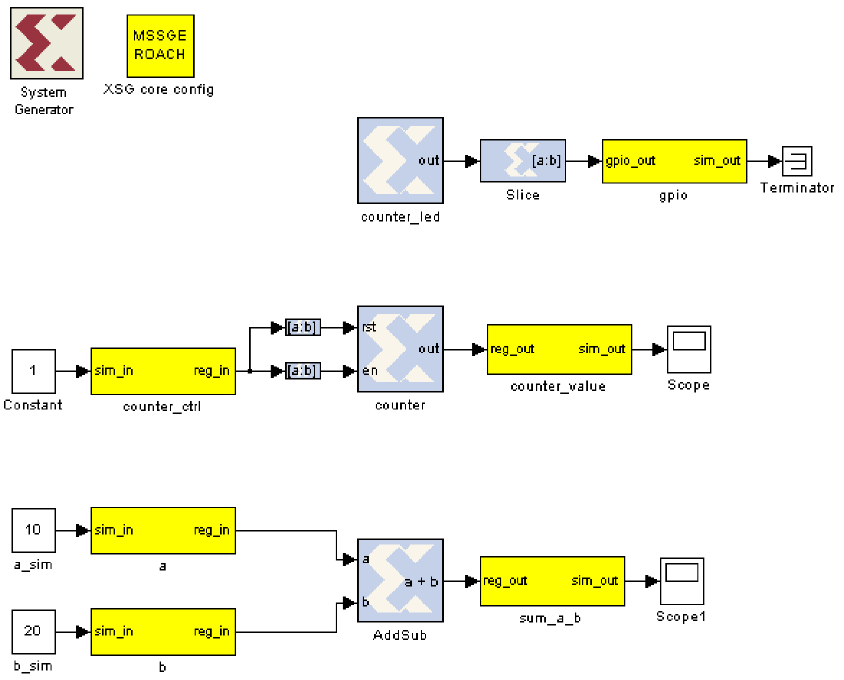
Author: M. Wagner, J. Manley and W. New

Expected completion time: 2hrs

SKA SOUTH AFRICA
SQUARE KILOMETRE ARRAY

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

Contents:



CASPER

1 Introduction

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

2 Setup

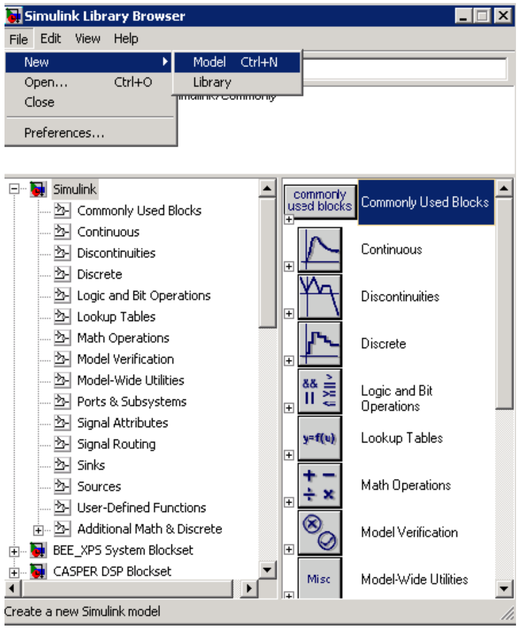
The lab at the workshop is preconfigured with the CASPER libraries, Matlab and Xilinx tools.

3 Creating Your Design

3.1 Create a New Model:

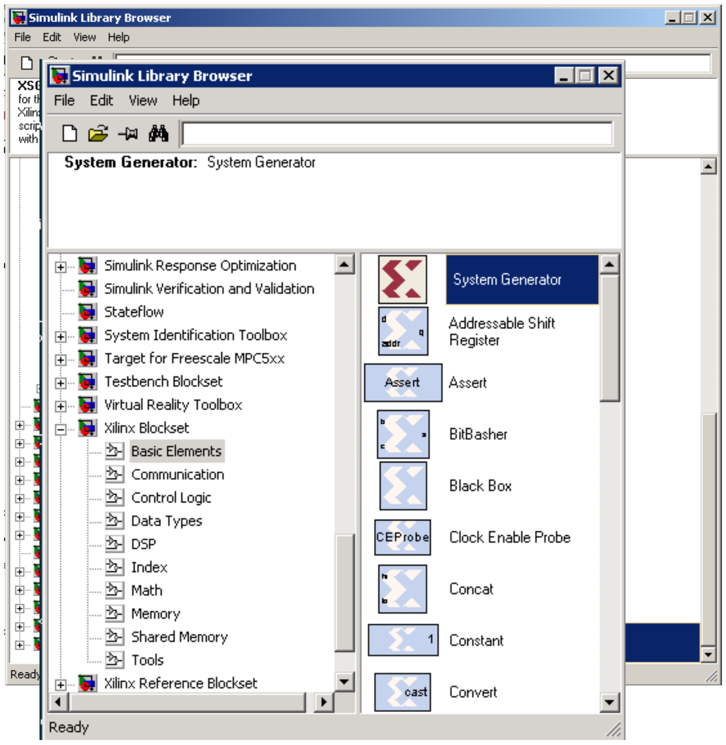
Start Matlab and open Simulink (either by typing *simulink* on the Matlab command line, or by click in the Simulink icon in the taskbar).

Create a new model:



3.2 Add Xilinx System Generator and XSG core config blocks:

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



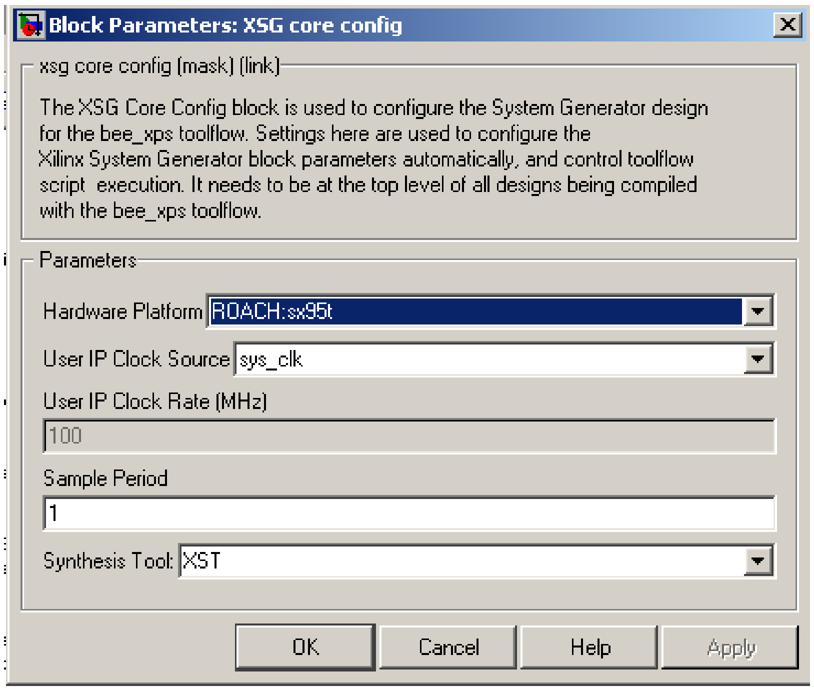
All hardware-related blocks are yellow and can be found in the *BEE_XPS* library. This library contains all the board-specific components colloquially called *Yellow Blocks.*

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

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

Leave everything else defaults and click *OK*.

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



xsg core config (mask) (link)

The XSG Core Config block is used to configure the System Generator design for the bee_xps toolflow. Settings here are used to configure the Xilinx System Generator block parameters automatically, and control toolflow script execution. It needs to be at the top level of all designs being compiled with the bee_xps toolflow.

Parameters

Hardware Platform: ROACH:sx95t

User IP Clock Source: sys_clk

User IP Clock Rate (MHz): 100

Sample Period: 1

Synthesis Tool: XST

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

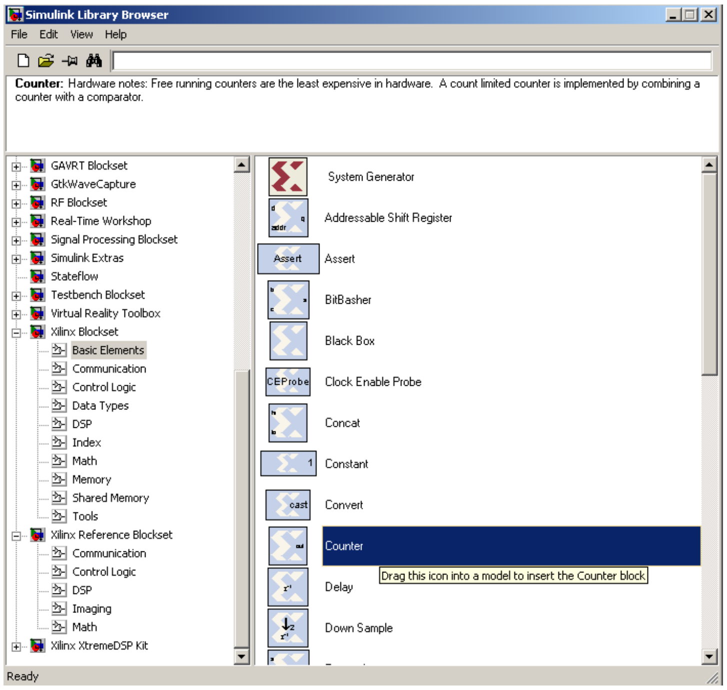
You need to add these two blocks for all CASPER designs.

3.3 Flashing LED

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

3.3.1 Add a counter

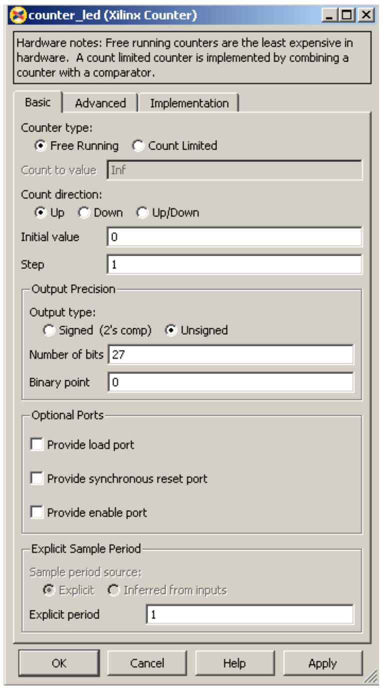
Add a counter to your design by navigating to *Xilinx Blockset -> Basic Elements -> Counter* and dragging it onto your model.



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.

Drag this icon into a model to insert the Counter block

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



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

Counter type: [✓ Free Running] Count Limited

Count to value: Inf

Count direction: [✓ Up] Down Up/Down

Initial value: 0

Step: 1

Output Precision

Output type: Signed (2's comp) [✓ Unsigned]

Number of bits: 27

Binary point: 0

Optional Ports

☐ Provide load port

☐ Provide synchronous reset port

☐ Provide enable port

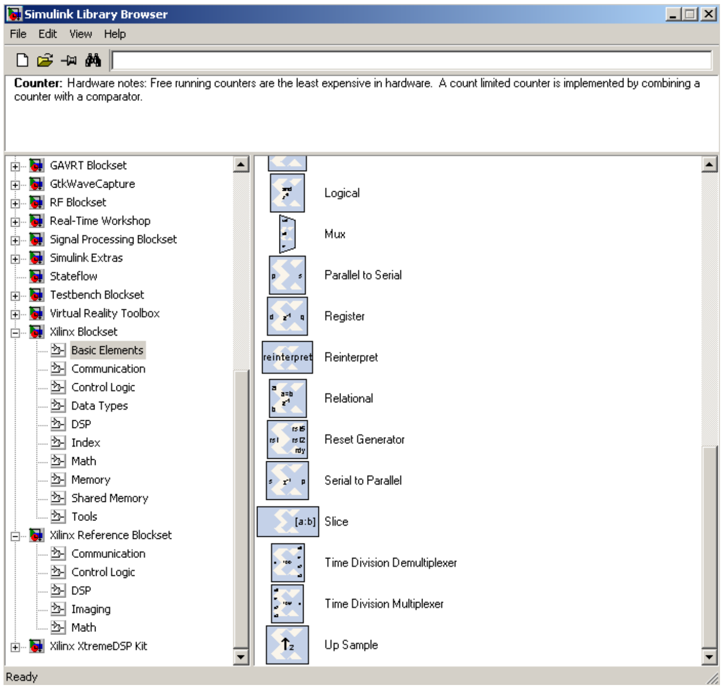
Explicit Sample Period

Sample period source: [✓ Explicit] Inferred from inputs

Explicit period: 1

3.3.2 Add a slice block to select out the msb

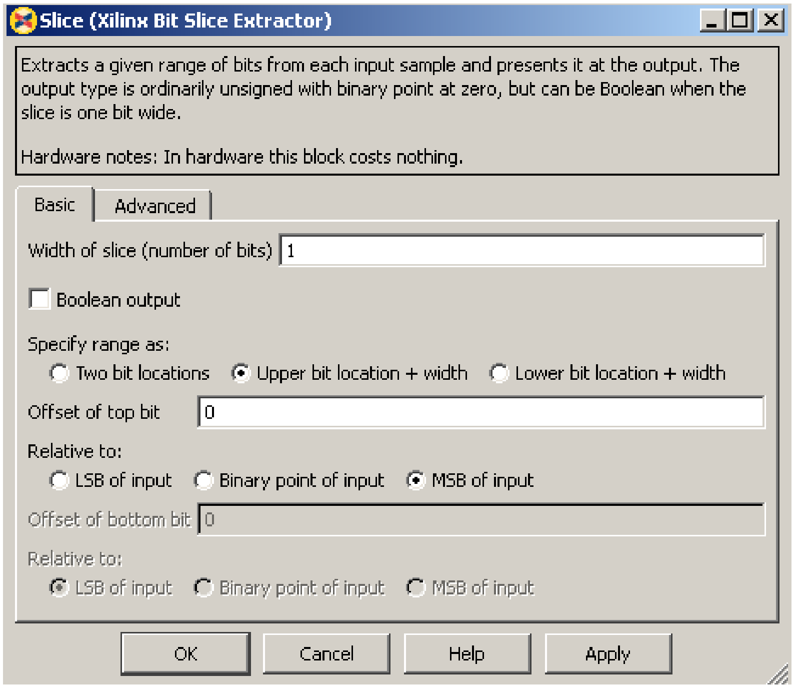
We now need to select the most significant bit of the counter. We do this using a slice block, which Xilinx provides. *Xilinx Blockset -> Basic Elements -> Slice*.


Simulink Library Browser

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.

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

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


Slice (Xilinx Bit Slice Extractor)

Extracts a given range of bits from each input sample and presents it at the output. The output type is ordinarily unsigned with binary point at zero, but can be Boolean when the slice is one bit wide.

Hardware notes: In hardware this block costs nothing.

Basic | Advanced

Width of slice (number of bits): 1

☐ Boolean output

Specify range as: Two bit locations [✓ Upper bit location + width] Lower bit location + width

Offset of top bit: 0

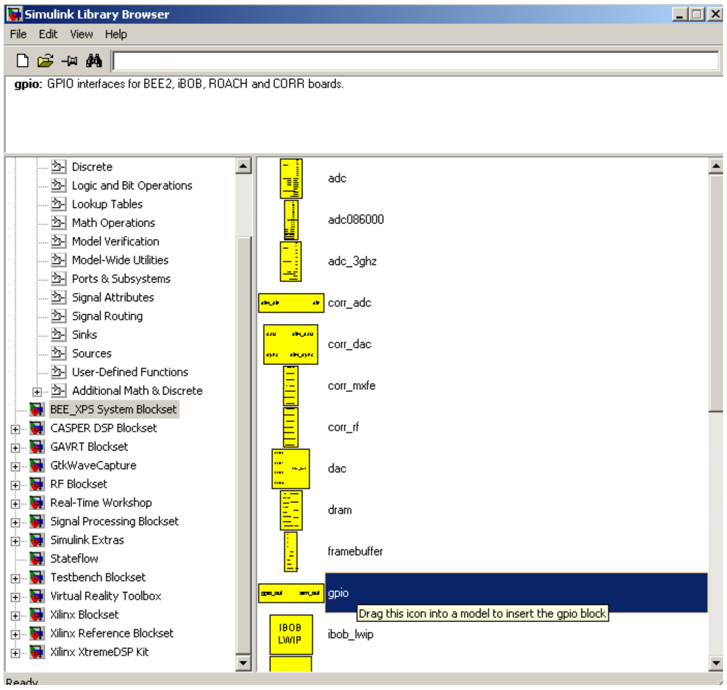
Relative to: LSB of input Binary point of input [✓ MSB of input]

Offset of bottom bit: 0

Relative to: [✓ LSB of input] Binary point of input MSB of input

3.3.3 Add a GPIO block

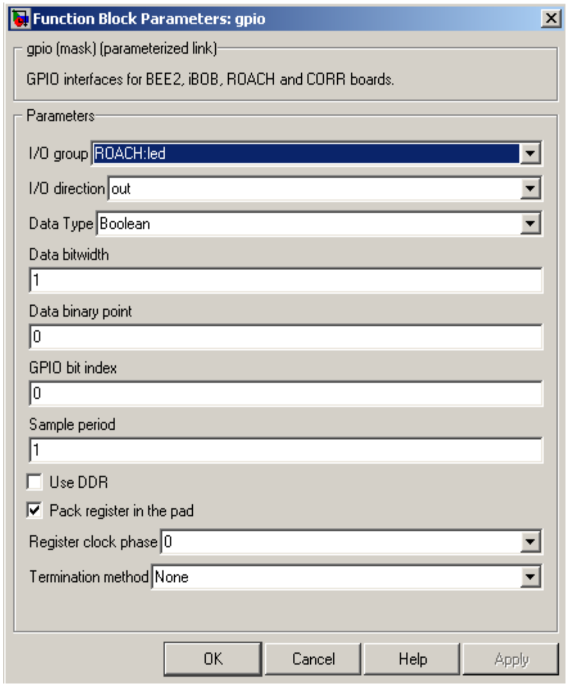
(BEE_XPS library -> gpio).



gpio: GPIO interfaces for BEE2, iBOB, ROACH and CORR boards.

Drag this icon into a model to insert the gpio block

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



Function Block Parameters: gpio

gpio (mask) (parameterized link)

GPIO interfaces for BEE2, iBOB, ROACH and CORR boards.

Parameters

I/O group: ROACH:led

I/O direction: out

Data Type: Boolean

Data bitwidth: 1

Data binary point: 0

GPIO bit index: 0

Sample period: 1

☐ Use DDR

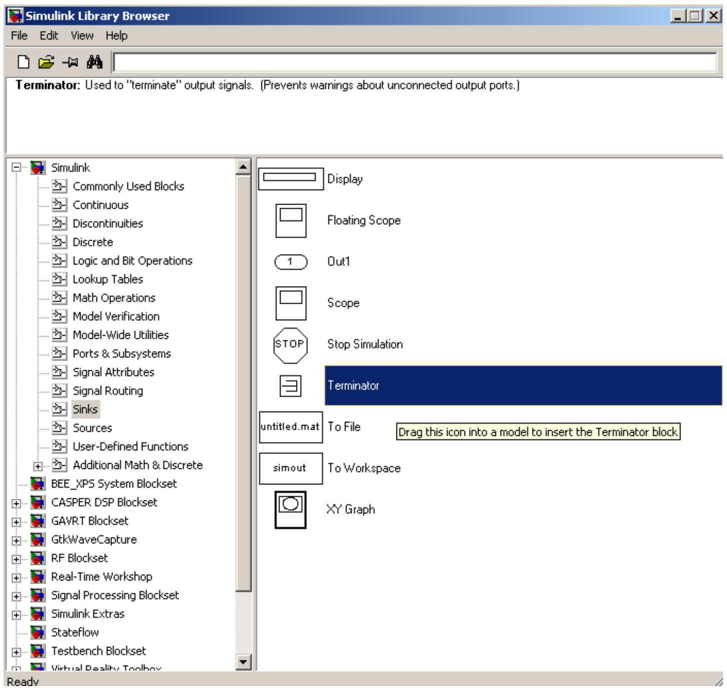
☑ Pack register in the pad

Register clock phase: 0

Termination method: None

3.3.4 Add a terminator

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



Terminator: Used to "terminate" output signals. (Prevents warnings about unconnected output ports.)

Drag this icon into a model to insert the Terminator block

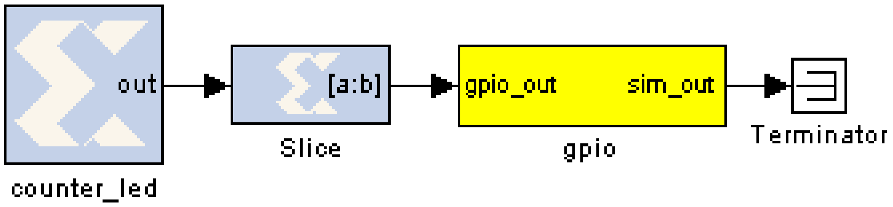
Note that all blocks from the "Simulink" library (usually white), will not be compiled into hardware. They are present for simulation only and expect continuous signals, not discreet.

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

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

3.3.5 Connect your design

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



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

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

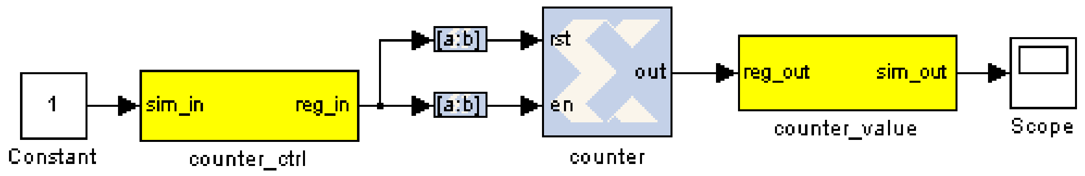
Total path length **cannot be more than 64 characters**. By “path”, I am refering to not only the file path, but also the path to any block within your design. For example, if you save this file to *c:\projects\myfile.mdl*, the longest Matlab-indexed path would be *c:\projects\myfile.mdl\counter_led*.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

Please save your design in your home directory.under ***/projects/<YOUR_INITIALS>_tut1.mdl.***

3.3.6 Software control

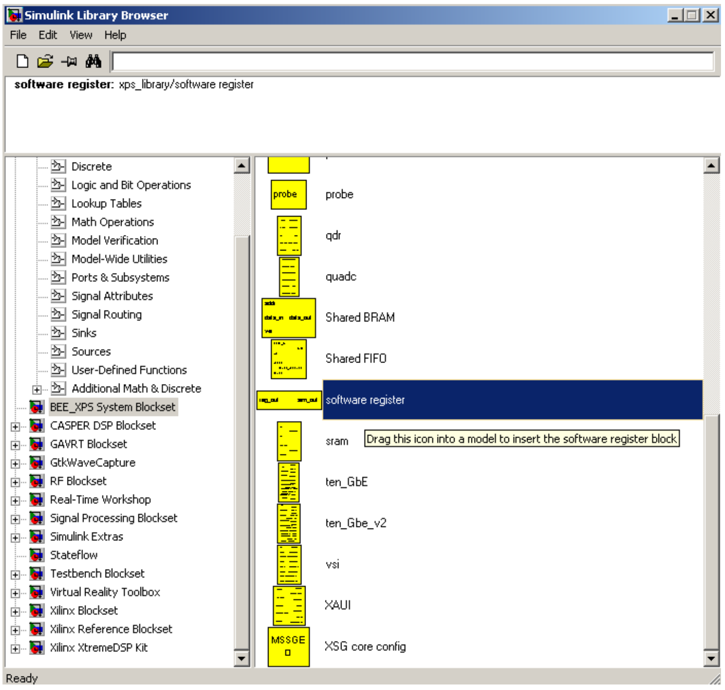
To demonstrate the use of software registers and control of the FPGA through the PPC, we will add a controllable counter to the above design. The counter can be started and stopped from software and also reset. We will be able to monitor the counter's current value too.

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

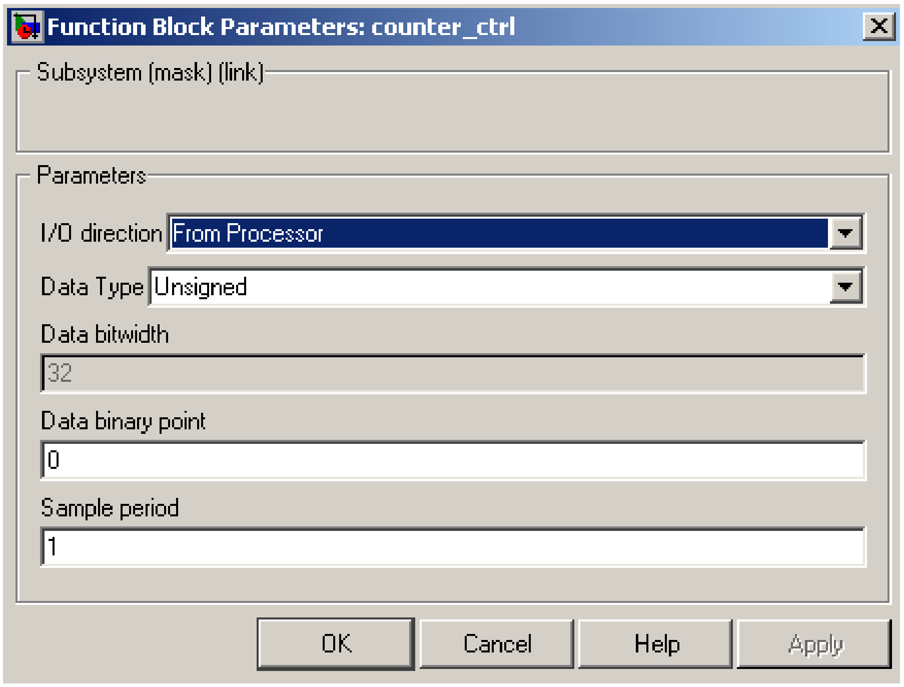


3.3.7 Add the software registers

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



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


Function Block Parameters: counter_ctrl

Subsystem (mask) (link)

Parameters

I/O direction: From Processor

Data Type: Unsigned

Data bitwidth: 32

Data binary point: 0

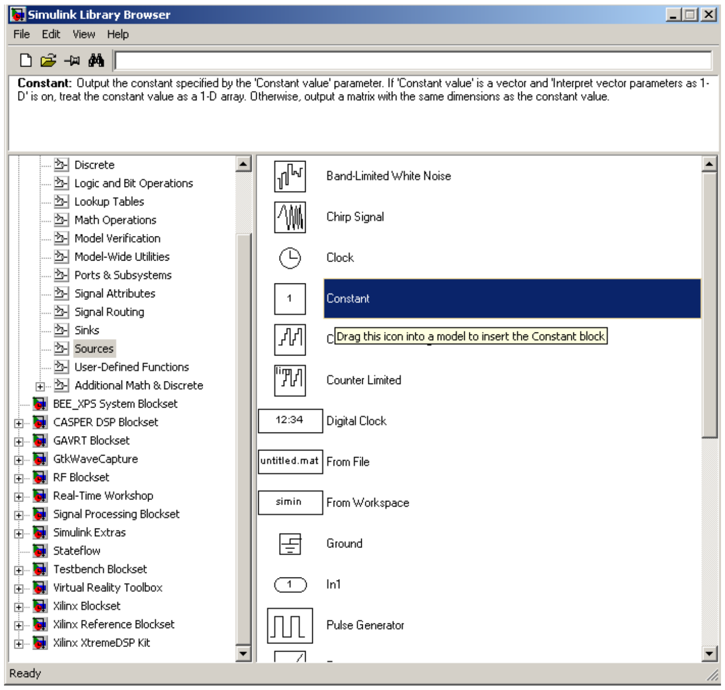
Sample period: 1

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

Rename the registers to something sensible, as these names are mapped to filenames in the PPC for controlling the design. Avoid using spaces, slashes and other funny characters in these names (spaces automatically get remapped to underscores anyway, but should be avoided for clarity). I suggest *counter_ctrl* and *counter_value,* to represent the control and output registers respectively.

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

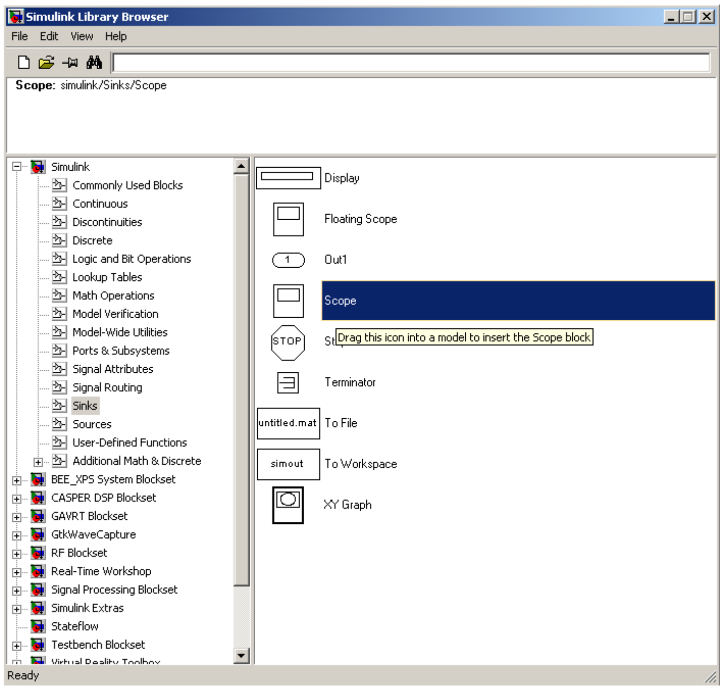
For now, set the *sim_in* port to constant one using a Simulink-type constant. This will enable the counter during simulations.



Constant: Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Drag this icon into a model to insert the Constant block

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



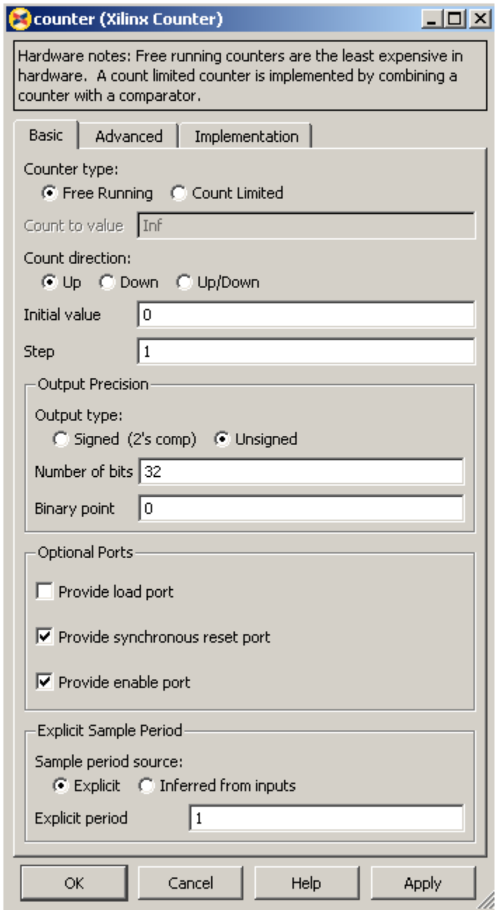
Scope: simulink/Sinks/Scope

Drag this icon into a model to insert the Scope block

3.3.8 Add the counter

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

Configure it with a reset and enable port as follows:



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.

Counter type: [✓ Free Running] Count Limited

Count to value: Inf

Count direction: [✓ Up] Down Up/Down

Initial value: 0

Step: 1

Output Precision

Output type: Signed (2's comp) [✓ Unsigned]

Number of bits: 32

Binary point: 0

Optional Ports

☐ Provide load port

☑ Provide synchronous reset port

☑ Provide enable port

Explicit Sample Period

Sample period source: [✓ Explicit] Inferred from inputs

Explicit period: 1

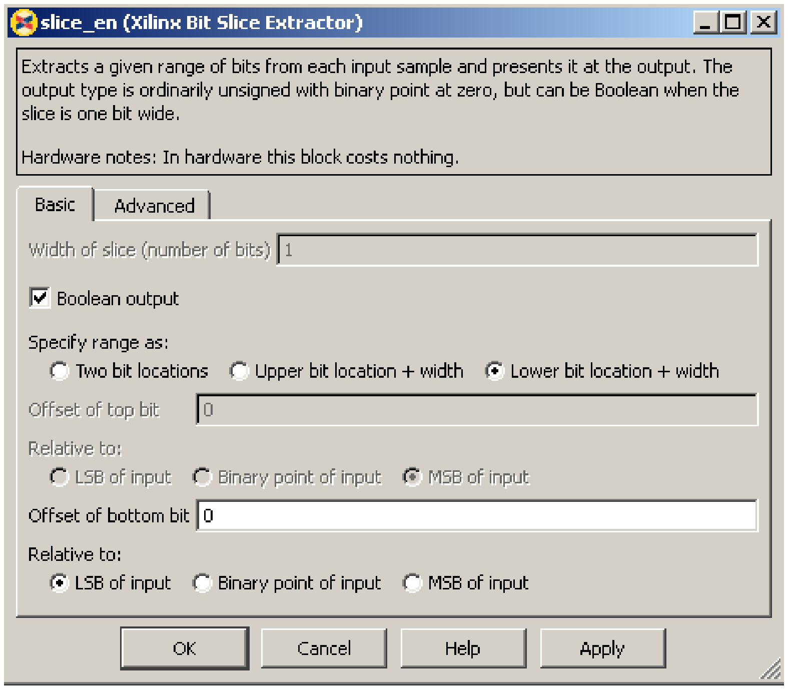
3.3.9 Add the slice blocks

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

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

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

Slice for enable:



slice_en (Xilinx Bit Slice Extractor)

Extracts a given range of bits from each input sample and presents it at the output. The output type is ordinarily unsigned with binary point at zero, but can be Boolean when the slice is one bit wide.

Hardware notes: In hardware this block costs nothing.

Width of slice (number of bits): 1

[✓] Boolean output

Specify range as: Two bit locations / Upper bit location + width / [✓ Lower bit location + width]

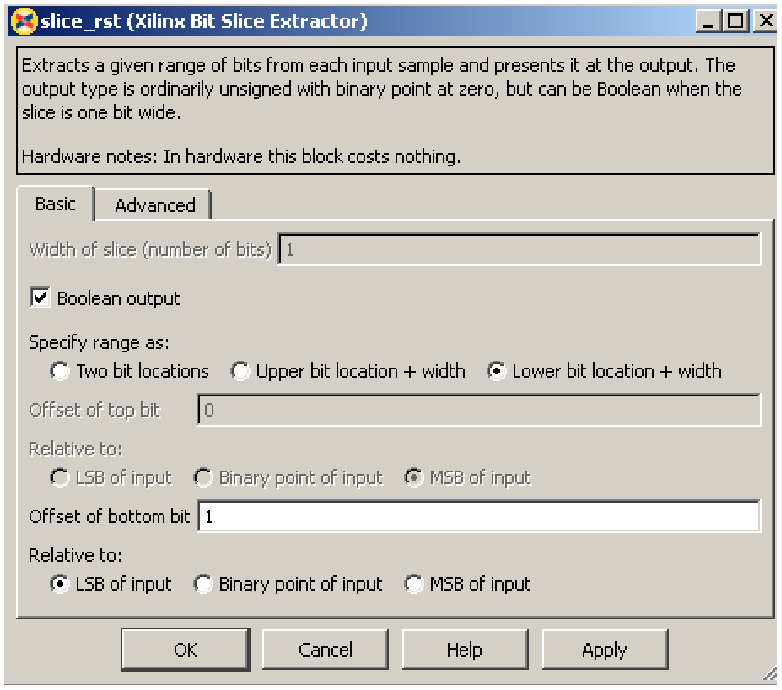
Offset of top bit: 0

Relative to: LSB of input / Binary point of input / [✓ MSB of input]

Offset of bottom bit: 0

Relative to: [✓ LSB of input] / Binary point of input / MSB of input

Slice for reset:



slice_rst (Xilinx Bit Slice Extractor)

Extracts a given range of bits from each input sample and presents it at the output. The output type is ordinarily unsigned with binary point at zero, but can be Boolean when the slice is one bit wide.

Hardware notes: In hardware this block costs nothing.

Width of slice (number of bits): 1

[✓] Boolean output

Specify range as: Two bit locations / Upper bit location + width / [✓ Lower bit location + width]

Offset of top bit: 0

Relative to: LSB of input / Binary point of input / [✓ MSB of input]

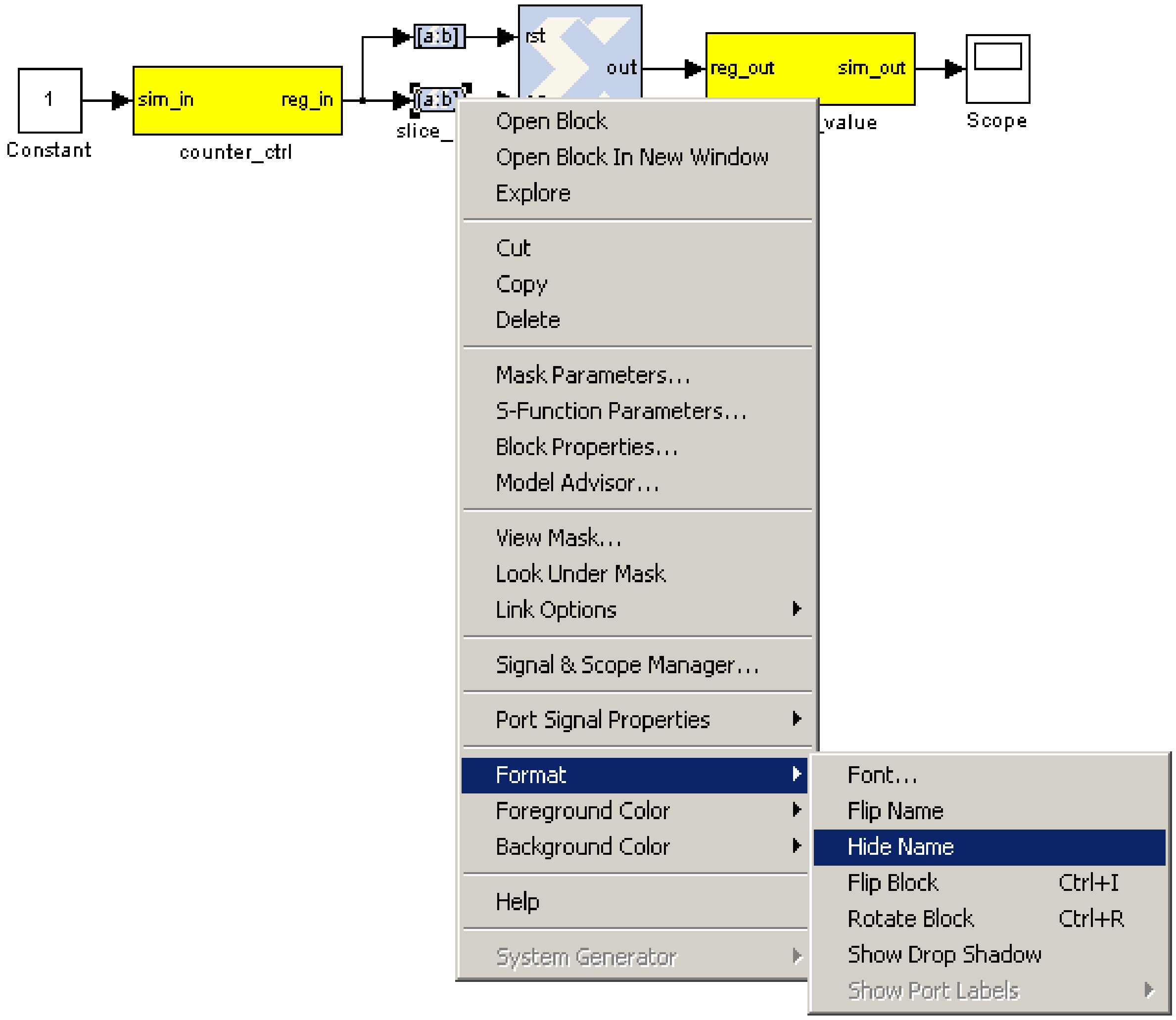
Offset of bottom bit: 1

Relative to: [✓ LSB of input] / Binary point of input / MSB of input

3.3.10 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.

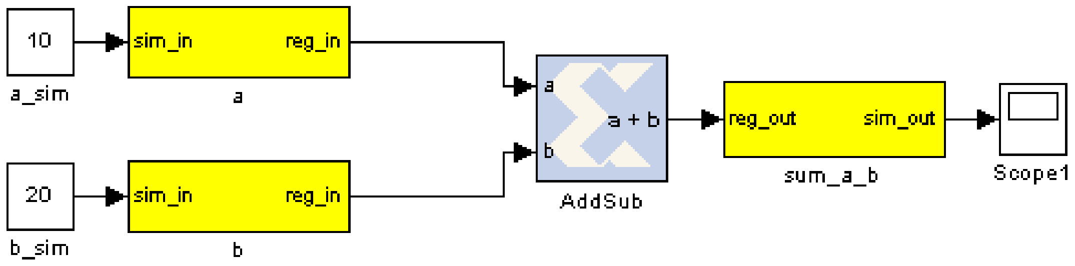
Do so by right-clicking and selecting Format → Hide Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



3.4 Adder

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

We will calculate *a+b=sum_a_b.*



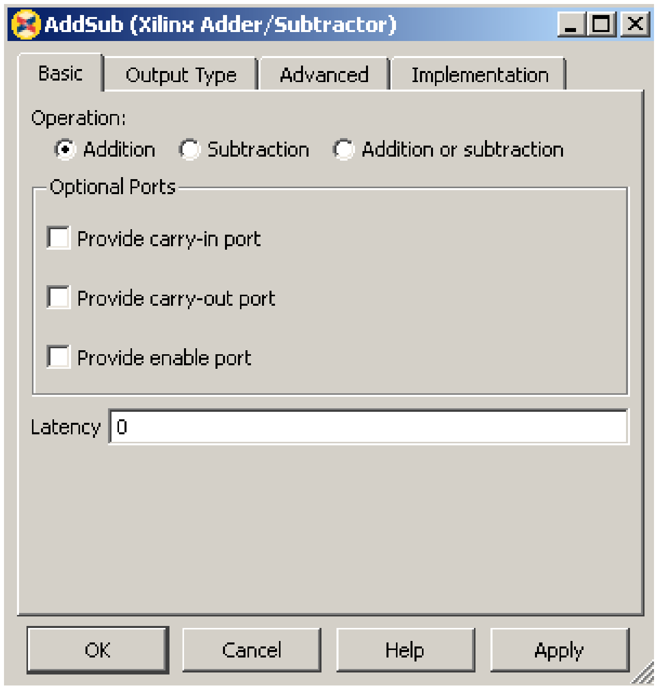
3.4.1 Add the software registers

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

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

3.4.2 Add the adder block

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


AddSub (Xilinx Adder/Subtractor)

Operation: [✓ Addition] Subtraction Addition or subtraction

Optional Ports

☐ Provide carry-in port

☐ Provide carry-out port

☐ Provide enable port

Latency 0

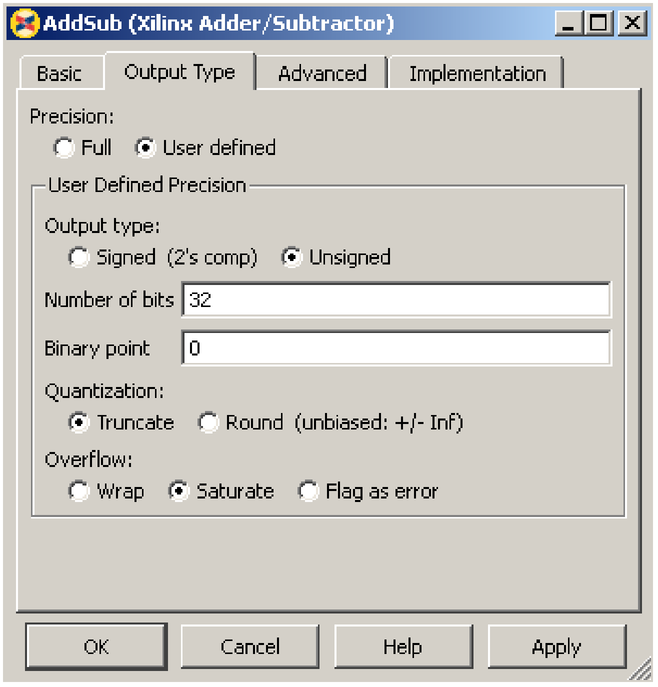
The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.

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.



Precision: ( ) Full [✓ User defined]

User Defined Precision

Output type: ( ) Signed (2's comp) [✓ Unsigned]

Number of bits: 32

Binary point: 0

Quantization: [✓ Truncate] ( ) Round (unbiased: +/- Inf)

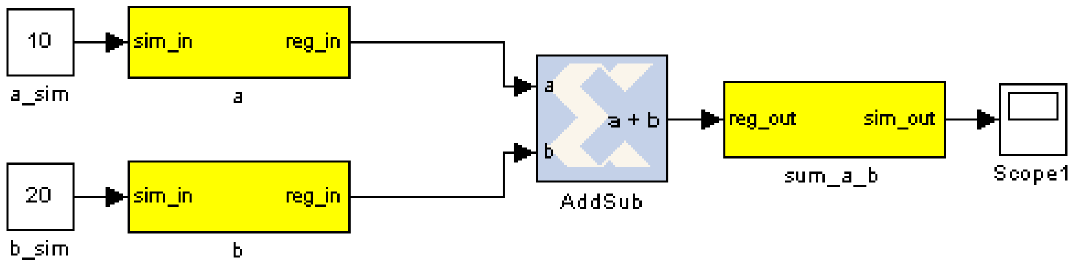
Overflow: ( ) Wrap [✓ Saturate] ( ) Flag as error

3.4.3 Add the scope and simulation inputs

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

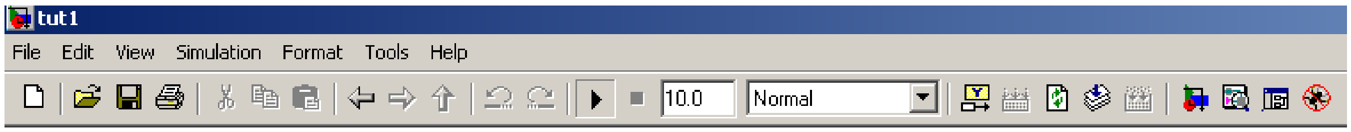
3.4.4 Connect it all together

Like this:



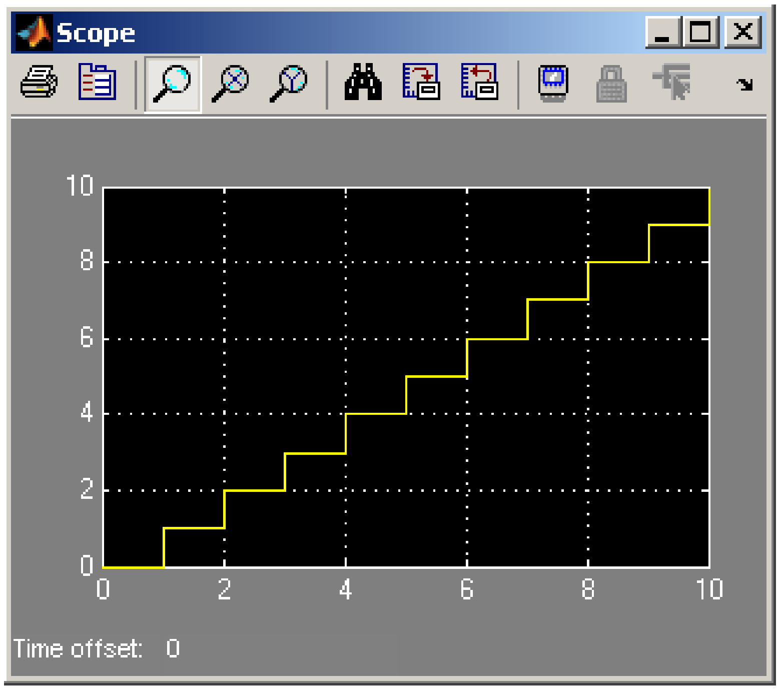
4 Simulating

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

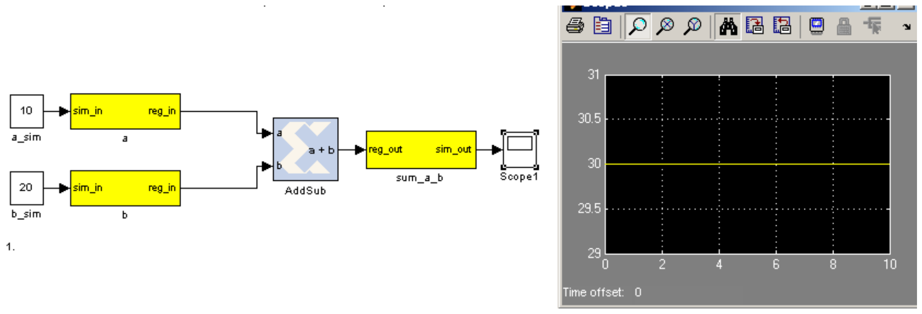


You can watch the simulation progress in the status bar in the bottom right. It will complete in the blink of an eye for this small design with just 10 clock cycles.

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



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

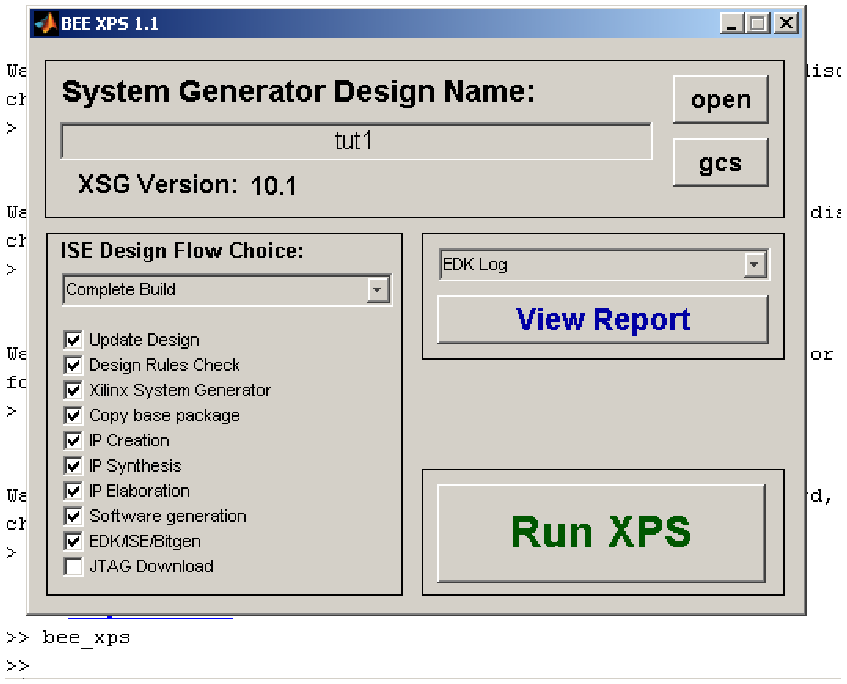


Once you have verified that that design functions as you'd like, you're ready to compile for the FPGA...

5 Compiling

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

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



BEE XPS 1.1

System Generator Design Name: tut1

XSG Version: 10.1

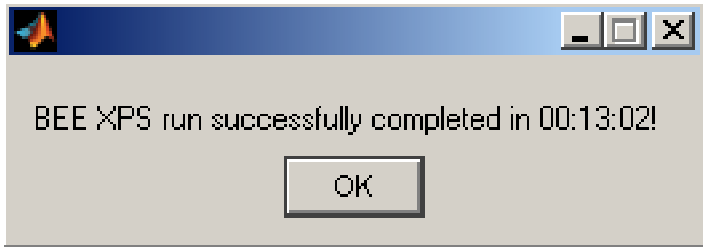
ISE Design Flow Choice: Complete Build

- [✓] Update Design
- [✓] Design Rules Check
- [✓] Xilinx System Generator
- [✓] Copy base package
- [✓] IP Creation
- [✓] IP Synthesis
- [✓] IP Elaboration
- [✓] Software generation
- [✓] EDK/ISE/Bitgen
- [ ] JTAG Download

EDK Log

```
>> bee_xps
>>
```

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


BEE XPS run successfully completed in 00:13:02!

6 Transferring you design to the FPGA

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

7 Communicating directly using BORPH

7.1 Programming the FPGA from BORPH

To program the FPGA, simply execute it as you would any other linux program. Recall that the filesystem is mapped from the server and the BORPH files were in */boffiles/*.

Let's change to that directory now and see which files are available:

```
root@roach020112:~# cd /boffiles/
root@roach020112:/boffiles# ls -al
total 8196
drwxrwxrwx  2 root root    4096 Aug 20  2009 .
drwxr-xr-x 23 root root    4096 Feb  2  2009 ..
-rwxr-xr-x  1 1000 1000 3894183 Aug 20  2009 das_blinken_lichte_2009_Feb_04_1837.bof
-rw-r--r--  1 1001 1001 4468934 Aug 14  2009 tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

We need to ensure that the file is marked as an executable, so that Linux is able to run it. You will notice that our tut1 file is not executable (look at the leftmost columns and see that there are no 'x's). Let's make it executable now:

```
root@roach020112:/boffiles# chmod a+x tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

This will change permissions on the file, by adding executable permissions for everybody. Rechecking:

```
root@roach020112:/boffiles# ls -al
total 8196
drwxrwxrwx  2 root root    4096 Aug 20  2009 .
drwxr-xr-x 23 root root    4096 Feb  2  2009 ..
-rwxr-xr-x  1 1000 1000 3894183 Aug 20  2009 das_blinken_lichte_2009_Feb_04_1837.bof
-rwxr-xr-x  1 1001 1001 4468934 Aug 14  2009 tut1_2009_Aug_14_1140.bof
root@roach020112:/boffiles#
```

Now we are ready to execute it.

```
root@roach020112:/boffiles# ./tut1_2009_Aug_14_1140.bof
```

This will go'n program the FPGA, and you should notice that LED0 on the ROACH hardware is now blinking. However, we've lost our terminal, because the process has captured it. To keep the process running, but return the prompt to us, we can now choose to background the task (ctrl-z), or we can kill it (ctrl-c) and restart it with an amperstand, which will start it backgrounded.

```
root@roach020112:/boffiles# ./tut1_2009_Aug_14_1140.bof &
[1] 323
root@roach020112:/boffiles#
```

Our FPGA is now programmed (check for the flashing LED) and we have our prompt back!

We can now see that a process in Linux has started...

```
root@roach020112:/boffiles# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND

<snip>

root       284  0.1  0.0      0     0 ?        SN   07:36   0:00 [jffs2_gcd_mtd3]
root       301  0.0  0.2   6700  1160 ?        Ss   07:36   0:00 /usr/sbin/sshd
root       311  0.0  0.0    784   192 ?        S    07:36   0:00 tcpborphserver
root       318  0.1  0.2   3772  1188 ttyS0    Ss   07:36   0:00 /bin/login --
root       319  0.1  0.3   3516  1796 ttyS0    S+   07:36   0:00 -bash
root       323  0.0  0.0   1632   304 ttyS0    S    07:36   0:00 ./tut1_2009_Aug_14_1140.bof
root       325  4.4  0.5  10000  2672 ?        Ss   07:36   0:00 sshd: root@pts/0
root       328  0.8  0.3   3524  1800 pts/0    Ss   07:36   0:00 -bash
root       332  0.0  0.1   2780   996 pts/0    R+   07:37   0:00 ps aux
root@roach020112:/boffiles#
```

Notice that PID 323 (yours may be different) is our process. We can now navigate to the *proc* directory which contains our software registers.

```
root@roach020112:/boffiles# cd /proc/323/hw/ioreg/
root@roach020112:/proc/323/hw/ioreg# ls -al
total 0
dr-xr-xr-x 2 root root 0 Aug 21 08:00 .
drwxr-xr-x 2 root root 0 Aug 21 08:00 ..
-rw-rw-rw- 1 root root 4 Aug 21 08:00 a
-rw-rw-rw- 1 root root 4 Aug 21 08:00 b
-rw-rw-rw- 1 root root 4 Aug 21 08:00 counter_ctrl
-r--r--r-- 1 root root 4 Aug 21 08:00 counter_value
-r--r--r-- 1 root root 4 Aug 21 08:00 sum_a_b
-rw-rw-rw- 1 root root 4 Aug 21 08:00 sys_board_id
-rw-rw-rw- 1 root root 4 Aug 21 08:00 sys_clkcounter
-rw-rw-rw- 1 root root 4 Aug 21 08:00 sys_rev
-rw-rw-rw- 1 root root 4 Aug 21 08:00 sys_rev_rcs
-rw-rw-rw- 1 root root 4 Aug 21 08:00 sys_scratchpad
root@roach020112:/proc/323/hw/ioreg#
```

Now you can see all our software registers. We have *a, b, counter_ctrl, counter_value* and *sum_a_b* as expect. However, in addition, the toolflow has automatically added a few other registers.

sys_board_id is simply a constant which allows software to identify what hardware platform is running. For ROACH, this is a constant 0xb00b001.

sys_clkcounter is a 32-bit counter that increments automatically on every FPGA clock tick. This allows software to estimate the FPGA's clock rate. Useful for debugging boards with bad clock inputs.

sys_rev is not yet implemented, but will eventually indicate the revision of the software toolchain that was used to compile the design

sys_rcs is also not yet implemented, but will eventually indicate the SVN revision of the CASPER library that was used to compile your design.

sys_scratchpad is simply a read/write software register where you can write a register and read it back again as a sanity check.

7.2 Communicating with your FPGA process in BORPH

The registers contain binary data. To read and represent these on our text-based terminal, we will use a Linux utility called hexdump, which simply prints out the ASCII representation of hex values (base-16) in binary files. To write into these files, we'll use Linux's echo utility. Echo does not support writing hex values, but does support octal (base-8).

Let's start by having a look at our counter value. Since we haven't started it yet, we expect it to be zero.

```
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  00 00 00 00                                     |....|
00000004
root@roach020112:/proc/323/hw/ioreg#
```

We notice that the register is indeed 32 bits long and that it contains the value zero. The column on the left tells us the memory addresses, while the four space separated values on the right give us the 4 byte (32 bit) value of the software register in hexadecimal. Right now, both registers report all zeros. According to our Simulink design, we can disable or enable the counter by setting cnt_en to 0 or 1 respectively.

Let's now start the counter and watch it increment.

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\001" > counter_ctrl
```

Here we have told *echo* not to append a newline character to the end of the line (*-n*), and told it to interpret the incoming string's escape characters (*\0* specifies octal values). Then we pipe it's output into *counter_ctrl.* Now let's relook at the counter value...

```
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  da 12 42 de                                     |..B.|
00000004
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  dd 32 7c 3c                                     |.2|<|
00000004
```

You can see that the counter is indeed incrementing, and that it is happening very quickly (remember that it's incrementing by 100 million every second, since the FPGA is running at 100MHz).

```
0xda1242de =   14291522 in decimal.
0xdd327c3c = 3711073340 in decimal.
```

You should see the register values increasing until they reach 2^32-1 and then repeat. Resetting the counter has the desired effect...

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\002" > counter_ctrl
root@roach020112:/proc/323/hw/ioreg# hd counter_value
00000000  00 00 00 00                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg#
```

For you to try: what happens if you try to reset and enable the counter at the same time? Try it in simulation and on the hardware. Do they do the same thing? Is this what you expect?

Let's now consider our adder: All registers initialise to zero upon startup, so we'd expect *a* and *b* to be zero now.

```
root@roach020112:/proc/323/hw/ioreg# hd a
00000000  00 00 00 00                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg# hd b
00000000  00 00 00 00                                       |....|
00000004
```

Indeed, this is the case. Let's write something in there now and have a look at the output. Let's add 5 and 12, so we expect 17.

We use *0x* to indicate hex, *0o* for octal and *0b* for binary. No prefix indicates decimal.

```
05 = 0o05 = 0x05
12 = 0o14 = 0x0c
17 = 0o21 = 0x11
```

```
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\005" > a
root@roach020112:/proc/323/hw/ioreg# echo -n -e "\000\000\000\014" > b
root@roach020112:/proc/323/hw/ioreg# hd a
00000000  00 00 00 05                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg# hd b
00000000  00 00 00 0c                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg# hd sum_a_b
00000000  00 00 00 11                                       |....|
00000004
root@roach020112:/proc/323/hw/ioreg#
```

Great! Exactly as expected.

For you to try: What happens if you try to add two very large numbers (try to make the 32-bit register overflow).

This shows you a basic view of BORPH and interfacing to the proc files directly from the ROACH using Linux utilities. This method of accessing the shared memory and registers is good for quick verification that your design is running and loaded correctly, but for more advanced command, control and data acquisition, we recommend using the tcpborphserver and KATCP that starts up automatically when booting ROACH.

8 Interacting using KATCP

KATCP is a process running on the ROACH boards which listens for TCP connections on port 7147 (*tcpborphserver*). It talks using machine-parseable ASCII text strings. It was designed this way so that it is easy to debug by watching the exchange of network traffic, whilst still being easy to program clients and servers.

8.1 Connecting to your ROACH

Telnet allows you to connect directly to an open TCP port. Let's connect to ROACH's KATCP port now.

Telnet roach<SerialNumber> 7147

You will be greeted with the KATCP welcome header.

```
#version tcpborphserver-1.0
#build-state tcpborphserver-0.
#log info 1250851713365 tcpborphserver
new\_connection\_192.168.14.1:53088\_to\_192.168.14.112:7147
```

From here you can type KATCP commands. A full list, reference guide and specification sheet can be found on the CASPER wiki at http://casper.berkeley.edu/wiki/KATCP.

To summarize:
•All commands must be preceded by a *?,* and submitted with a *CR* or *LF.*
•All commands will be acknowledged with a response, preceded by a *!.*
•Multi-line responses will be preceded by a # on each line, followed by the standard acknowledge response.
•Spaces are considered argument delimiters. Arguments with spaces are escaped using _.

Let's look at some examples:

To get online help and see a list of commands available, type *?help* and press *enter*.

```
?help
#help tap-stop stop\_the\_tap\_server\_(?tap-stop)
#help tap-start start\_the\_tap\_server\_(?tap-start\_10gbe-register-name...
#help echotest basic\_network\_echo\_tester\_(?echotest\_ip-address\_ip-p...
#help wordwrite writes\_data\_words\_to\_a\_named\_register
#help indexwrite writes\_arbitrary\_data\_lengths\_to\_a\_numbered\_register
#help write writes\_arbitrary\_data\_lengths\_to\_a\_named\_regi...
#help wordread reads\_data\_words\_from\_named\_a\_register
```

```
#help indexread reads\_arbitrary\_data\_lengths\_from\_a\_numbered\_register
#help read reads\_arg3\_bytes\_starting\_at\_arg2\_offset\_from\_register\_arg1\...
#help listdev displays\_available\_device\_registers
#help listcmd displays\_available\_shell\_commands
#help listbof displays\_available\_images
#help status displays\_image\_status\_information
#help progdev programs\_an\_image\_(?progdev\_[boffile])
#help sensor-sampling sets\_the\_sensor\_reporting\_mode
#help sensor-list lists\_available\_sensors
#help watchdog pings\_the\_system
#help log-level sets\_the\_minimum\_reported\_log\_priority
#help help displays\_this\_help
#help restart restarts\_the\_system
#help halt shuts\_the\_system\_down
!help ok 21
```

Here you can see what a multiline response looks like. Although a little hard to read by us humans, this is easy to parse by a machine. We have clients available for decoding this text and constructing commands automatically. They will be demonstrated in Tutorial 2. For now, let's look at constructing some commands by hand.

To get a list of bof files available, type *?listbof.*

```
?listbof
#listbof das_blinken_lichte_2009_Feb_04_1837.bof
#listbof tut1_2009_Aug_14_1140.bof
!listbof ok
```

To program the FPGA with one of these bitstreams, type *?progdev <my_boffile>.*

```
?progdev tut1_2009_Aug_14_1140.bof
!progdev ok
```

You will notice that commands are ignored if you don't include the ? in the front.

To list the registers available to you, type *?listdev*.

```
!progdev ok
?listdev
#listdev sum_a_b
#listdev counter_value
#listdev counter_ctrl
#listdev b
#listdev a
#listdev sys_clkfreq
#listdev sys_clkcounter
#listdev sys_scratchpad
#listdev sys_rev_rcs
#listdev sys_rev
#listdev sys_board_id
!listdev ok
```

Here you can see we have the same list as we had before in BORPH.

Normally, machines using this interface would read and write to these registers using raw binary numbers using the *read* or *write* commands. For manual interaction, there are *wordwrite* and *wordread* commands which do the same with ASCII hex representations of 32-bit values. Let's try and add two numbers together now.

```
?wordwrite a 0 0x02
!wordwrite ok
?wordwrite b 0 0x07
!wordwrite ok
?wordread sum_a_b 0
!wordread ok 0x9
```

You may be wondering what the extra zero in the arguments is for. This is the index offset. It is used when writing to blocks of memory, rather than software registers. For example, if you wanted to write a single 32 bit number into a 1GB DRAM memory chunk, at address 0x12808, you would say ?wordwrite <my_dram> 0x12808 <my_value>.

As you can see, the same FPGA functions that are available in BORPH are accessible through KATCP, with the difference that it can be configured remotely over a TCP network stream.

9 Conclusion

This concludes Tutorial 1. You have learnt how to constuct a simple Simulink design, transfer the files to a ROACH board and interact with it using BORPH and KATCP.

In Tutorial 2, you will learn how to use the 10GbE network interfaces and interact with your design using the KATCP Python client.