CASPER Workshop

Tutorial 1 : Introduction to Simulink

Dev. By : M. Wagner, J. Manley and W. New

Doc. By : Irappa M. Halagali GMRT/NCRA/TIFR

Expected completion time: 1 hr

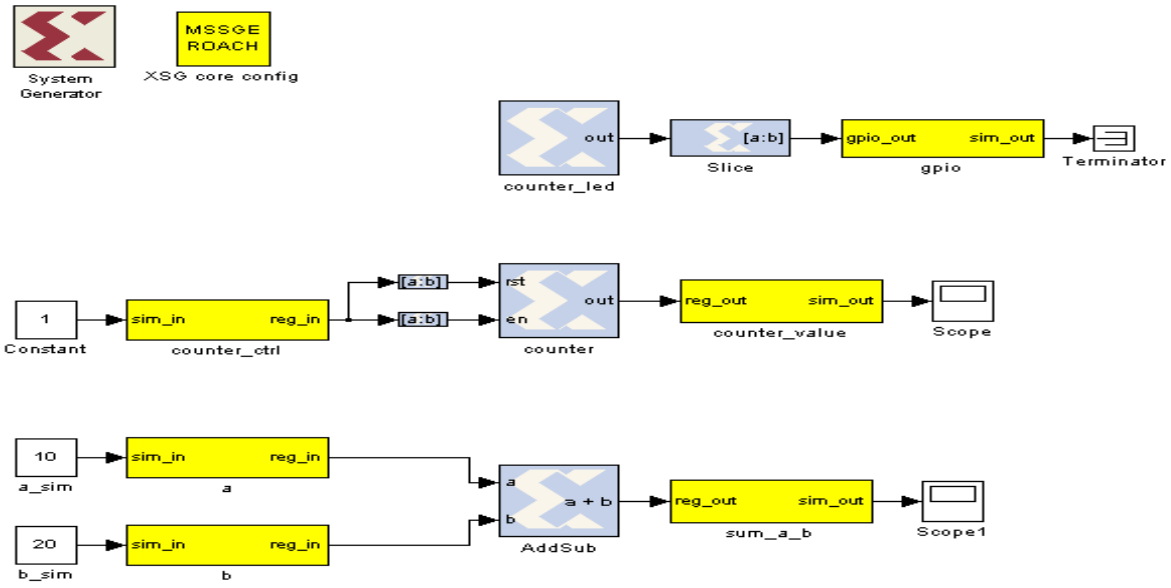
Modified from Mark Wagner's original ROACH tutorial at casper.berkeley.edu/wiki/Roach_Tutorial.

Contents :

1 The Hardware and Software required for this tutorial.

1. **PC : Dell Intel(R) Core(TM) i3 CPU 530 @ 2.93GHz width 64 bit & 4GB RAM**
2. **OS : Linux 2.6.35-30-generic #54-Ubuntu 10.10 SMP x86_64 GNU/Linux**
3. **Matlab : 2008a**
4. **Xilinx : version 11.5**
5. **Casper : gits_100511**
6. **minicom : version 2.4 (compiled on Jun 3 2010)**
7. **ROACH unit : version 1.0 Rev 3 2009 , uboot : uboot-2010-07-15-r3231-dram , Linux Kernel Image : uImage-jiffy-20091110**

2 The final view of the design



3 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.

4 Setup

The lab at the workshop is preconfigured with the CASPER libraries, Matlab and Xilinx tools. Please refer the file **"LOCATIONSandFILES.pdf"** in the **home/Desktop area** or **LOCATIONSandFILES slides displayed** , for the locations/directories and files information required in the tutorial. Note : The Date and Time portion of the BOF file name will be different ! It depends upon when (Date & Time) you complile your model file !

Note : All the following cable connections and entries in the /etc/* files of the workshop PCs are already done.

1. Connect the Serial port cable between the ROACH board's P2 connector and serial port of the PC (on which minicom program exists).
2. Connect the Ethernet cable to J25 port of the ROACH board from the PCs eth1 port. /etc/ethers file should have mac address and corresponding ip address. In the /etc/network/interfaces file , eth1 should be configured. And in the file /etc/hosts , ip address and corresponding roach board (host) name entry to be done.
3. Create your own directory at **"[MATLAB_START_DIR]" ,** to save and compile your model file.
4. **Start the matlab :**

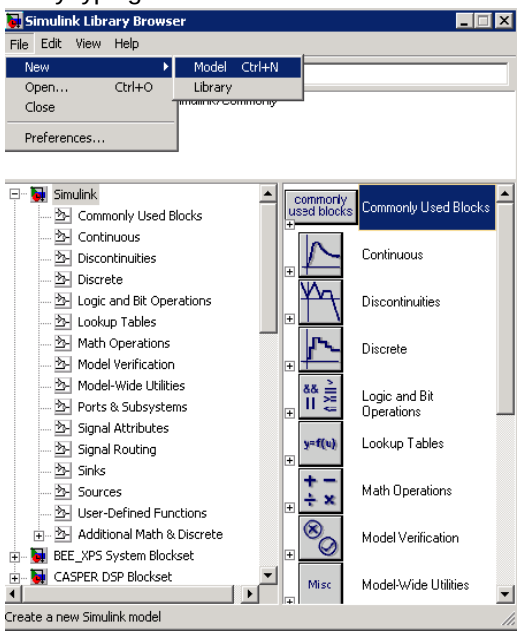
$ cd [MATLAB_START_DIR]
[MATLAB_START_DIR]**$** ./[MATLAB_START_FILE] &

5 Creating Your Design

5.1 Create a New Model:

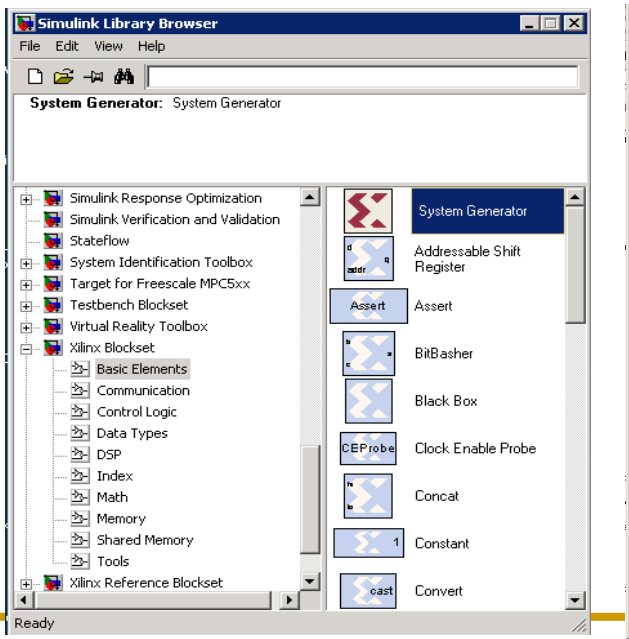
Start Matlab and open Simulink (either by typing *simulink* on the Matlab command line, or by click in the Simulink icon in the taskbar).

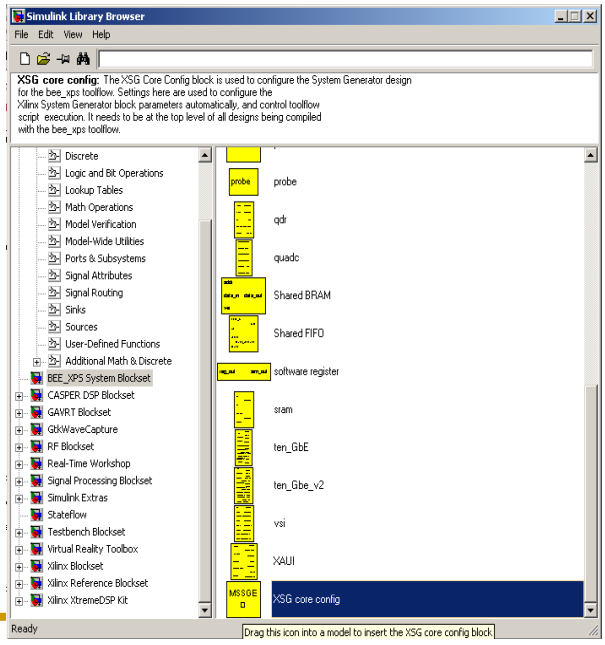
Create a new model:



5.2 Add Xilinx System Generator and XSG core config blocks:

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

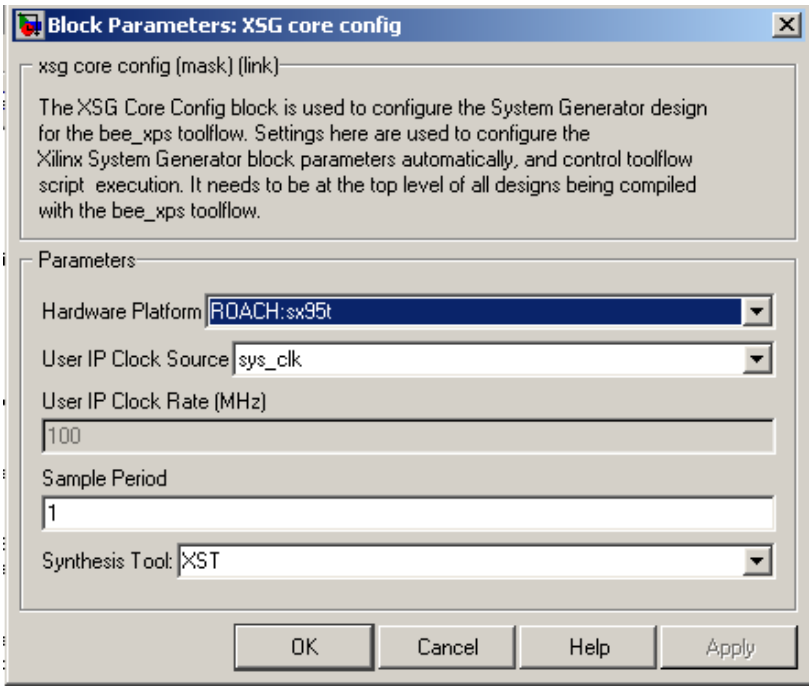




All hardware-related blocks are yellow and can be found in the *BEE_XPS* library. This library contains all the board-specific components colloquially called *Yellow Blocks.*
DSP related blocks are found in the CASPER DSP library and have other colours. Tutorial 3 will introduce you to these blocks.

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

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



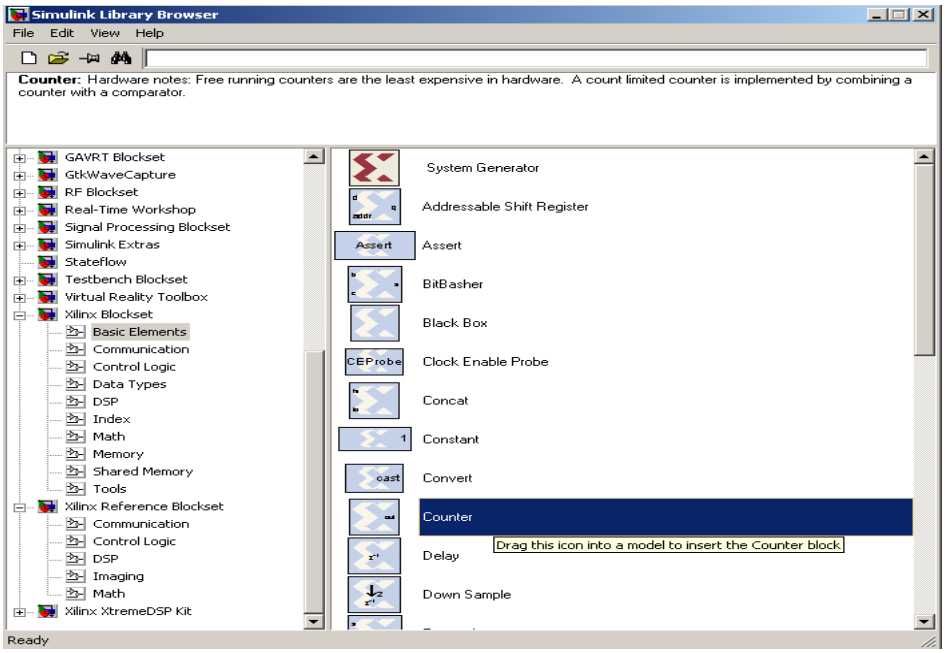
This will go off and configure the System Generator block which you previously added.
You need to add these two blocks for all CASPER designs.

5.3 Flashing LED

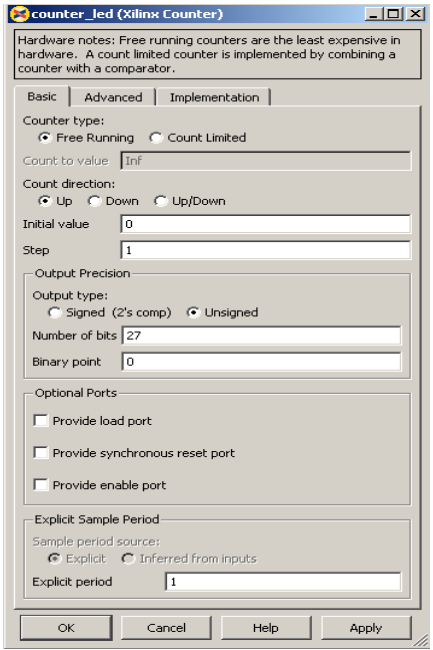
To demonstrate the basic use of hardware interfaces, we will make an LED flash. With the FPGA running at 100MHz, the most significant bit (msb) of a 27 bit counter will toggle every 0.745 seconds. We can output this bit to an LED on ROACH. ROACH has four green LEDs. We will now connect a counter to the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

5.3.1 Add a counter

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

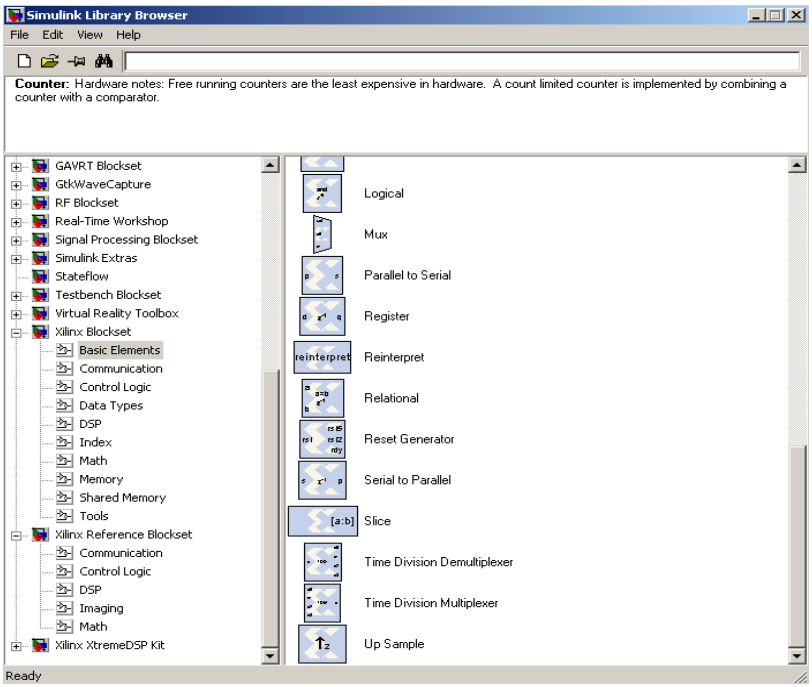


Double-click it and set it for free running, 27 bits, unsigned.



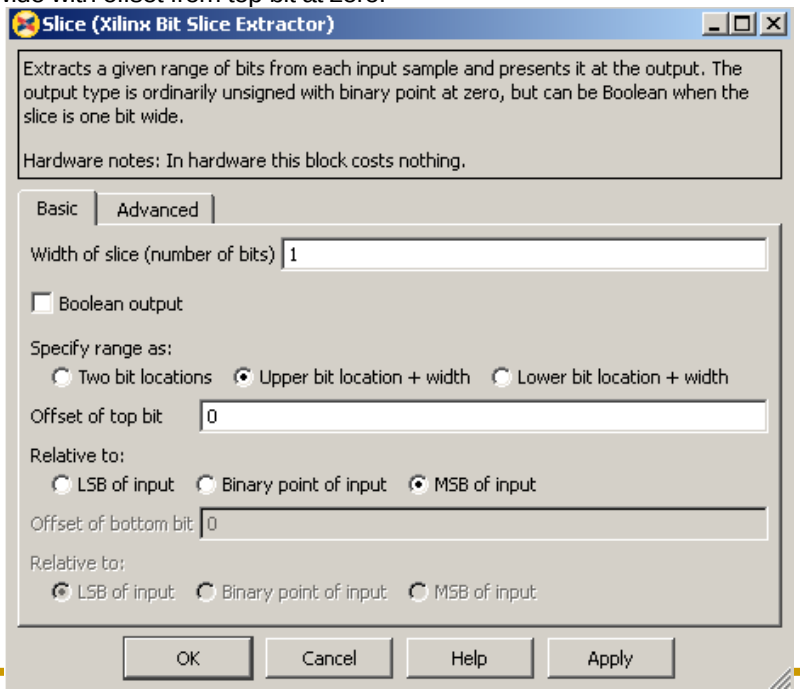
5.3.2 Add a slice block to select out the msb

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



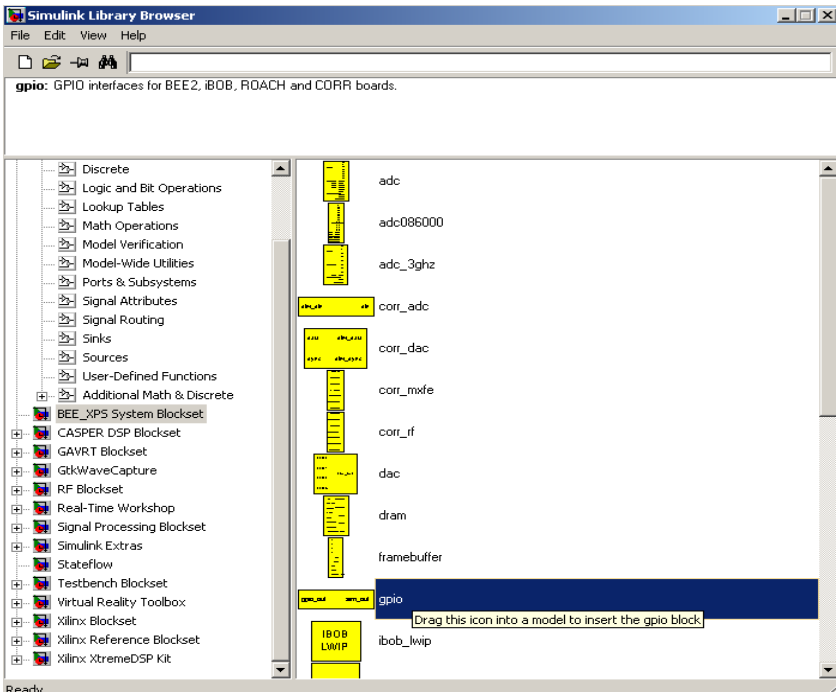
Double-click on the newly added slice block. There are multiple ways to select which bit(s) you want. In this case, I find it simplest to index from the upper end and select the first bit. If you wanted the lsb, you could also index from the lsb,. You can either select the width and offset, or two bit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

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

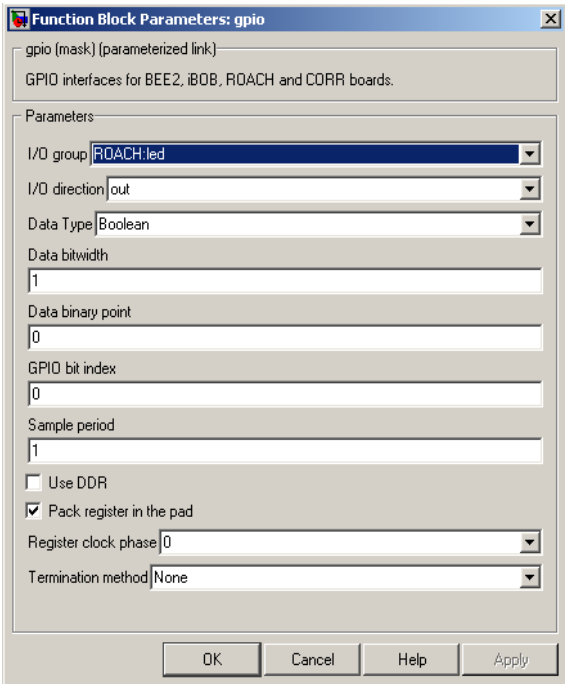


5.3.3 Add a GPIO block

(BEE_XPS library -> gpio).

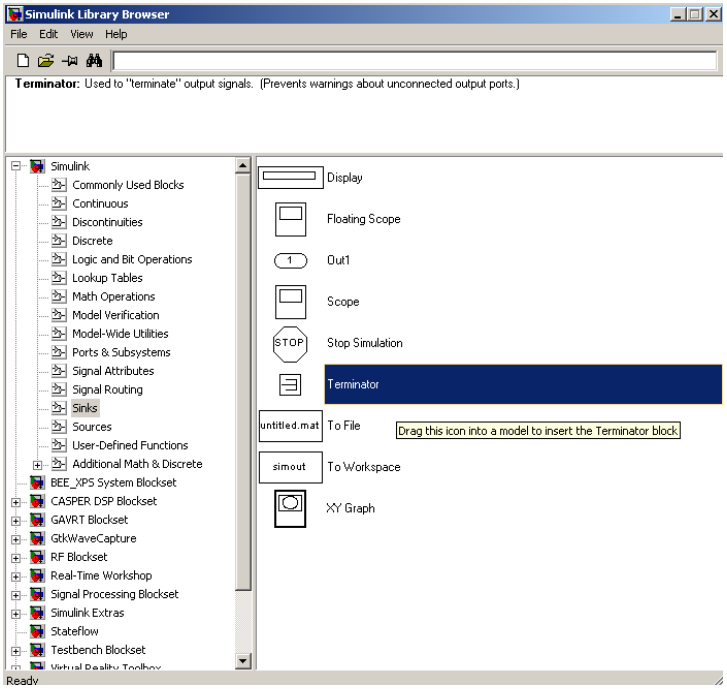


Set it to use ROACH's LED bank as output, GPIO bit index 0 (the first LED).



5.3.4 Add a terminator

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



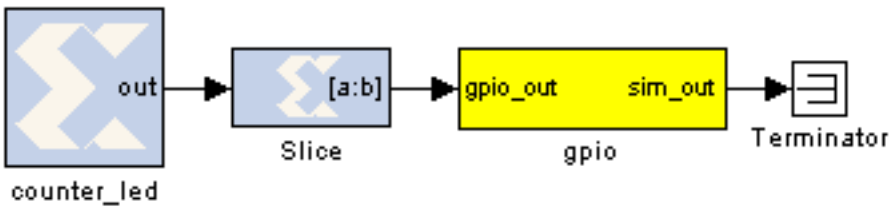
Note that all blocks from the "Simulink" library (usually white), will not be compiled into hardware. They are present for simulation only and expect continuous signals, not discreet.

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

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

5.3.5 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:

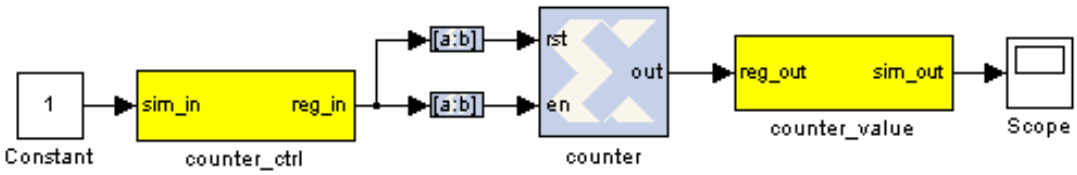
Do not use spaces in your filenames, or anywhere in the file path as it will break the toolflow.

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.***

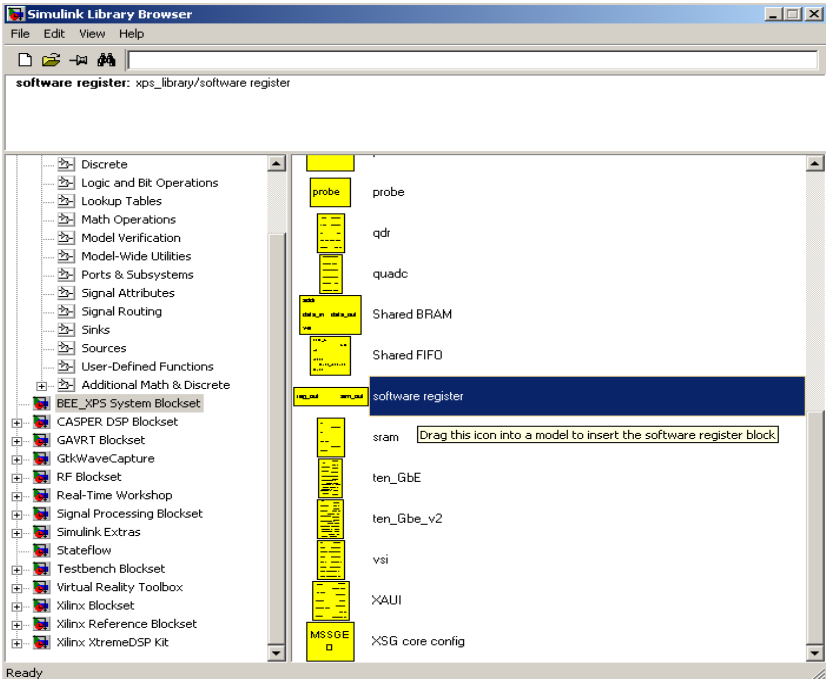
5.3.6 Software control

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:

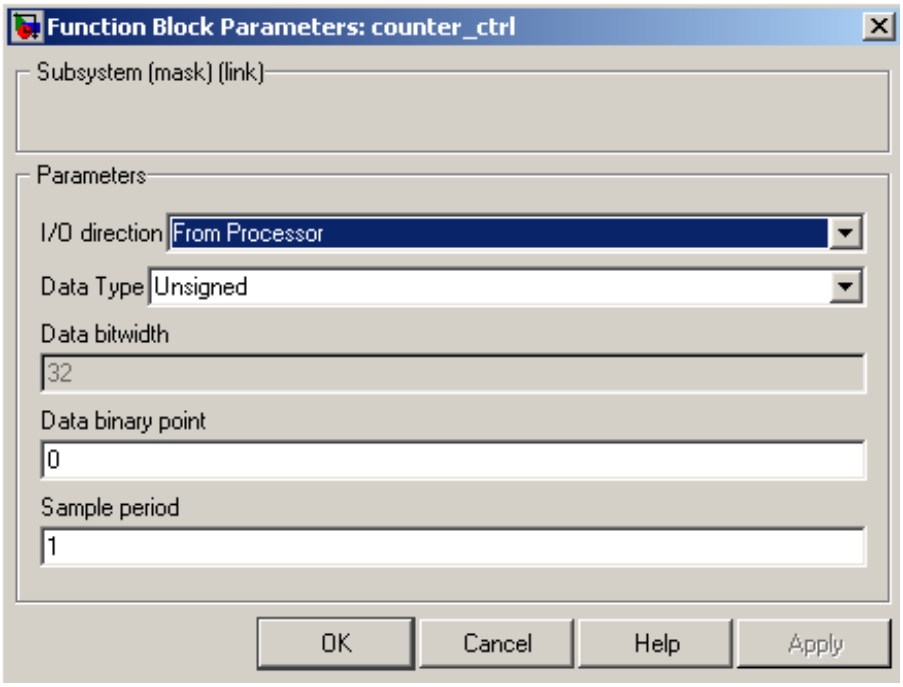


5.3.7 Add the software registers

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



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

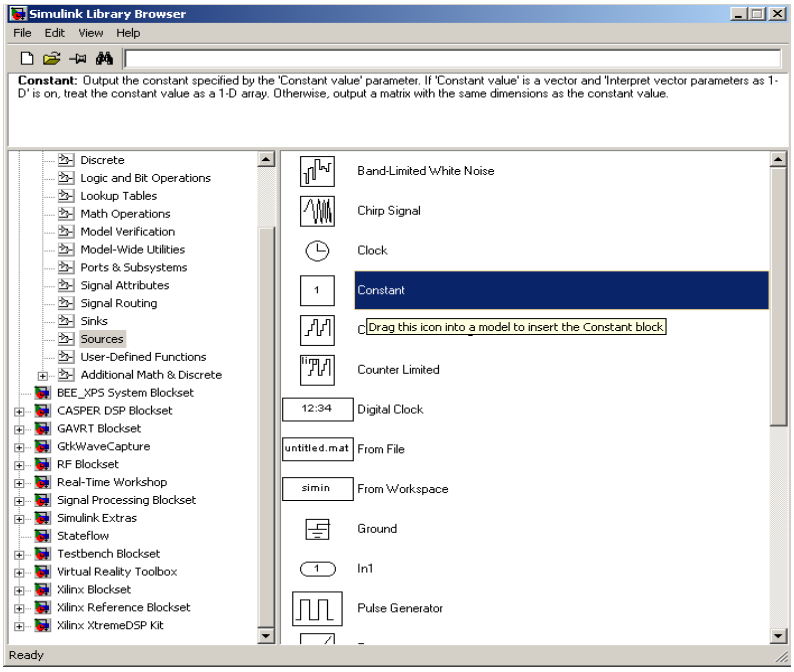


Note the field *Data bitwidth* is greyed out with a value 32. This is because all software registers have a fixed data bitwidth of 32 bits.

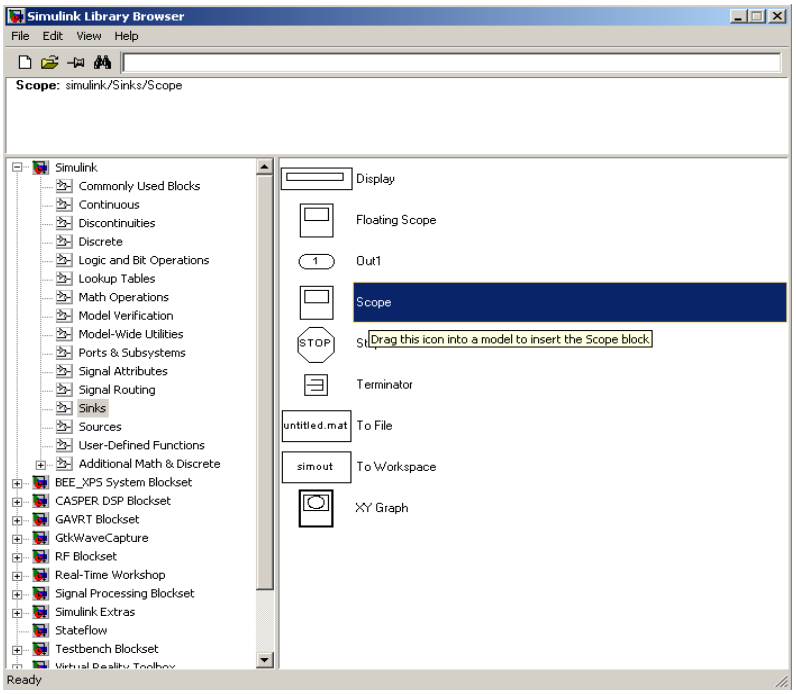
Rename the registers to something sensible, as these names are mapped to filenames in the PPC for controlling the design. Avoid using spaces, slashes and other funny characters in these names (spaces automatically get remapped to underscores anyway, but should be avoided for clarity). I suggest *counter_ctrl*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

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

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



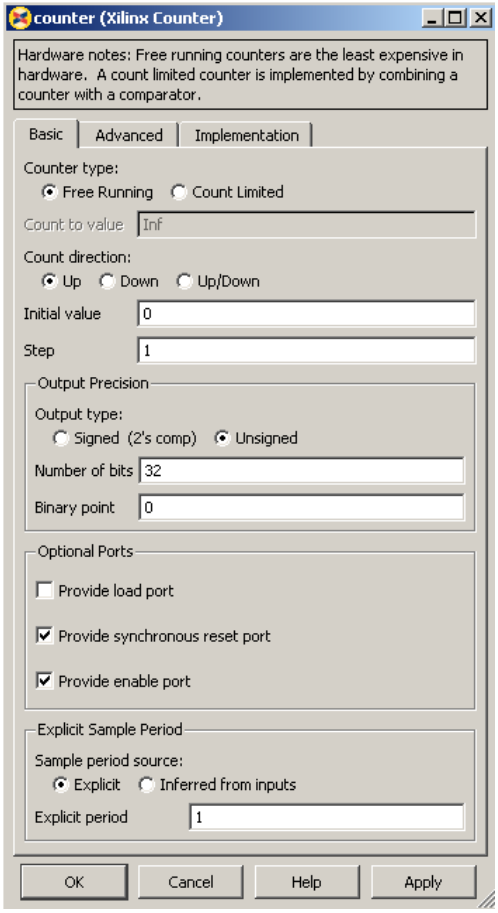
During simulation, we can monitor the counter's value using a scope:



5.3.8 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:



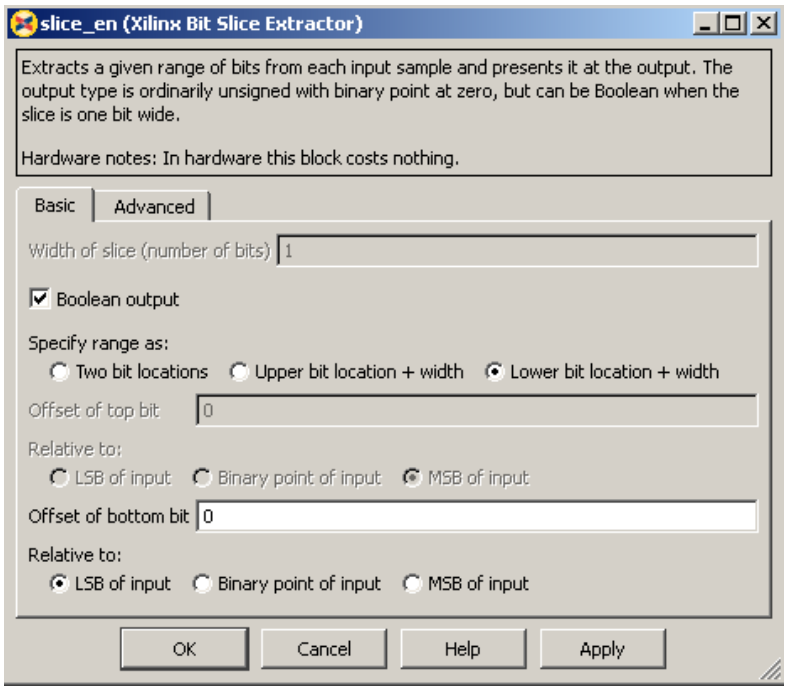
5.3.9 Add the slice blocks

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

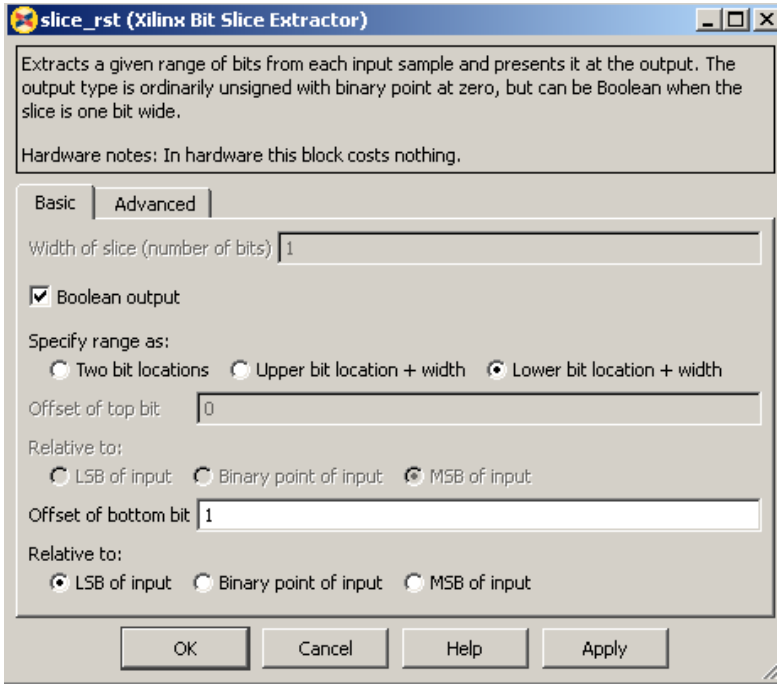
So we'll use a single register and slice out one bit for enabling the counter, and another bit for resetting it. Either copy your existing slice block (copy-paste it or hold ctrl while dragging/dropping it) or add two more from the library.

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

Slice for enable:



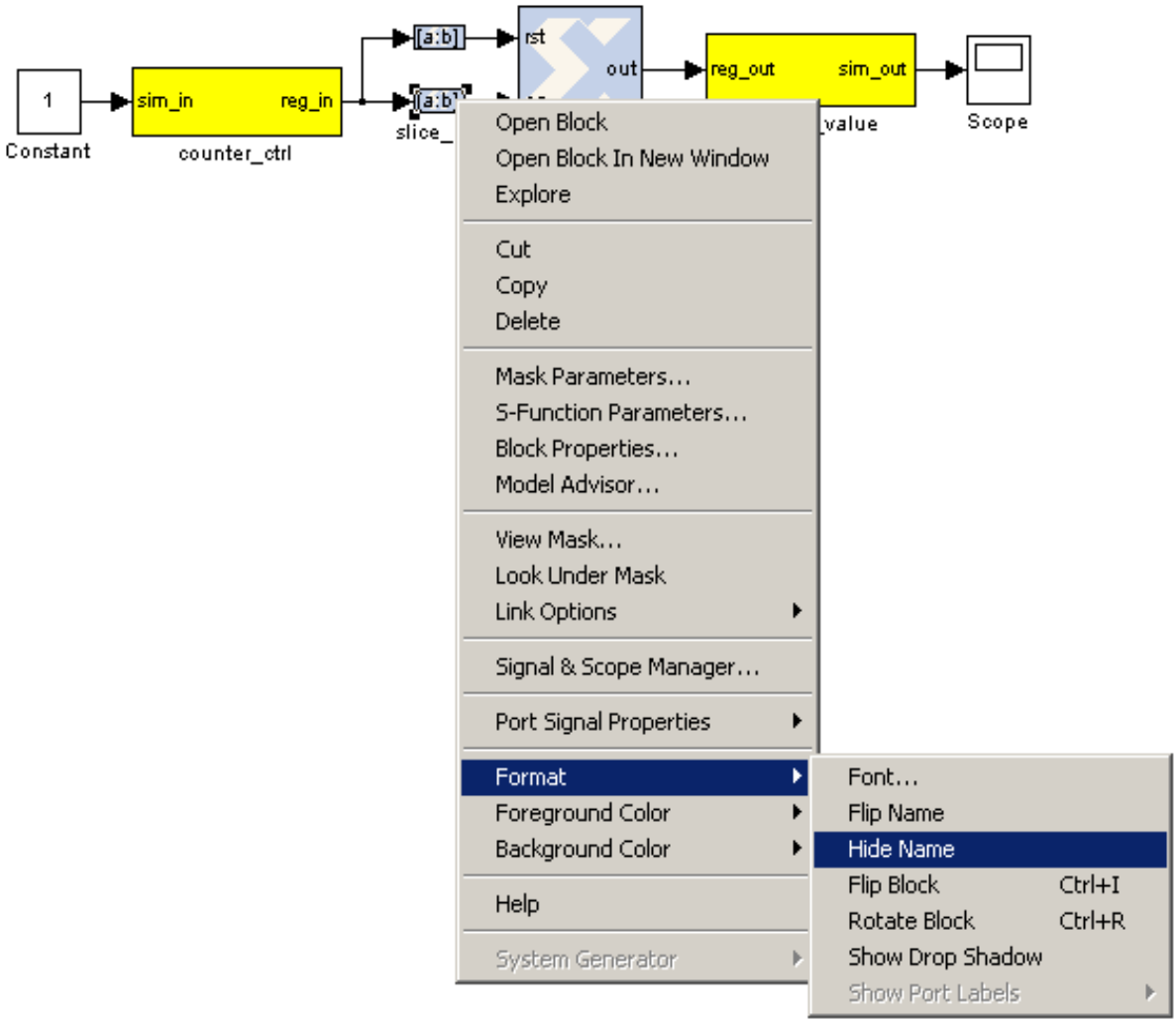
Slice for reset:



5.3.10 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.

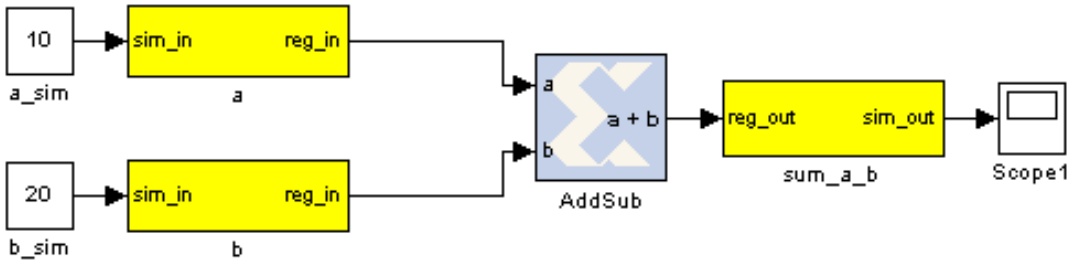
Do so by right-clicking and selecting Format → Hide Name. You could do this with the counter too, but it's not a good idea with the software registers, because otherwise you wouldn't know how to address them when looking at your diagram.



5.4 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*.



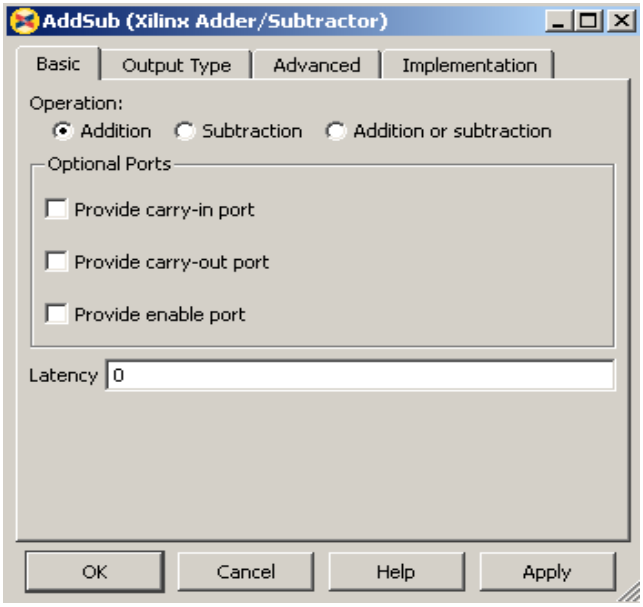
5.4.1 Add the software registers

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.

5.4.2 Add the adder block

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

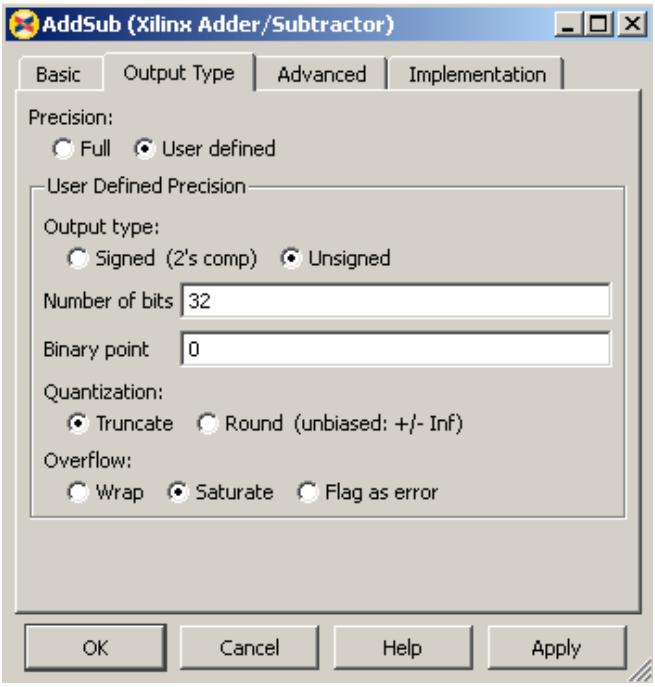


The output register is 32 bits. If we add two 32 bit numbers, we will have 33 bits.
There are a number of ways of fixing this:

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

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

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

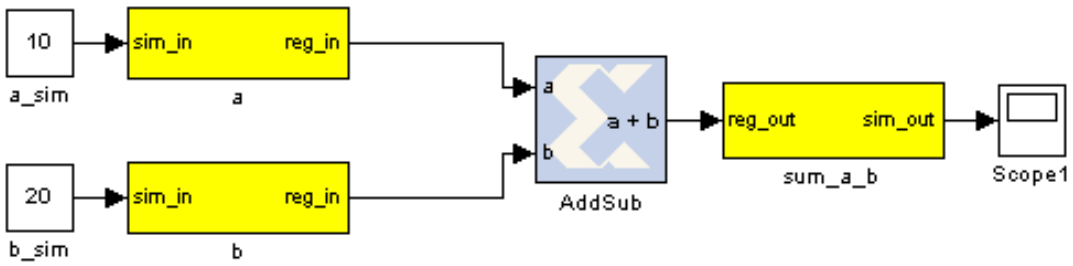


5.4.3 Add the scope and simulation inputs

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.

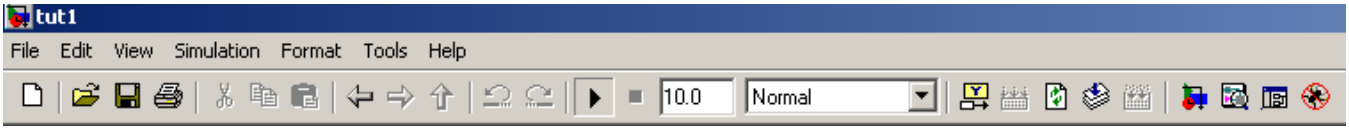
5.4.4 Connect it all together

Like this:

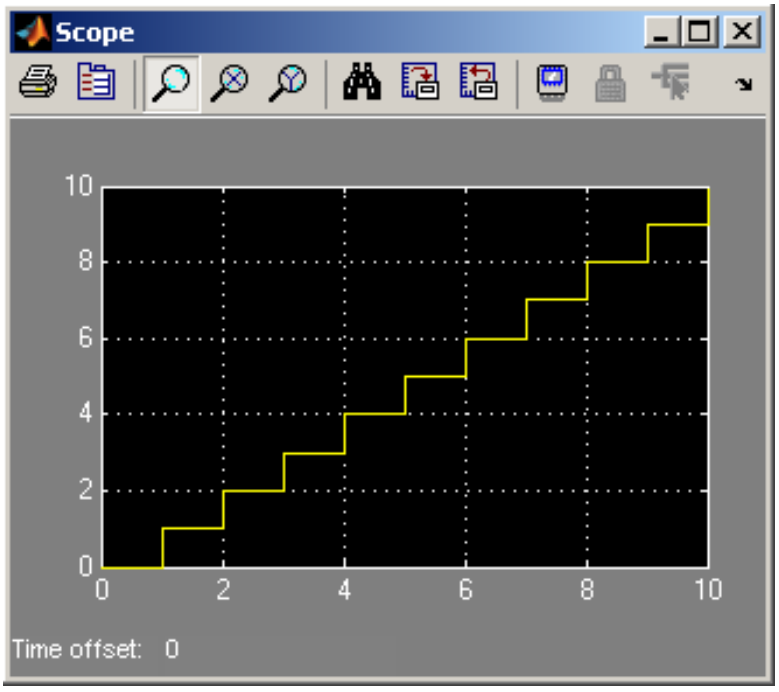


6 Simulating

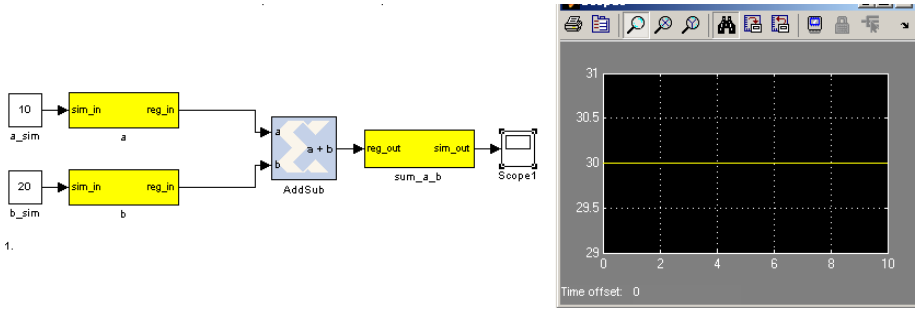
The design can be simulated with clock-for-clock accuracy directly from within Simulink. Set the number of clock cycles that you'd like to simulate and press the play button in the top toolbar.



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:



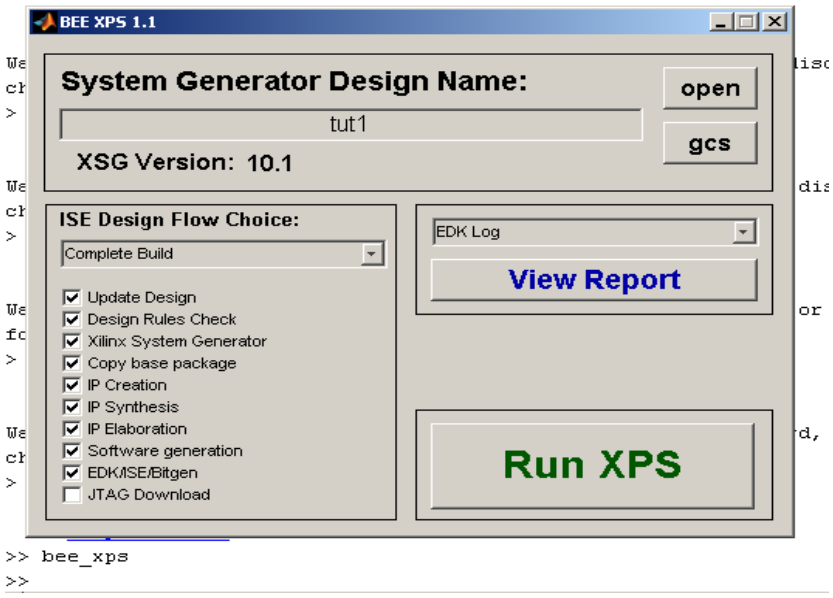
The one connected to your adder should return a constant, equal to the sum of the two numbers you entered. You might have to press the *Autoscale* button to scale the scope appropriately.



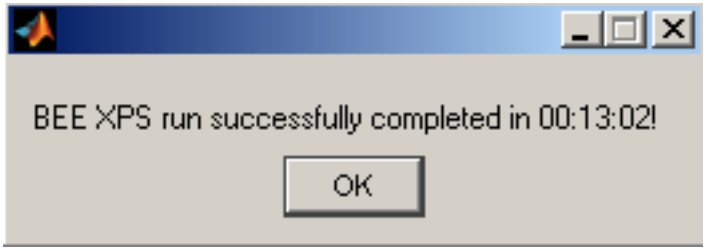
Once you have verified that that design functions as you'd like, you're ready to compile for the FPGA.

7 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*.After compilation , it creates a directory named after the model file name without the .mdl extension. There is a sub directory named bit_files. In this bit_files directory there are .bit and .bof file. We need the .bof file to program the FPGA. You need to save this .bof file at location **[FPGA_PROG_BOF_DIR]** .



Compile time is approximately 4 minutes on the above mentioned computer. When complete, you should receive a popup box like this:



6 Programming the FPGA using BORPH

1. Copy the bof file to be programed which is compiled by you in the directory **"[FPGA_PROG_BOF_DIR]"** after changing the permissions of the file.

eg. for the bof file
[TUT1_BOF_FILE] in the area **[STD_BOF_DIR]**

$ **chmod a+x [STD_BOF_DIR]/[TUT1_BOF_FILE]**

$ **cp [STD_BOF_DIR]/ [TUT1_BOF_FILE] [FPGA_PROG_BOF_DIR]**

Note : All these cable connections and entries in the /etc/* files of the workshop PCs done.

2. Connect the Serial port cable between the ROACH board's P2 connector and serial port of the PC (on which minicom program exists).

3. Connect the Ethernet cable to J25 port of the ROACH board from the PCs eth1 port. /etc/ethers file should have mac address and corresponding ip address. In the /etc/network/interfaces file , eth1 should be configured. And in the file /etc/hosts , ip address and corresponding roach board(host) name entry to be done.

4. Start the minicom program after **login as root**.

eg. $ **minicom**

Initializing Modem

Welcome to minicom 2.2
OPTIONS: I18n
Compiled on Mar 9 2007, 07:21:40.
Port /dev/ttyS0
Press CTRL-A Z for help on special keys.

NOTE : Press CTRL + A and then Z ; to get Minicom command summary.
Then press W for Line Wrap ON/OFF. This enables us to see the messages from the ROACH board during BOOT-UP.

5. **Switch ON the ROACH board..** Refer the file **"[DOC_DIR]/[ROACH_BOOT_PROC_FILE]"** for complete boot procedure of the ROACH BOARDs. For this purpose it is not required to refer.

6. roach030172 login: **root** # login as root w/o any password.

7. root@roach030172:/# **cd /boffiles/**

8. root@roach030172:/boffiles# **ls** #The ouput display on your PC may be different.

imh_tut1a_2011_Jul_18_1439.bof
tut1_intro_ise_2011_Jul_13_1555.bof
tut2_10gbe_2011_Jul_05_1639.bof
tut3_r_spec_2048_r105_2011_Jul_06_1251.bof
tut4_poco_wide_10_r314_2011_Jul_06_1525.bof
tut5_gpu_spec_2011_Jul_06_1707.bof
root@roach030172:/boffiles#

9. root@roach030172:/boffiles# **./[TUT1_BOF_FILE] &**
[1] 230
root@roach030172:/boffiles#

Our FPGA is now programmed and we have our prompt back!
Look at the output ie LED on the ROACH board is blinking !!!

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

10. root@roach030172:/boffiles# **ps aux**

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
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@roach030172:/boffiles#
```

Notice that PID “230” (yours may be different) is our process. We can now navigate to the proc directory which contains our software registers.

11. root@roach030172:/boffiles# **cd /proc/230/hw/ioreg/** #Pl chage the PID nuber if different.
12. root@roach030172:/proc/230/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_ctrl1
-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@roach030172:/proc/230/hw/ioreg#
```

Now you can see all our software registers. We have a, b, counter_ctrl, counter_value and sum_a_b as expected. 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_rev_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 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).

COUNTER

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

1. root@roach030172:/proc/230/hw/ioreg# **hd Counter_value**
00000000 00 00 00 00 |....|
00000004
root@roach030172:/proc/230/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.
2. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\001" > Counter_ctrl1**

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_ctrl1. Now let's relook at the counter value...

3. root@roach030172:/proc/230/hw/ioreg# **hd Counter_value**
00000000 da 12 42 de |..B.|
00000004

4. root@roach030172:/proc/230/hw/ioreg# **hd Counter_value**
00000000 dd 32 7c 3c |..w.<|
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...

5. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\002 Counter_ctrl1**
6. root@roach030172:/proc/230/hw/ioreg# **hd Counter_value**
00000000 00 00 00 00 |....|
00000004
root@roach030172:/proc/230/hw/ioreg#

ADDER :

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

7. root@roach030172:/proc/230/hw/ioreg# **hd a**
00000000 00 00 00 00 |....|
00000004

8. root@roach030172:/proc/230/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
Note : Inpu the data as OCTAL by using \0 eg \005 for decimal 5 & \014 for 12.

9. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\005" > a**
10. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\014" > b**
11. root@roach030172:/proc/230/hw/ioreg# **hd a**
00000000 00 00 00 05 |....|
00000004

12. root@roach030172:/proc/230/hw/ioreg# **hd b**
00000000 00 00 00 0c |....|
00000004
13. root@roach030172:/proc/230/hw/ioreg# **hd sum_a_b**
00000000 00 00 00 11 |....|
00000004

Note : Inpu the data as HEXADECIMAL by using \x. eg \005 for decimal 5 & \x0c for 12.
14. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\x05" > a**
15. root@roach030172:/proc/230/hw/ioreg# **echo -n -e "\000\000\000\x0c" > b**
16. root@roach030172:/proc/230/hw/ioreg# **hd a**
00000000 00 00 00 05 |....|
00000004
17. root@roach030172:/proc/230/hw/ioreg# **hd b**
00000000 00 00 00 0c |....|
00000004
18. root@roach030172:/proc/230/hw/ioreg# **hd sum_a_b**
00000000 00 00 00 11 |....|
00000004
root@roach030172:/proc/230/hw/ioreg#
19. Press CTRL + A and then Q to QUIT from the minicom. Then exit from root.
Great! Exactly as expected.

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 PROGRAMMING THE FPGA 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, while still being easy to program clients and servers.

Note : steps 1 , 2 & 3 are need not to be done here again .

1. Copy the bof file to be programed which is compiled by you in the directory **"[FPGA_PROG_BOF_DIR]"** after changing the permissions of the file.

eg. for the bof file

[TUT1_BOF_FILE] in the area [STD_BOF_DIR]

$ **chmod a+x [STD_BOF_DIR]/[TUT1_BOF_FILE]**

$ cp [STD_BOF_DIR]/ [TUT1_BOF_FILE] [FPGA_PROG_BOF_DIR]

Note : All these cable connections and entries in the /etc/* files of the workshop PCs done.

2. Connect the Ethernet cable to J25 port of the ROACH board from the PCs eth1 port. /etc/ethers file should have mac address and corresponding ip address. In the /etc/network/interfaces file eth1 should be configured. And in the file /etc/hosts ip address and corresponding roach board(host) name entry to be done.

3. **Switch ON the ROACH BOARD.**Refer the file **"[DOC_DIR]/[ROACH_BOOT_PROC_FILE]"** for the complete boot procedure of the ROACH BOARDs. But not required for this purpose.

4. From the PC connect to the ROACH via net using the command ;

telnet <ROACH IP assigned> <port #> note : port # is 7147 decided by protocol.

eg. $ **telnet 192.168.100.72 7147**

Trying 192.168.5.251...
Connected to 192.168.5.251.
Escape character is '^]'. : To quit from this Press "CTRL" + "]" (bracket])
then enter. And then quit from it.
#version poco-0.1
#build-state poco-0.0a271150300

5.**?listbof** : To see the BORPH (*.bof) files on the ROACH for programming FPGA. Pl. notice that commands are starting with ? in the front. The files below may be different in your PC!

imh_tut1a_2011_Jul_18_1439.bof
tut1_intro_ise_2011_Jul_13_1555.bof #This is file to be programmed in the FPGA!
tut2_10gbe_2011_Jul_05_1639.bof
tut3_r_spec_2048_r105_2011_Jul_06_1251.bof
tut4_poco_wide_10_r314_2011_Jul_06_1525.bof
tut5_gpu_spec_2011_Jul_06_1707.bof
!listbof ok 6

6.**?progdev <borph file>** : This puts the bof program file in the FPGA.

eg. **?progdev [TUT1_BOF_FILE]**

!progdev ok 231

9 Communicating with your FPGA process in KATCP

1. **?listdev**
#listdev sum_a_b
#listdev b
#listdev a
#listdev Counter_value
#listdev Counter_ctrl1
#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.

2. **?wordwrite a 0 0x02**
!wordwrite ok
3. **?wordwrite b 0 0x07**
!wordwrite ok
4. **?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.

10 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.