iBOB ADC Tutorial

CASPER Reference Design

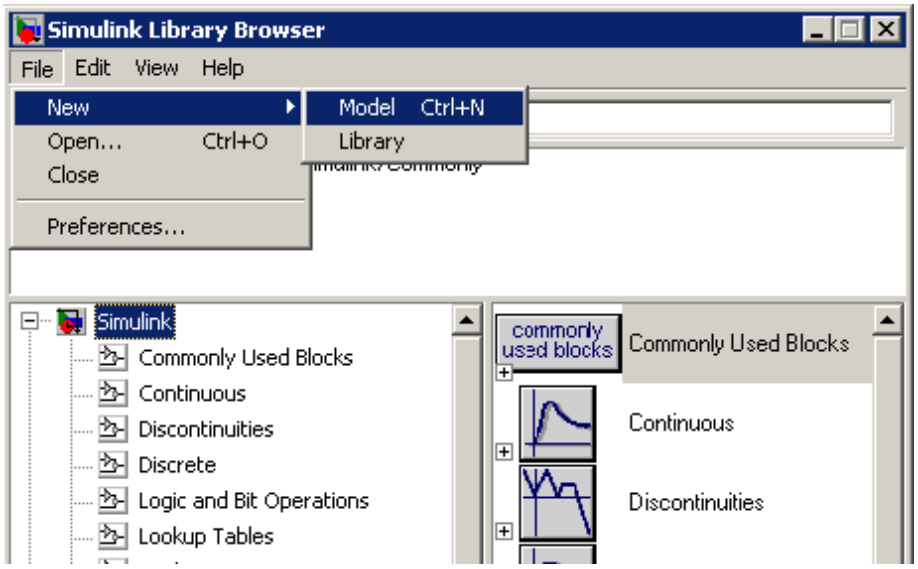
Author: Griffin Foster
March 24, 2009 (v1.0)

Hardware Platforms Used: iBOB, iADC
FPGA Clock Rate: 100 MHz
Sampling Rate: 400 MHz
Software Environment: TinySH

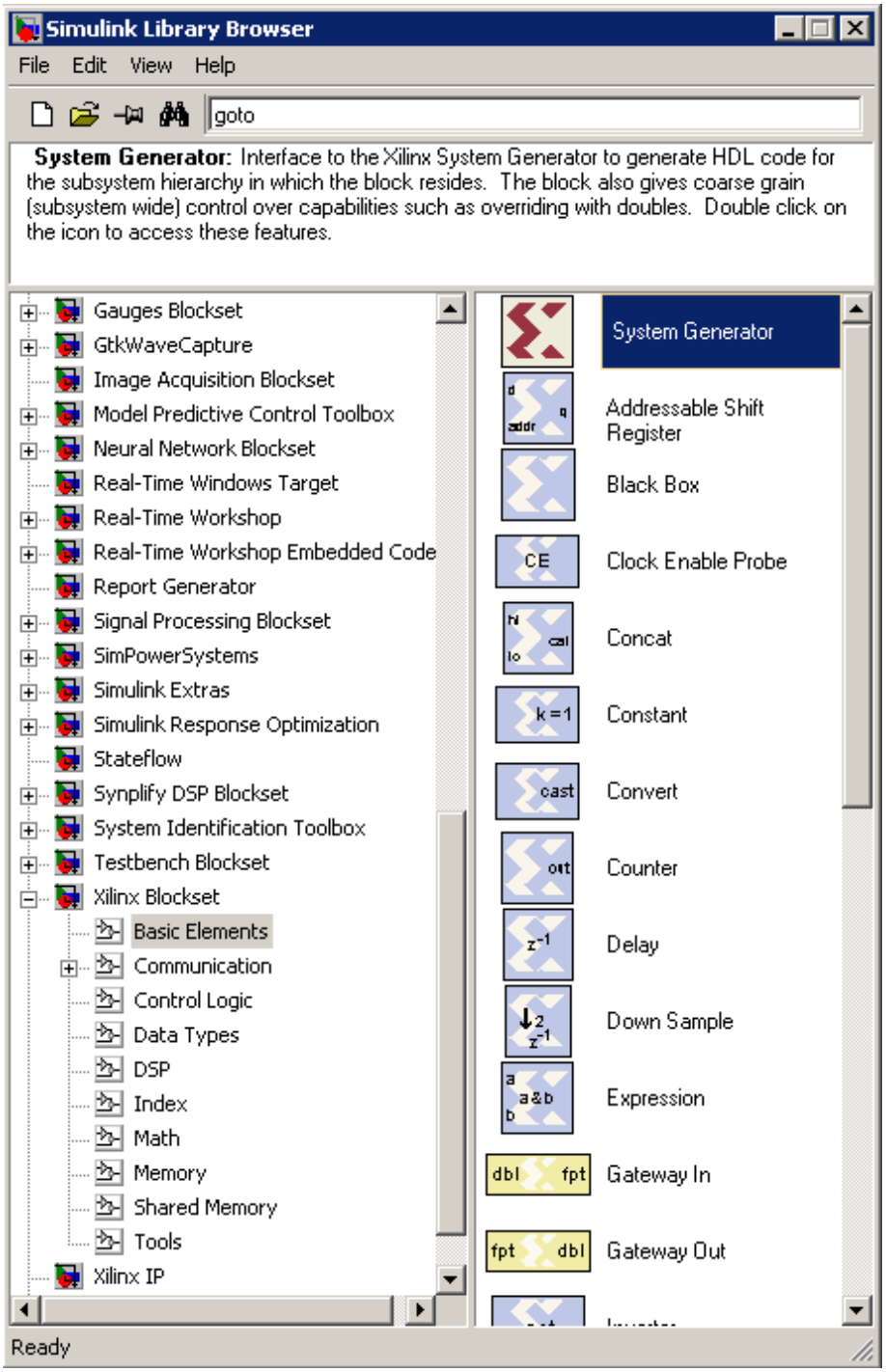
This tutorial walks through a design which uses an iBOB and iADC board to record time domain data into a BRAM and calculate the summed power of the samples. The first section of this tutorial walks through the Simulink design. We then look at simulating the design and loading it onto a board to test. In the last section we go through using lightweight IP to interface with the iBOB.

Start a New Design

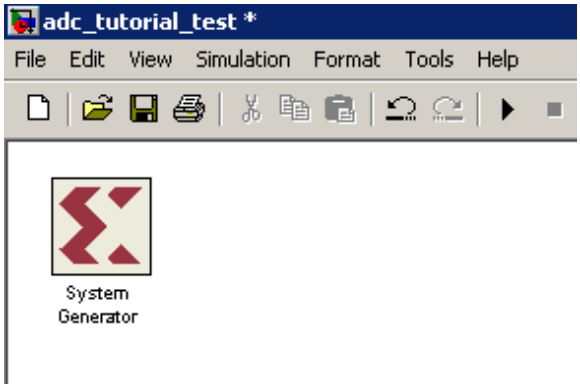
Start using Matlab, we will be using MLib Devel 7.1 for this design. Run Simulink by typing `simulink` in the Matlab command prompt. To create a new design, select *File → New → Model* from the Simulink Library Browser. Save as "*adc_tutorial.mdl*"

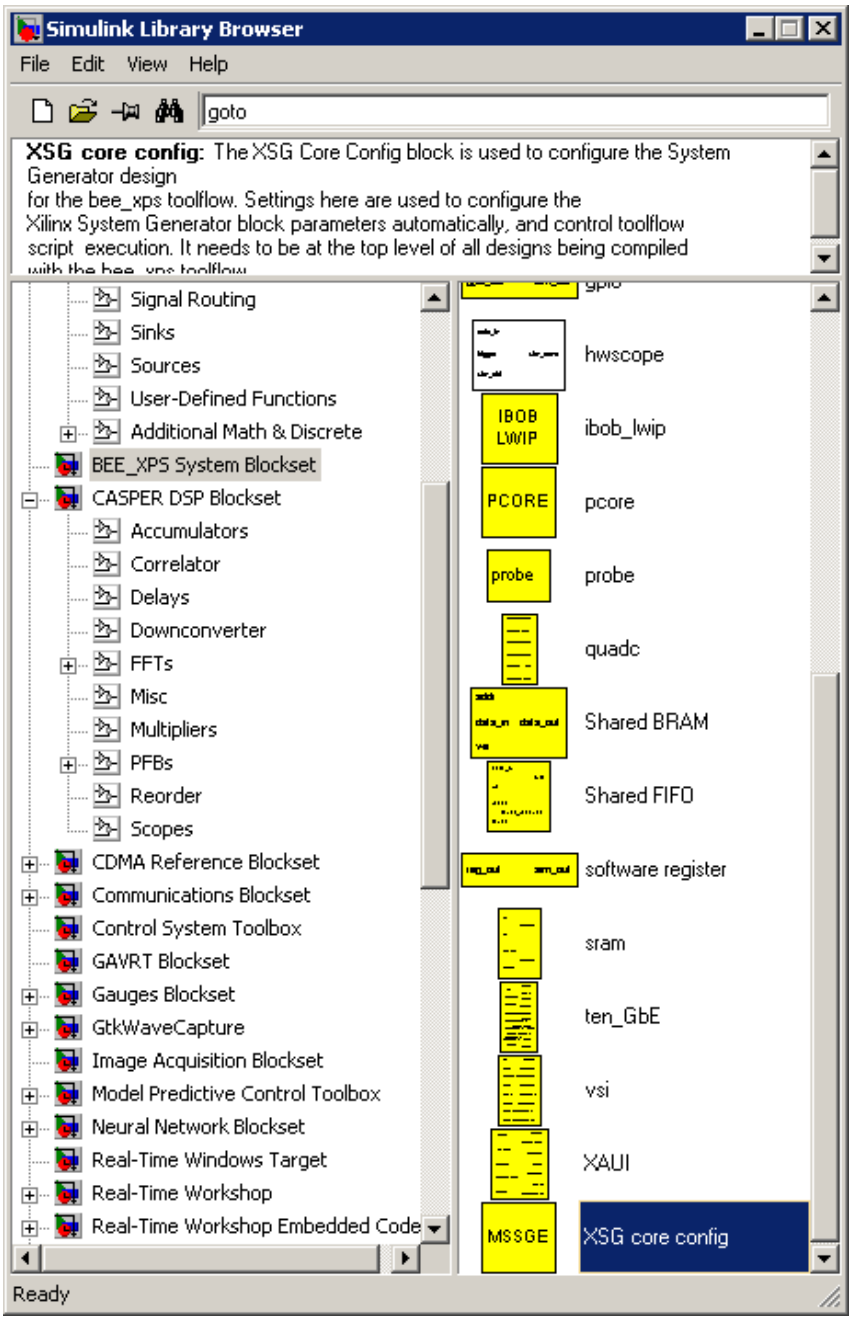


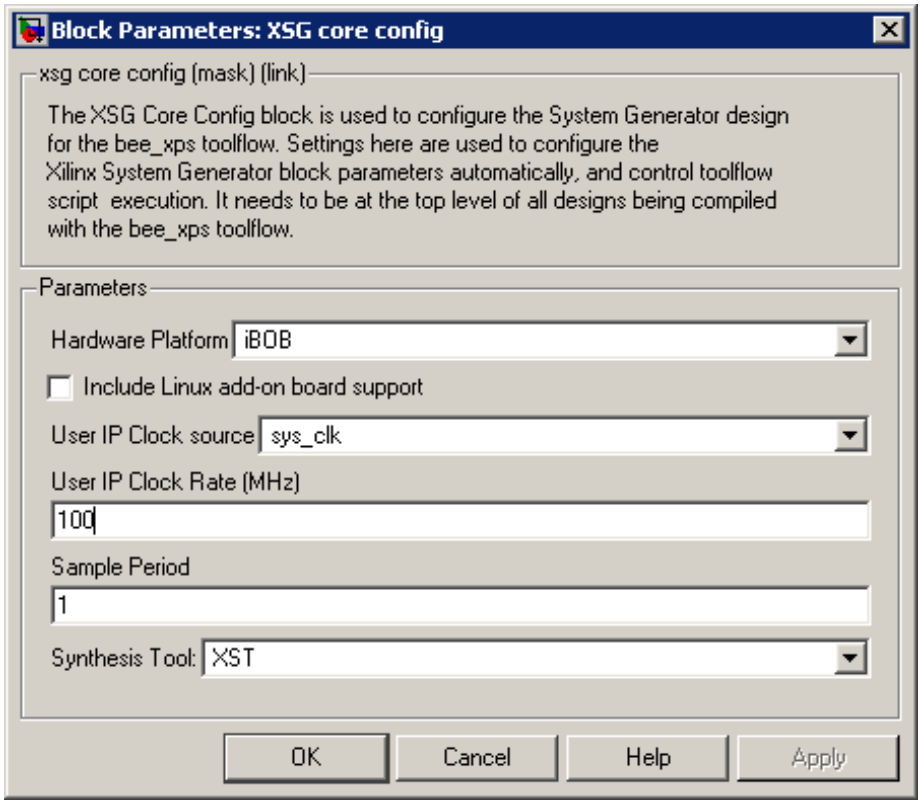
We will be using a number of blocksets in this design, you may want to load BEE_XPS System, CASPER DPS, and Xilinx. Since we will be using Xilinx blocks the first block to place is the **System Generator** block which is in the **Xilinx** blockset from **Basics Elements**.



The System Generator block does not need any modifications, it just needs to be in the design. Next, insert an **XSG core config** block from the **BEE_XPS System Blockset** library. This block is required by the *bee_xps* toolflow, and is used to define the compilation parameters for the design. The settings in this block are used to automatically define the settings of the System Generator block during system generation.

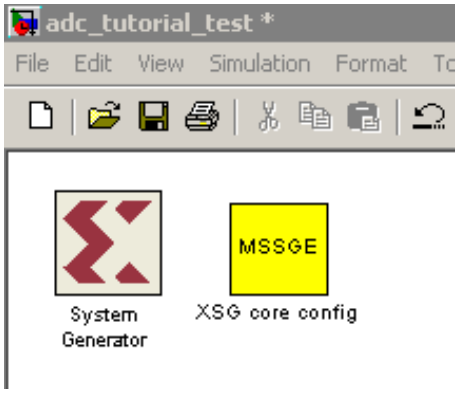

adc_tutorial_test *
File Edit View Simulation Format Tools Help
System Generator


Simulink Library Browser
File Edit View Help
goto
XSG core config: 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
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete
BEE_XPS System Blockset
CASPER DSP Blockset
Accumulators
Correlator
Delays
Downconverter
FFTs
Misc
Multipliers
PFBs
Reorder
Scopes
CDMA Reference Blockset
Communications Blockset
Control System Toolbox
GAVRT Blockset
Gauges Blockset
GtkWaveCapture
Image Acquisition Blockset
Model Predictive Control Toolbox
Neural Network Blockset
Real-Time Windows Target
Real-Time Workshop
Real-Time Workshop Embedded Code
hwscope
IBOB LWIP
ibob_lwip
PCORE
pcore
probe
probe
quadc
Shared BRAM
Shared FIFO
software register
sram
ten_GbE
vsi
XAUI
MSSGE
XSG core config
Ready

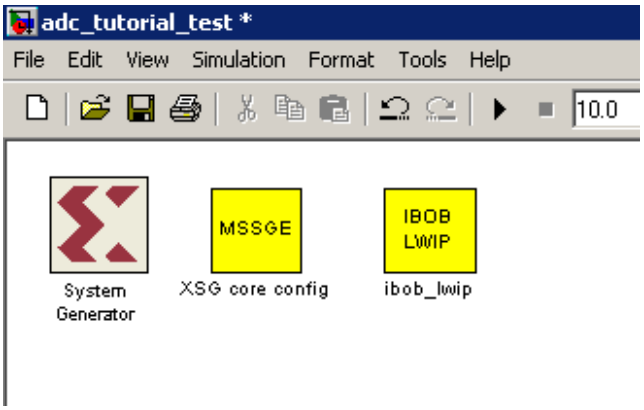


Double-click on the XSG Core Config block to set the parameters:

- Set **Hardware Platform** to *iBOB*
- *Uncheck* **Include Linux add-on board support**
- Set **User IP Clock source** to *sys_clk*
- Set **User IP Clock Rate** to *100MHz*
- Set **Sample Period** to *1*
- Select *XST* as the **Synthesis Tool**
- Click **OK**

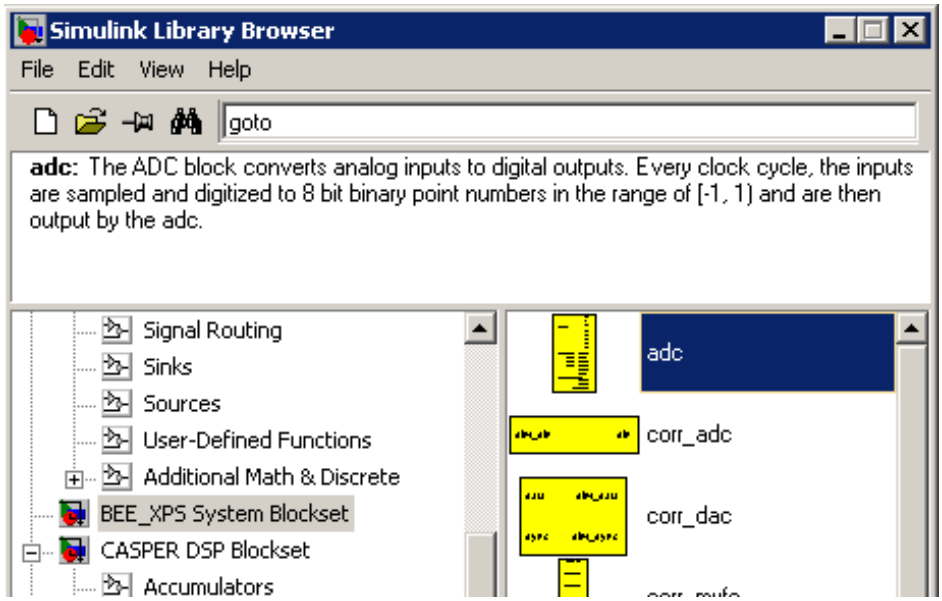


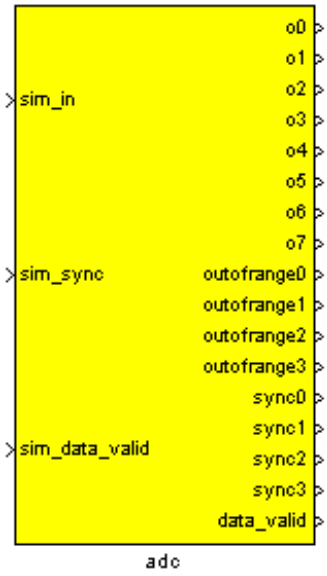
When the design is complete we will be using the lightweight IP interface to interact with the board. To include this in the design add the **ibob_lwip** block from the **BEE_XPS System Blockset**.



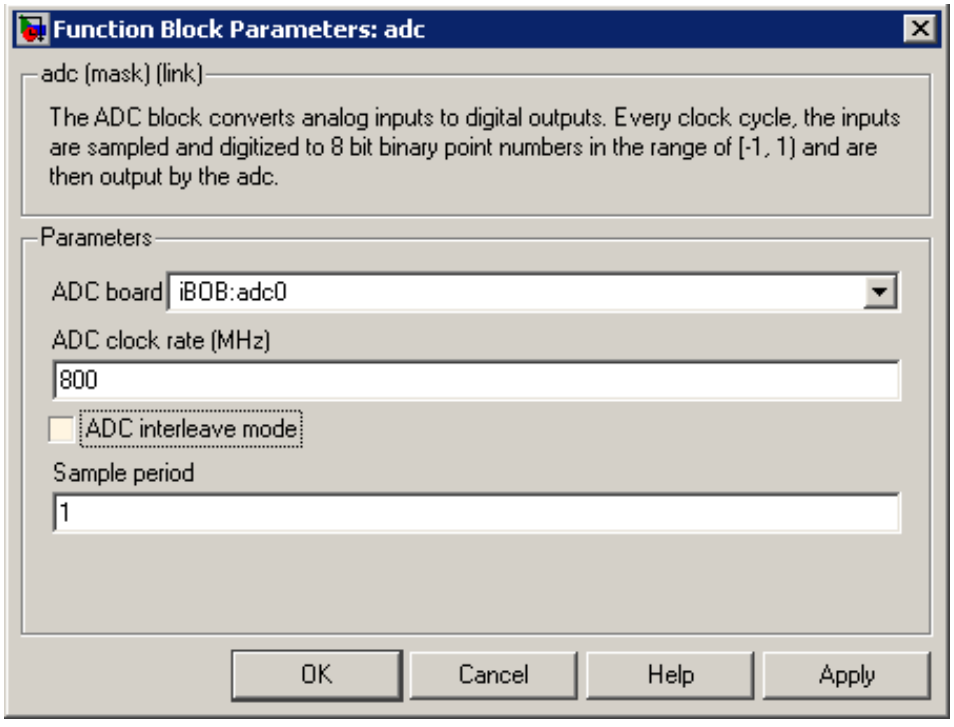
ADC Block

Now that we have the basic blocks down we are ready to start our ADC interface design. This iBOB design uses the **adc** block from the **BEE_XPS System Blockset**. We will only be using one ADC, thought the iBOB can physically interact with two(if you have a design that uses two ADCs, add two **adc** blocks to your design).





There are two modes to the ADC, we can use the ADC in interleave mode to increase the sample speed by sacrificing one of the inputs. Since we will be using a low clock speed for this design then we do not need to use it's interleave mode. Double-click on the ADC block and uncheck the *ADC Interleave Mode* option.



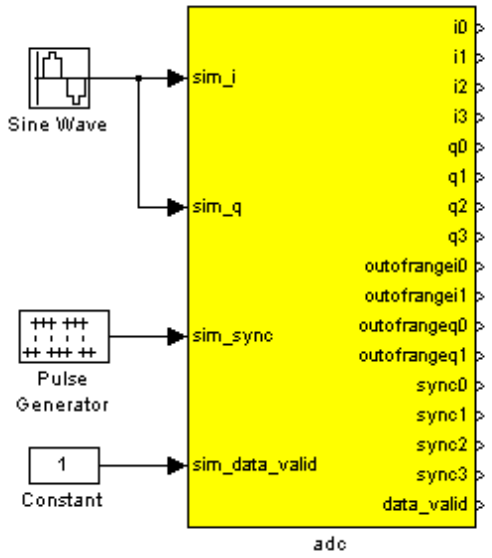
Now the ADC block will have 4 inputs: *sim_i, sim_q, sim_sync, sim_data_valid.*

Quick Note about Simulink: a block will have inputs on the left and outputs on the right. The outputs do not need to be hooked up to anything but the inputs of a block do.

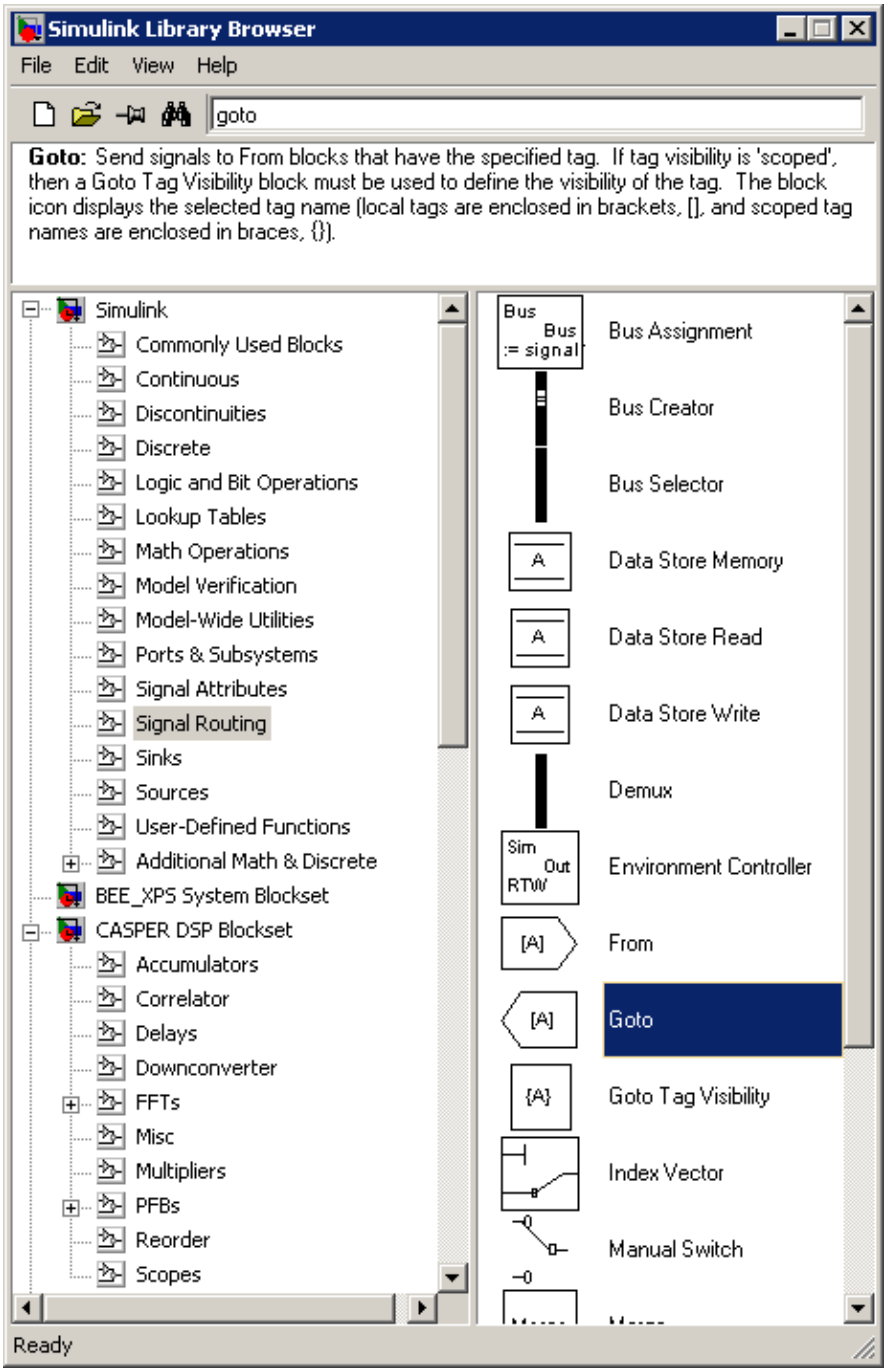
For the inputs to the ADC we will hook up some simulated data blocks.

Another Quick Note about Simulink: In the designs we create we take blocks from various block sets. But not all the blocks are compiled into the final FPGA bitstream. Many of the blocks are used to simulate the final design. Such blocks as **Band-limited White Noise**, **Simulink Constant**, and **Sine Wave** are useful in simulating the output of some design logic but will not be part of the final design.

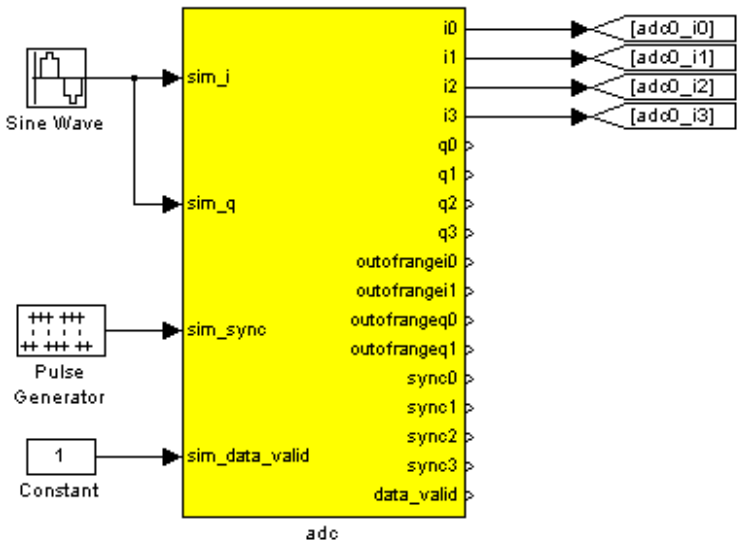
To simulate our design properly we will use a **Sine Wave** in the **Simulink Blockset** from **Sources**. Drag the block to the left of the ADC block and hook it up to both *sim_i* and *sim_q*. Simulink **Tip:** to create multiple paths from one wire hold *Control* and drag from a spot on the wire to create a new wire. Also hook up a **Pulse Generator** in the **Simulink Blockset** from **Sources** to *sim_sync* and a Simulink **Constant** in the **Simulink Blockset** from **Commonly Used Blocks** to *sim_data_valid*.



The ADC has a number of outputs, many of which are useful for reporting system diagnostics. For now we will concentrate on just the ADC data output which is four sample from the Q and I inputs for each clock (there are four samples because the ADC card runs at 4x the clock of the iBOB). Each one of these samples is an 8 bit number, more precisely it is an 8.7 binary number. That is to say that there are 8 bits and the binary point is between the 7^{th} and 8^{th} bit, so the range of output is -1 to 1-(1/127).



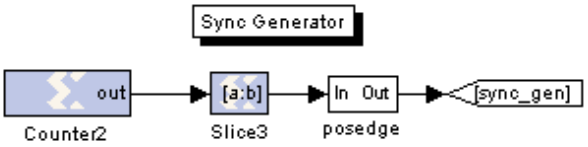
We will only use the I samples for this design, the rest of the outputs will not be used. Attach a **Goto** block in the **Simulink Blockset** from **Signal Routing** to each of the *I* outputs. **Tip:** It is easy to duplicate blocks by dropping one block into the design, then *Control-click* the block and drag, a duplicate block will appear. The **Goto** and **From** blocks are useful fro keeping a clean design but are not necessary. We will use them because there is a summed power section and snap block section of this design which use the same inputs but are separate subsystems.



Each **Goto** block can be named by *double-clicking* the block and renaming the tag, for this design we used *adc0_i0…adc0_i3* for tag names. To use a **From** block *double-click* and rename the tag to the name of the Goto tag you want to use. The ADC subsystem of our design is complete, we now move onto a sync generator.

Sync Generator

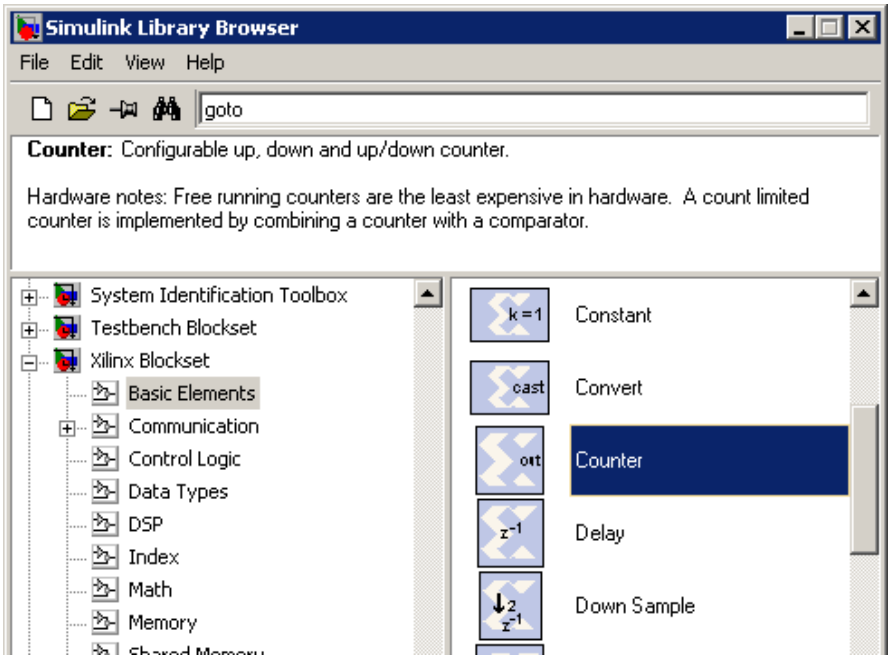
It is often useful to have a sync pulse which keep all the parts of a design well…in sync. A sync pulse is repeatedly generated after a specific number of clock cycles, the sync pulse propagates through the design and will trigger various blocks along the way. In our case we will use it to sum 8192 squared ADC samples(summed power) and write those samples to a BRAM to be read off to a computer later. The sync pulse design is rather easy and only uses a few blocks.

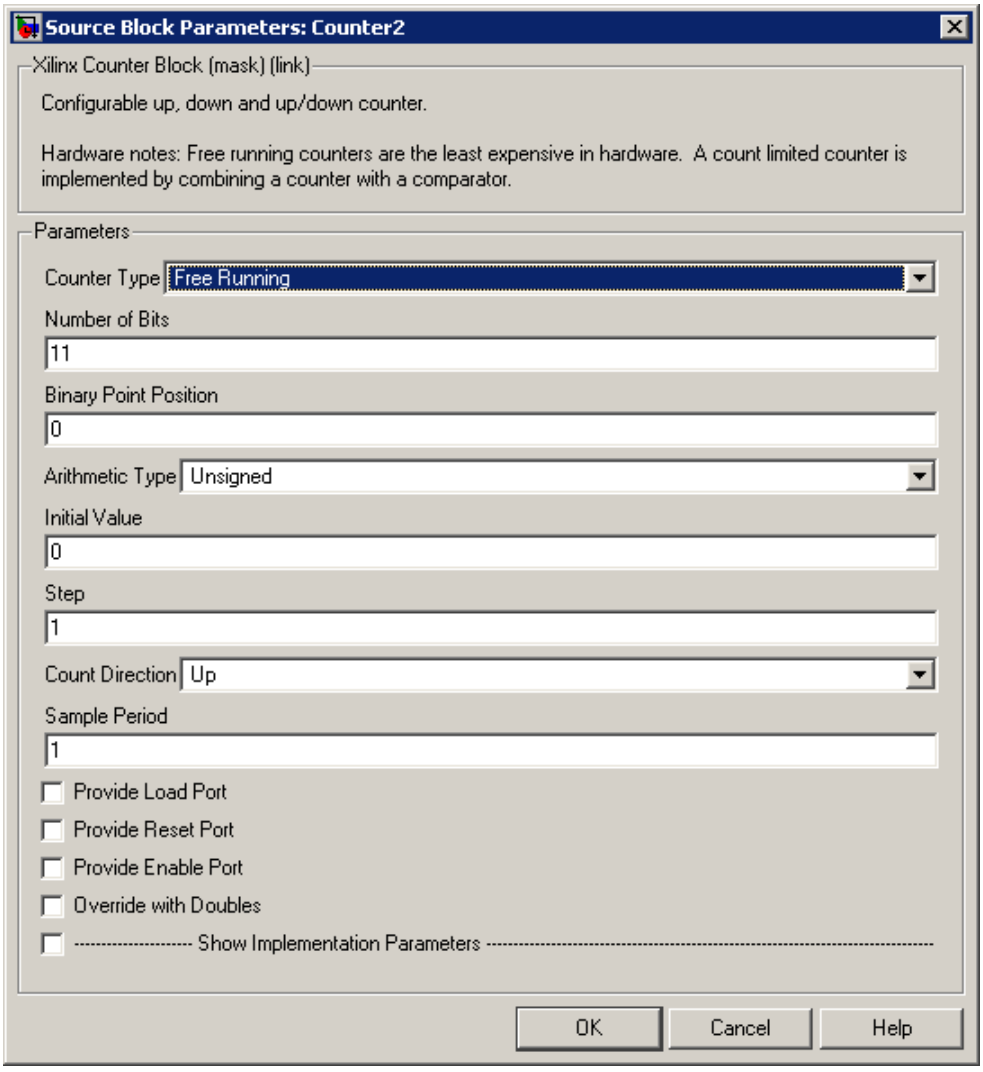


The first block is a counter which will run from 0 to 2047 and then start back at 0 again. The choice of 2048 is based on the number of samples we will write to BRAM and sum up. The **Counter** block is in the **Xilinx** blockset from **Basic Elements**.

Once you drop the block into the design *double-click* on the **Counter** block to set the parameters:

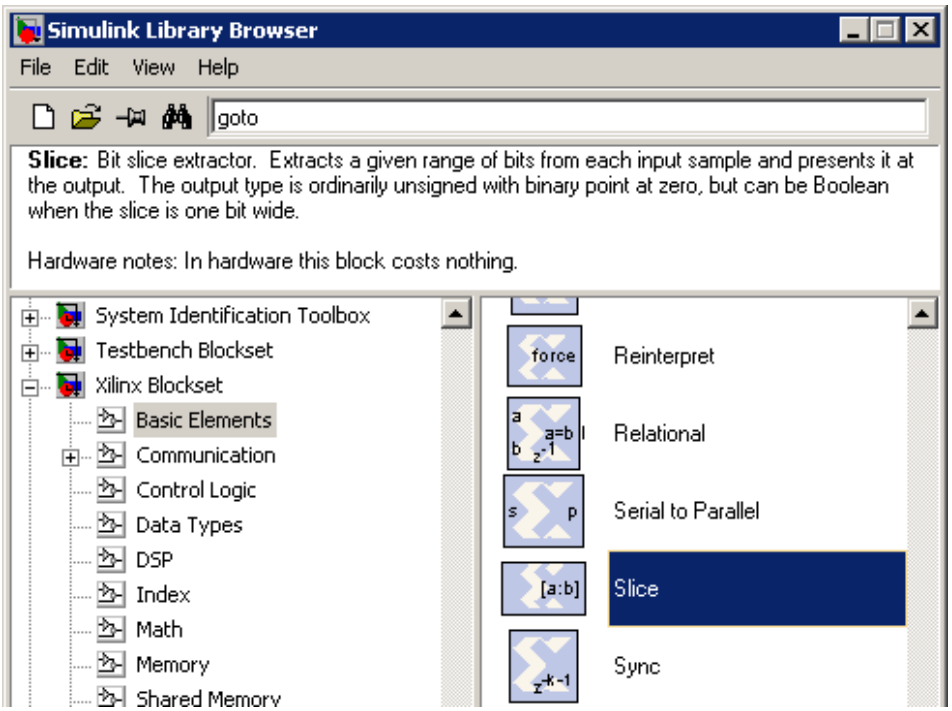
- Set **Counter Type** to *Free Running*
- Set **Number of Bits** to *11*


Simulink Library Browser
File Edit View Help
goto
Counter: Configurable up, down and up/down 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.
System Identification Toolbox
Testbench Blockset
Xilinx Blockset
Basic Elements
Communication
Control Logic
Data Types
DSP
Index
Math
Memory
Constant
Convert
Counter
Delay
Down Sample

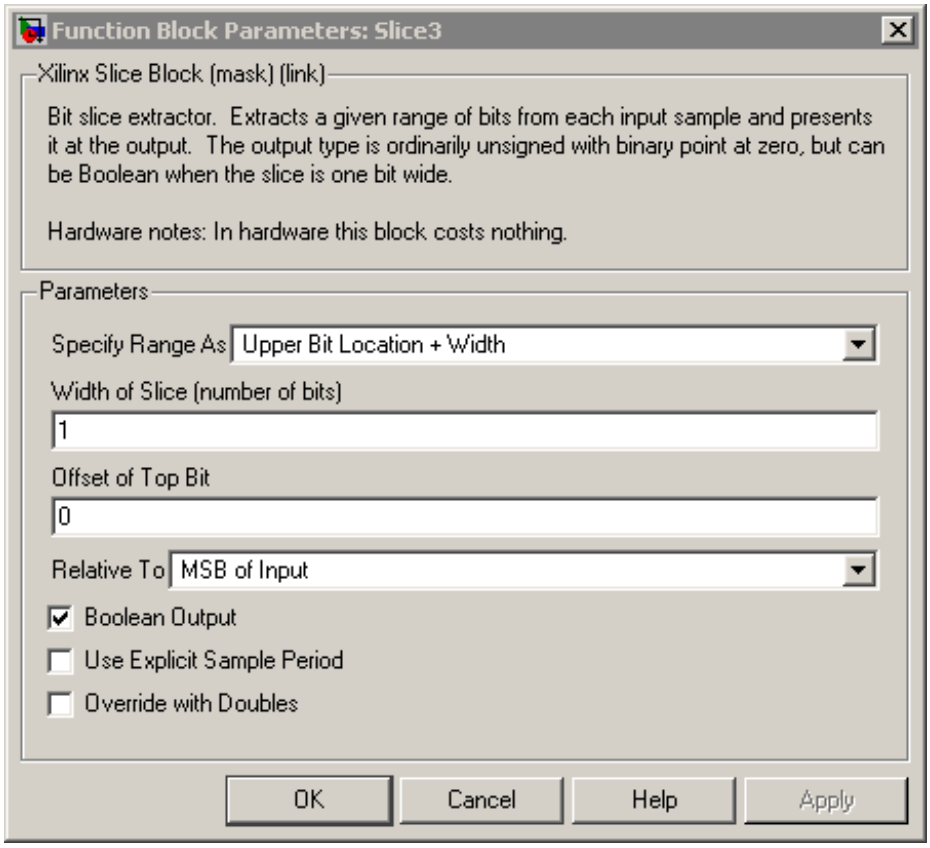

Source Block Parameters: Counter2
Xilinx Counter Block (mask) (link)
Configurable up, down and up/down counter.
Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.
Parameters
Counter Type Free Running
Number of Bits
11
Binary Point Position
0
Arithmetic Type Unsigned
Initial Value
0
Step
1
Count Direction Up
Sample Period
1
Provide Load Port
Provide Reset Port
Provide Enable Port
Override with Doubles
Show Implementation Parameters
OK
Cancel
Help

The counter will increment by one on every clock cycle, we would like to use this fact to send out a single pulse every 2048 clock cycles. There are probably a few ways to do this but the way that is pretty easy so to select the highest bit, the so called MSB(most significant bit), and when it changes from 0 to 1 then to send out a pulse. In every 2048 clock cycles the MSB will only go from 0 to 1 once, when the counter is going from 1023 to 1024. We can send a sync pulse out when we detect this in the counter by using a **Slice** block and **Posedge** block.

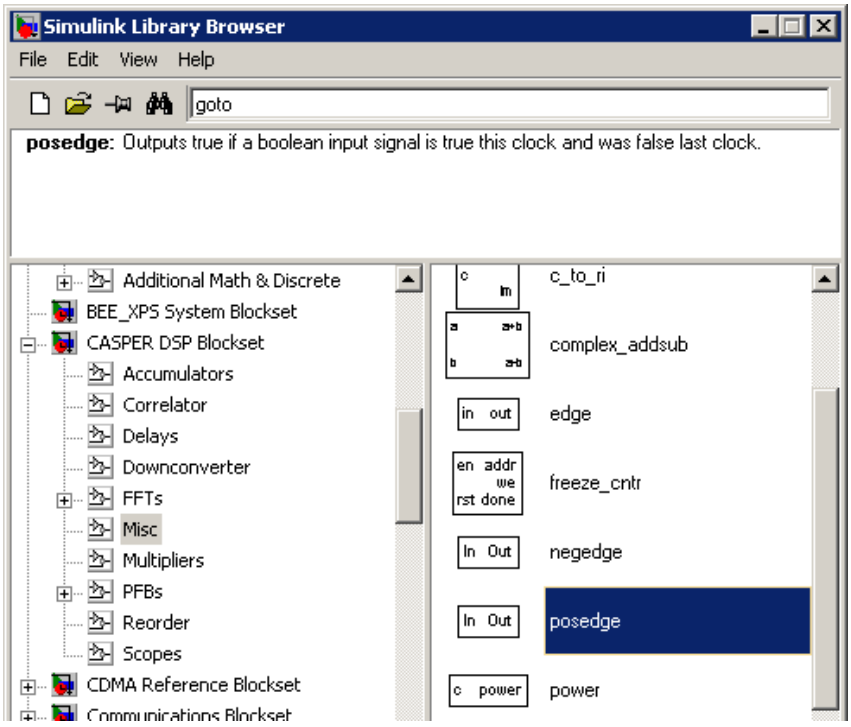
Place a **Slice** block from **Basic Elements** in **Xilinx** into the design and connect the input of the Slice to the output of the Counter.



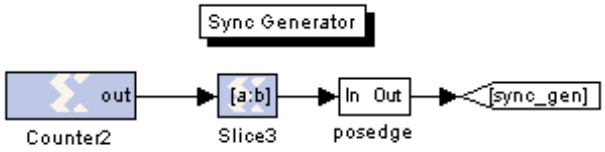
The Slice block outputs an extracted range of bits from its input, in our case we only want the MSB. The output of the default Slice block will do exactly what we want.



The **Posedge** block from **Misc** in **CASPER DSP** blockset will output True when the input singal is True and the previous signal was False. Drag this block into the design and connect it's input to the output of the **Slice** block.



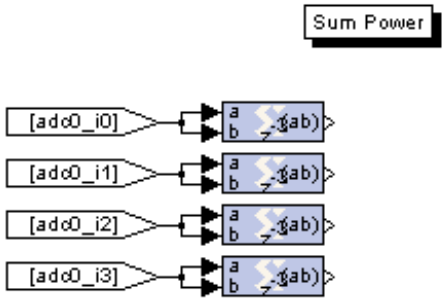
Finally add a **Goto** block from **Signal Routing** in the **Simulink** blockset to the output of the **Posedge** block. Name the tag something useful like *'sync_gen'*. Now that we have a sync generator and the outputs of the ADC we can build a subsystem that accumulates the summed power of the analog inputs over a number of clocks and writes that value to a software register which we can read over a network interface.



Summed Power

The values coming out of the ADC represent the voltage levels of the analog inputs, these values are related to the true voltages by a scale factor. Thus the power of the input signals can be computed by squaring these voltage levels and the true power of a signal is related to this power by another scale factor. For now we will not worry about this scale factor, and look at computing the summed power of the signal. First we use the second part of our Goto/From signal routing layout to place four **From** blocks each named to the corresponding **Goto** block from the ADC. The **From** block is from Signal Routing in the **Simulink** blockset.

The next step is to square these values, the simplest way to do this is to use the **Mult** block in **Math** from the **Xilinx** blockset. From each **From** block attach the output to both inputs of the **Mult** block, use the *Control-click* method to create a extra wire.

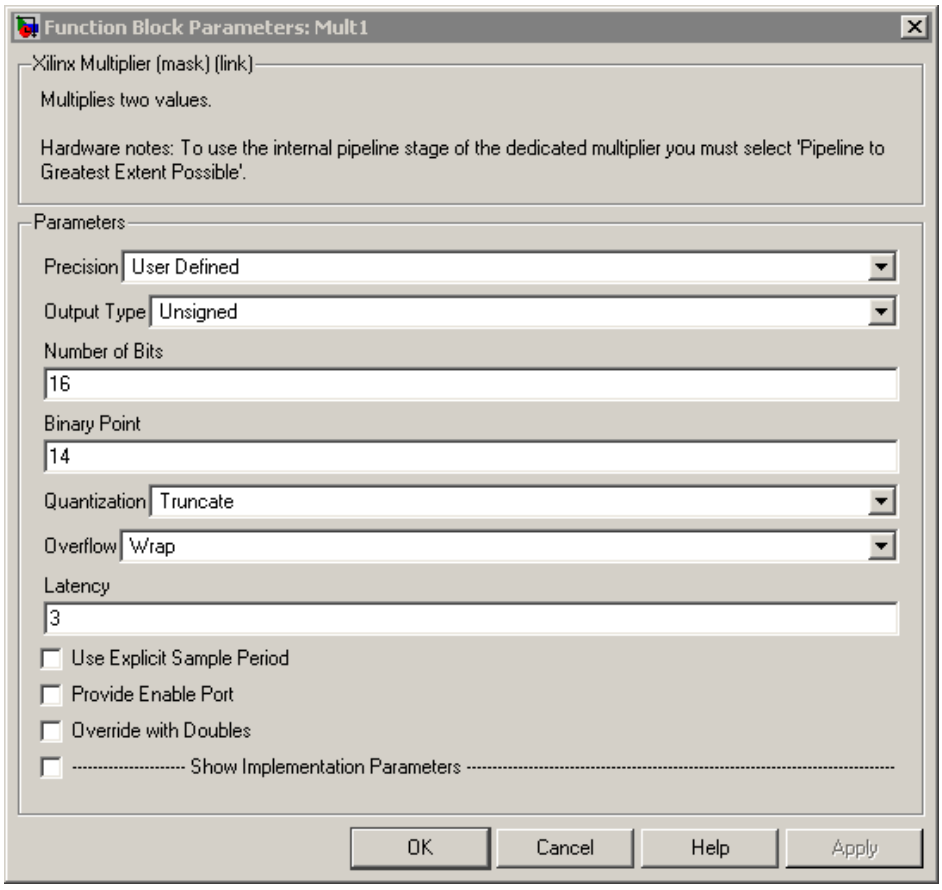


It is important to consider the size of the data as it propagates through a design. The samples from the ADC are of size 8.7 but when data has an arithmetic operation performed on it, such as add or multiply, the output is not always the same size. In the case of a multiplying two 8.7 numbers the output will fit in a 16.14 number. Simulink can usually account for this growth, but sometimes it is best to set a fixed size.

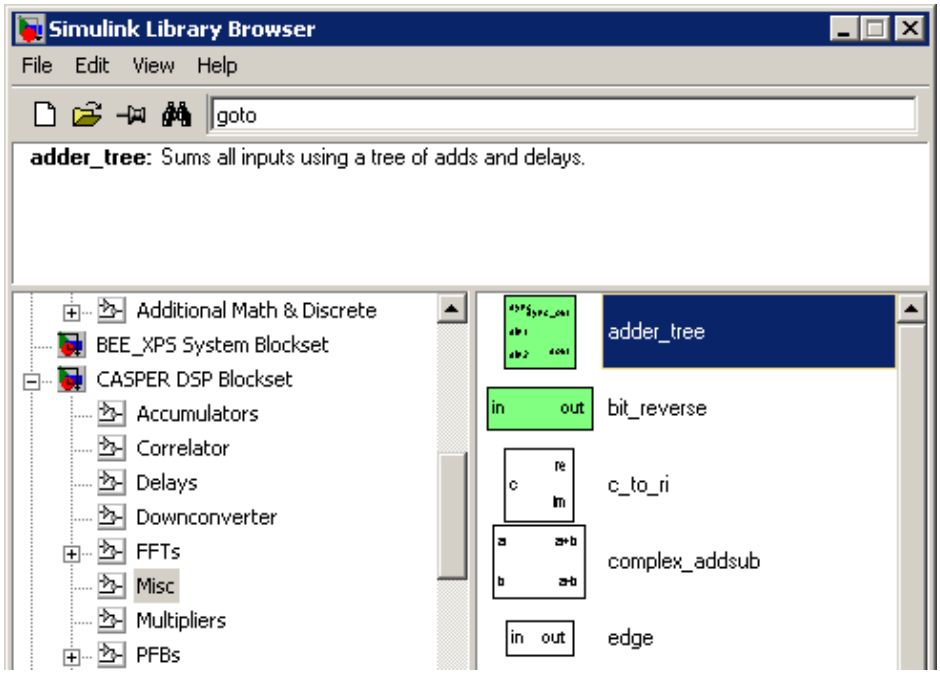
For the four **Mult** blocks we placed we can set this output size by *double-clicking* on each block:

- Set **Precision** to *User-Definded*, the parameters will then expand
- Set **Number of Bits** to 16
- Set **Binary Point** to 14

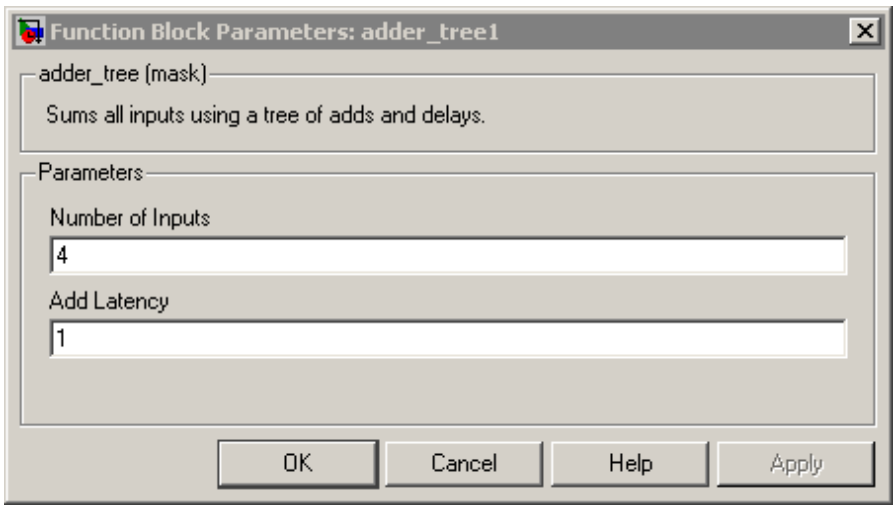
- For **Output Type** select *Unsigned*



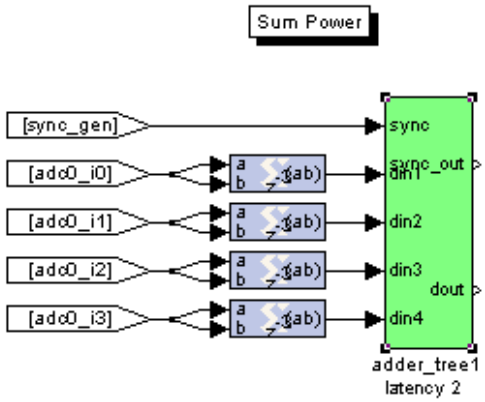
Since we will be computing the sum of the powers we will need to add together the samples. Rather then using four duplicate paths to add up the four ADC samples on every clock we can just add the four powers together after they are squared and then we will only need one path. To add the four samples together we could easily construct an Adder Tree which will add any number of inputs using a series of 2 input Add blocks. But since this is a common operation there is a prebuilt adder tree which will scale based on the number of inputs a design requires. We want to place an **Adder_tree** block from **Misc** in the **CASPER DSP** blockset.



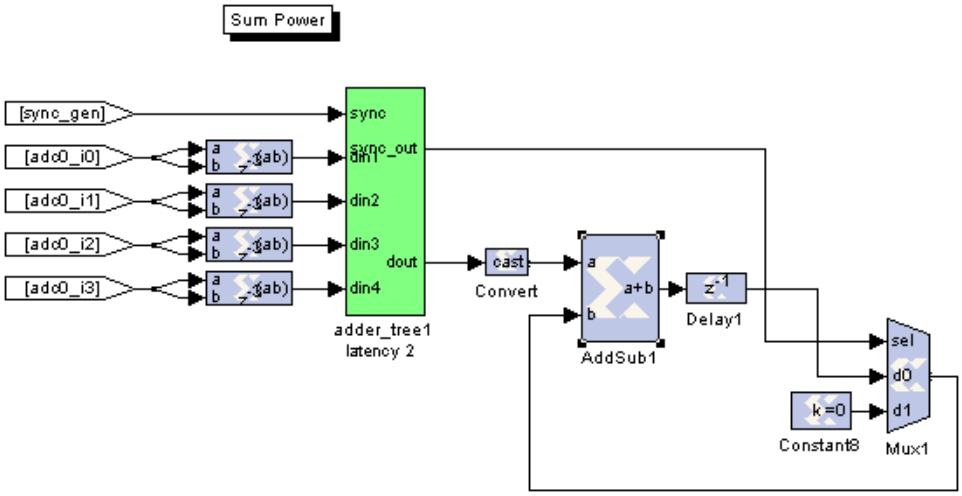
Double-click on the **Adder_tree** block and set **Number of Inputs** to *4*, then click OK.



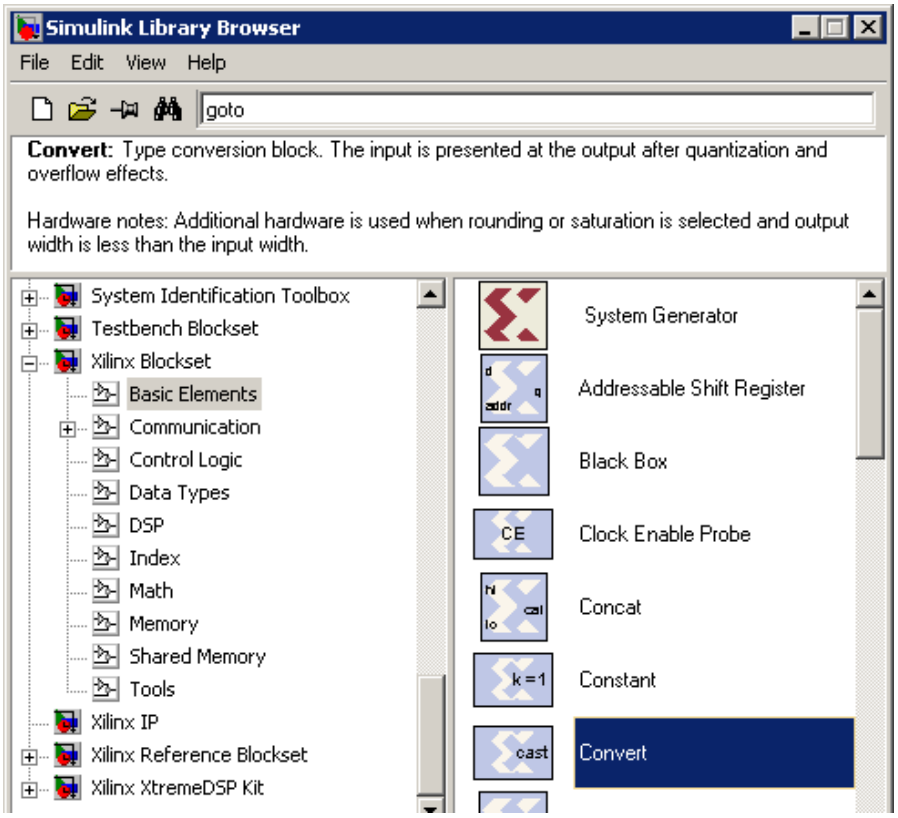
Connect the outputs of the four **Mult** blocks to the inputs of the **Adder_tree** block. The **Adder_tree** block also takes as input a sync, which is fine since we just created a sync generator for just this use. Add a **From** block which is in **Signal** Routing from the **Simulink** blockset, *double-click* on the block and set the tag name to the sync generator tag name. The connect the block to the *sync* input of the **Adder_tree**.



Now that we have summed the four samples we would like to add up this output 2048 times, and output the result to a software register. Once the value is output to a register then the summed power value should be reset to zero to begin summing the next 2048 samples. Lets start with an intermediate goal of summing 2048 samples then resetting the sum to zero. Here is the block layout, we'll step through each block.

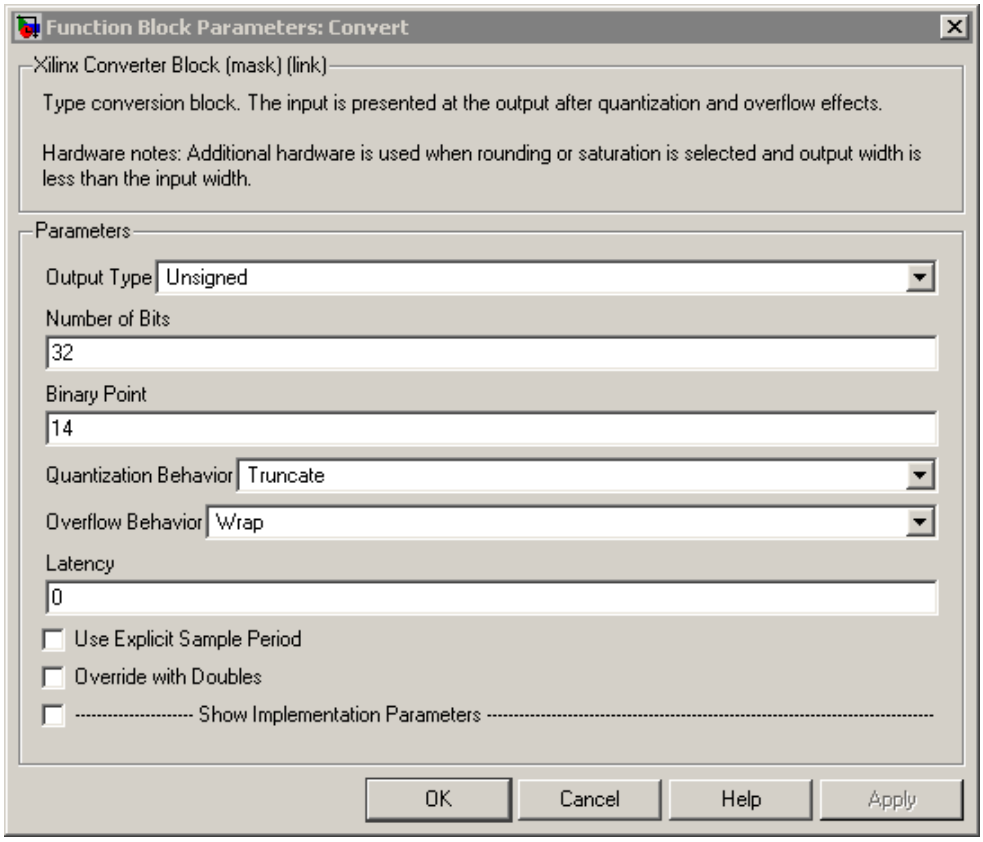


The output from *dout* on the **Adder_tree** block first goes through a **Convert** block which outputs the input data in a new bit size. The rationale for this is that we had a 16.14 number coming into the adder tree, then by adding 4 samples together the bit size has grown to 18.14 (when adding the precision past the binary point does not increase). Then we will add 2048 18.14 number together which will need an 29.14 number to hold the final total. We will actually set the number of bits to 32.14 since we will eventually be writing this value to a 32 bit register anyways. Lets place a **Convert** block from **Basic Elements** in the **Xilinx** blockset and conncet the block's input to the *dout* on the **Adder_tree** block.

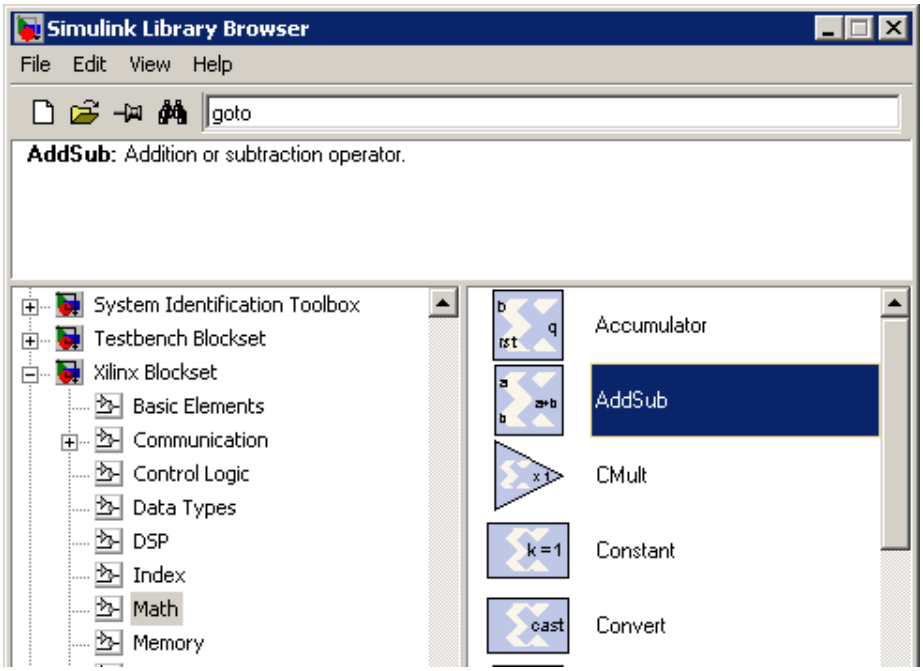


Once you drop the block into the design *double-click* on the **Convert** block to set the parameters:

- Set **Output Type** to *Unsigned*
- Set **Number of Bits** to *32*
- Set **Binary Point** to 14

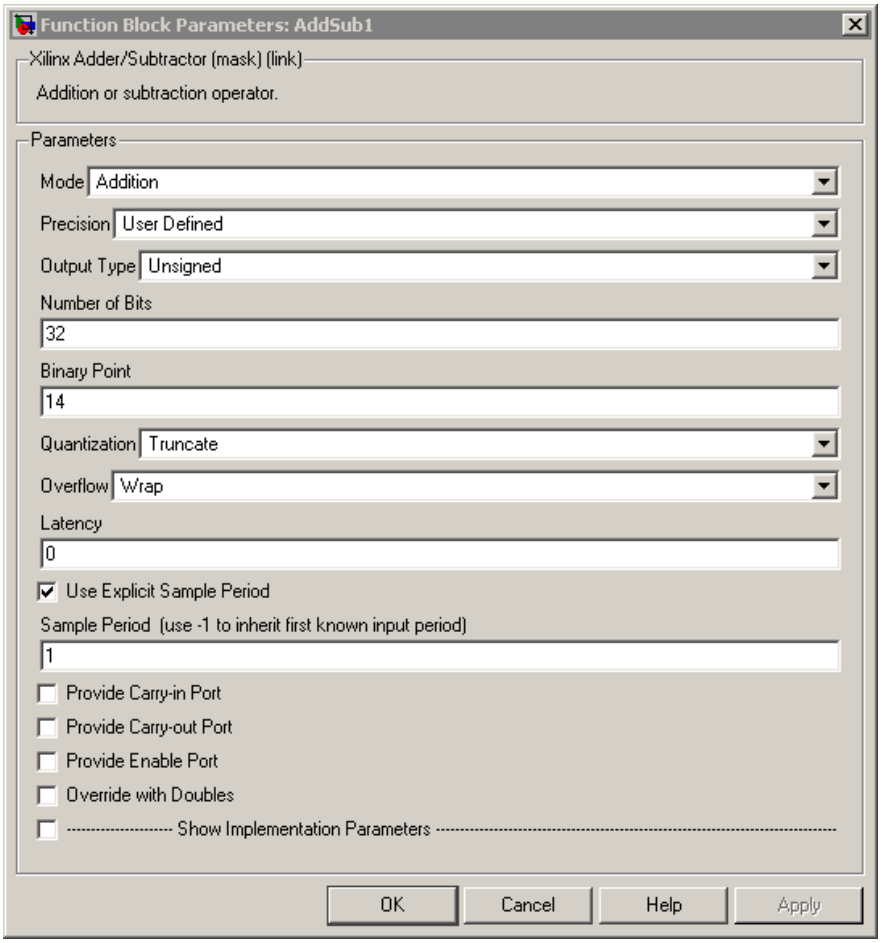


The next block is a **AddSub** block which we will use to add the output of the **Adder_tree** to the values from the previous clock cycles. One of the inputs is the **Adder_tree** output and the other input is the essentially the output of itself. This may seem confusing, but it works as a simple accumulator/summer as long as the other blocks are present. If the output of the **AddSub** block was connected directly to its input then there would be a logic error, the block would need to know the output of itself before it could be computed. What we want to do is add a 1 clock delay to the output before it reaches the input again. That way the **AddSub** block uses the it's output from 1 clock before to compute a new output, thus the block will act as an accumulator. We will worry about the intiail condition case and resetting after 2048 samples shortly.

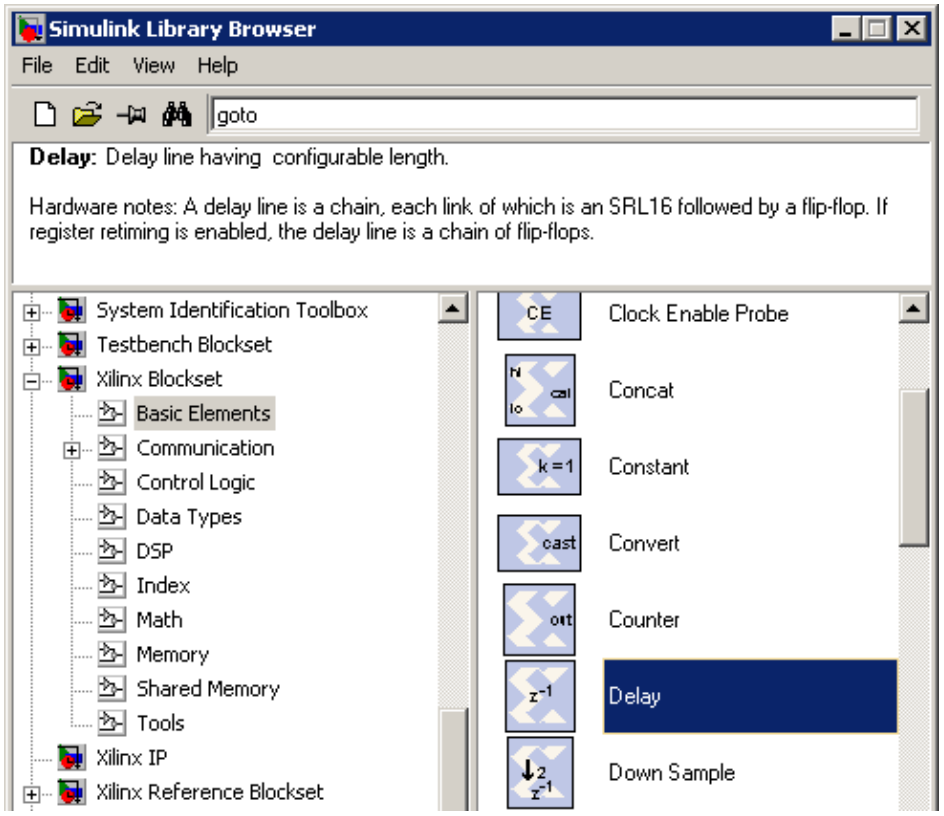


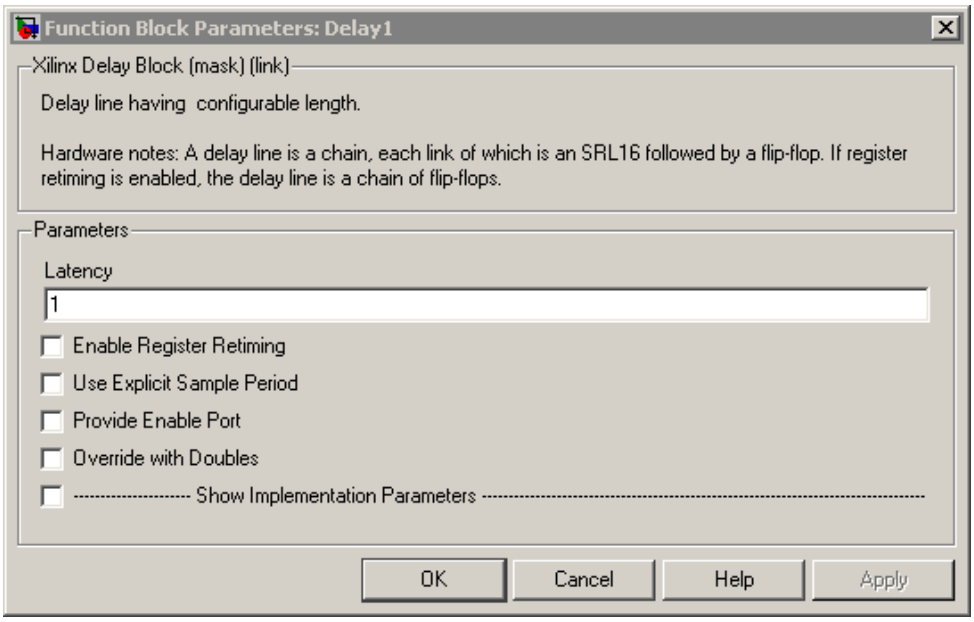
For now lets place an **AddSub** block from **Math** in the **Xilinx** blockset. Connect the first input to the output of the **Convert** block and *double-click* on the block.

- Change **Precision** to *User-Defined*
- Set **Number of Bits** to *32*
- Set **Binary Point** to *14*



Then we delay the output of the **AddSub** block by one clock with a **Delay** block. Drop the **Delay** block which is in **Basic Elements** in the **Xilinx** blockset and connect the input of the **Delay** block to the output of the **AddSub** block. The default setting of the **Delay** block is to delay by 1 clock.

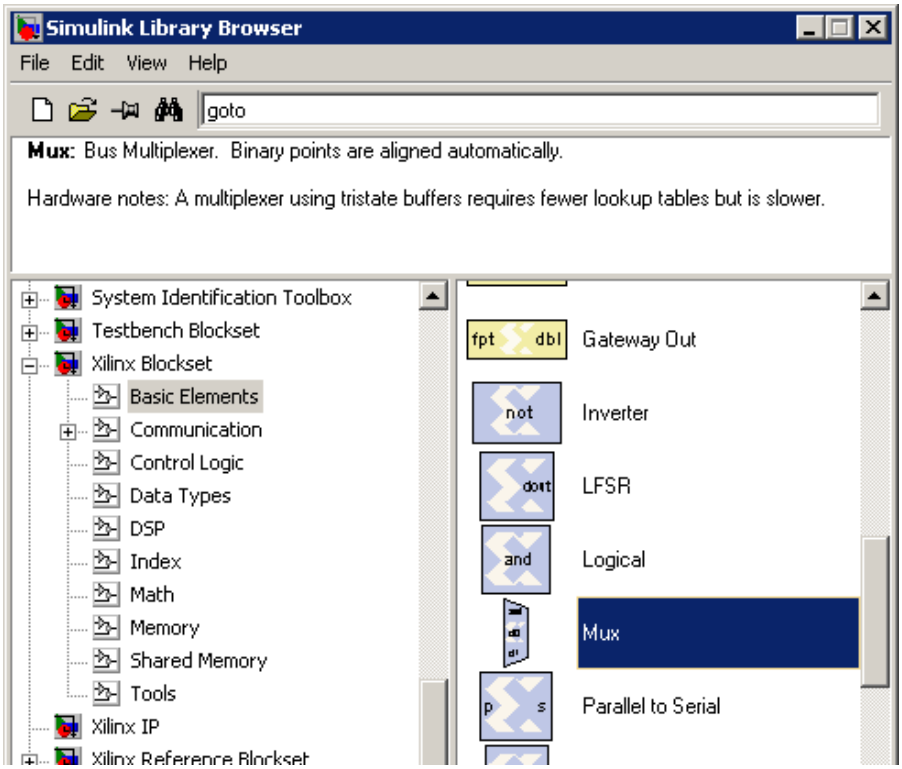


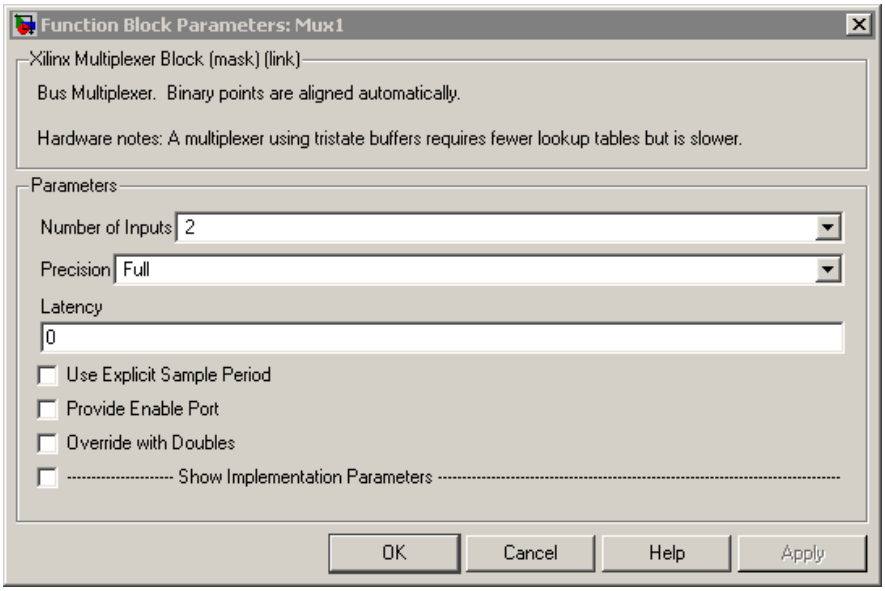


If we hooked up the wire from the **Delay** block to the second input of the **AddSub** block we would have 2 problems. The first is that our accumulator would loop forever and never reset, eventually we would run out of bits. We need to use the our sync generator to set the second input of the **AddSub** block to 0 every 2048 samples, thus resetting our accumulator. The second issue we have is the initial value of the second input to the **AddSub** block need to be 0. We can solve both of these problems by

using a Multiplexer, or Mux. The **Mux** block has three inputs: *sel, d0,* and *d1*. The inputs *d0* and *d1* can be any stream of data. The output od the **Mux** block is dependent on the *sel* input which select which input to output, it is effectively a switch. We can use the **Mux** block to send the output of the **AddSub** block to it's own input 2047 times, then on the 2048 time switch to a 0 which resets our accumulator.

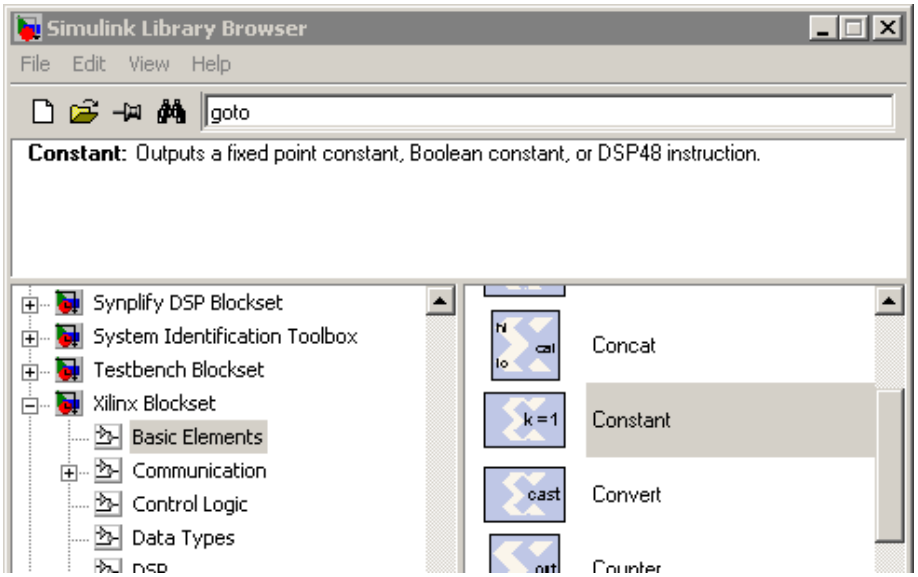
Drop a **Mux** block from **Basic Elements** in the **Xilinx** blockset into the design.





Connect the *sync_out* on the **Adder_tree** to the *sel* input on the **Mux** block. Connect the output of the **Delay** block to the *d0* input of the **Mux** block. To the *d1* input of the **Mux** block we want to add a constant 0 value. For this we need to use the Xilinx blockset constant and not the Simulink blockset

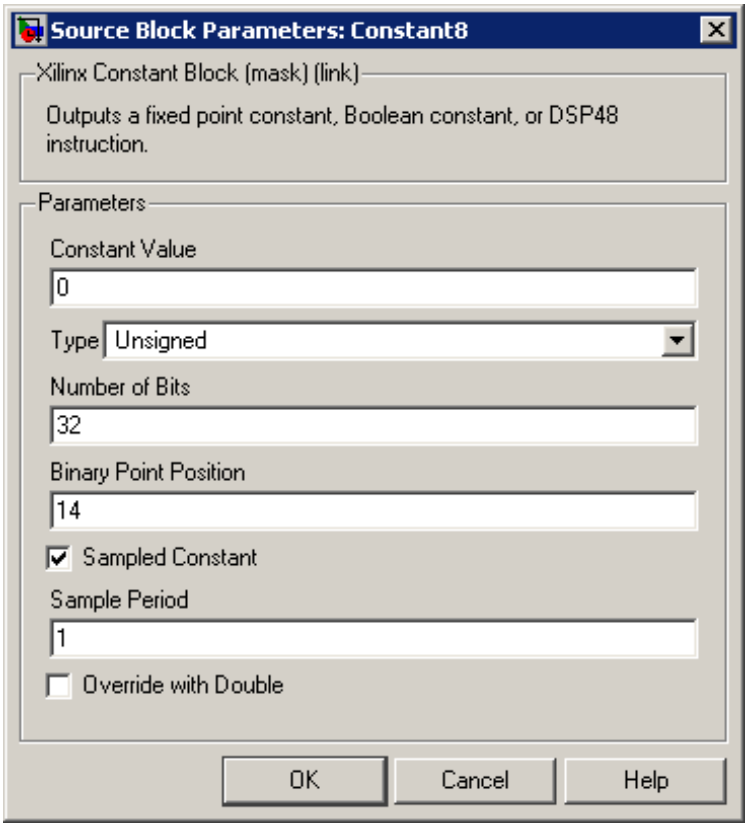
constant. As mentioned earlier, many of the blocks in the Simulink blockset are only used for simulation and will not be compiled into a final bitstream. IN this case we want this constant value to be in in our final design, so we use the Xilinx constant.



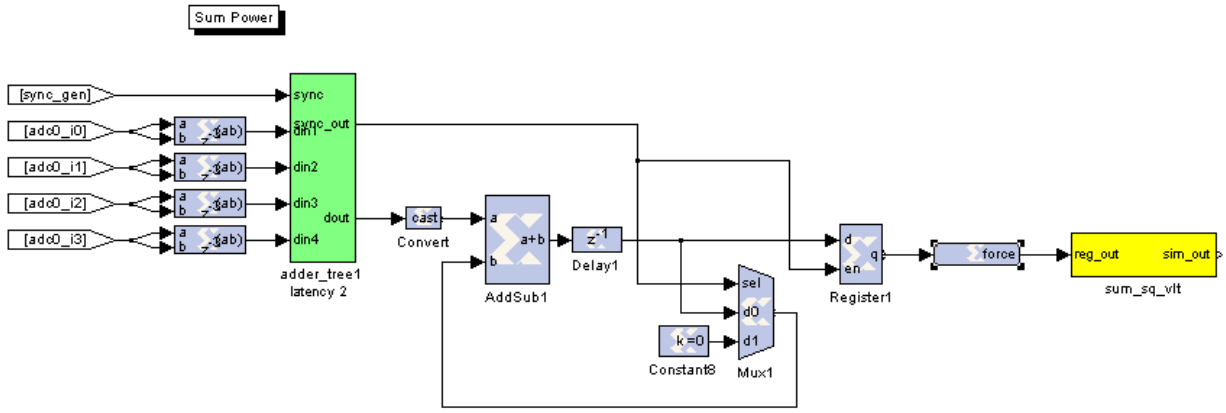
Drop a **Constant** block from **Basic Elements** in the **Xilinx** blockset into the design and double-click on the block:

- Set **Constant Value** to *0*
- Set **Number of Bits** to *32*
- Set **Binary Point** to *14*
- Check **Sampled Constant**

Connect the output of **Constant** block to the *d1* input of the **Mux** block. The output of the **Mux** block can now be connected to the second input of the **AddSub** block. We now have a simple accumulator which adds 2048 values, though we don't have an output…yet.



Now that we have an accumulator we can take it's output and place the value in a software register. But, we have to be careful about what we put in that software register. Here is the final design of the summed power subsystem. We'll go through the last few blocks dealing with the software register.

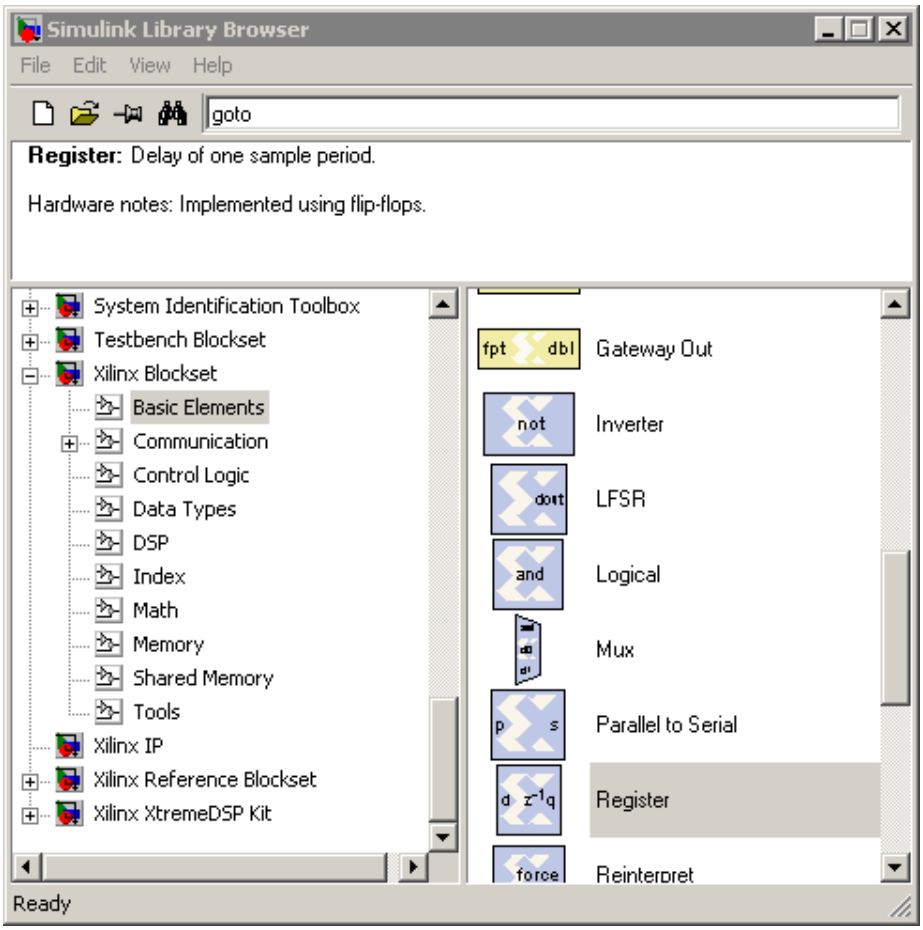


In this diagram the yellow block is the software register, we add another 2 blocks before it to make sure the value is really the 2048 accumulated samples and that value is correctly written to the software register.

Lets start with the register block. You'll notice that the output of our accumulator is a wire branched after the **Delay** block before the **Mux** block. If we wrote this value directly to a software register then every clock cycle the software register would be updated and would not represent a total of 2048 accumulated values, it could be any number of accumulations. We would like the software register to keep one value for 2048 clock cycles then update the register when 2048 new accumulations are made. To do this we use a Xilinx **Register** block which will input all the values of the accumulator

over 2048 cycles, but will only output a value when the sync generator sends a signal to its enable port. The sync generator is again the key to this design.

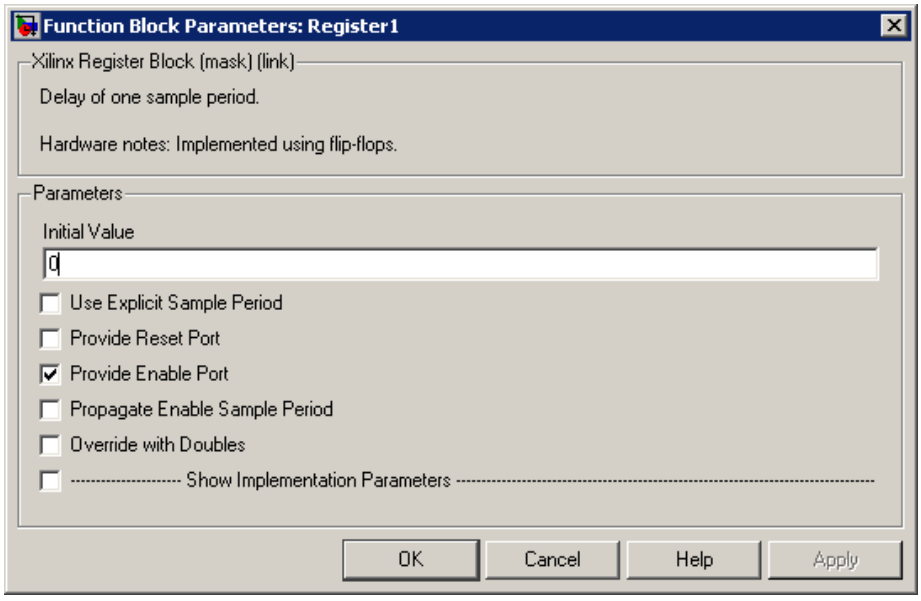
Select the **Register** block from **Basic Elements** in the **Xilinx** blockset.



Double-click on the **Register** block:

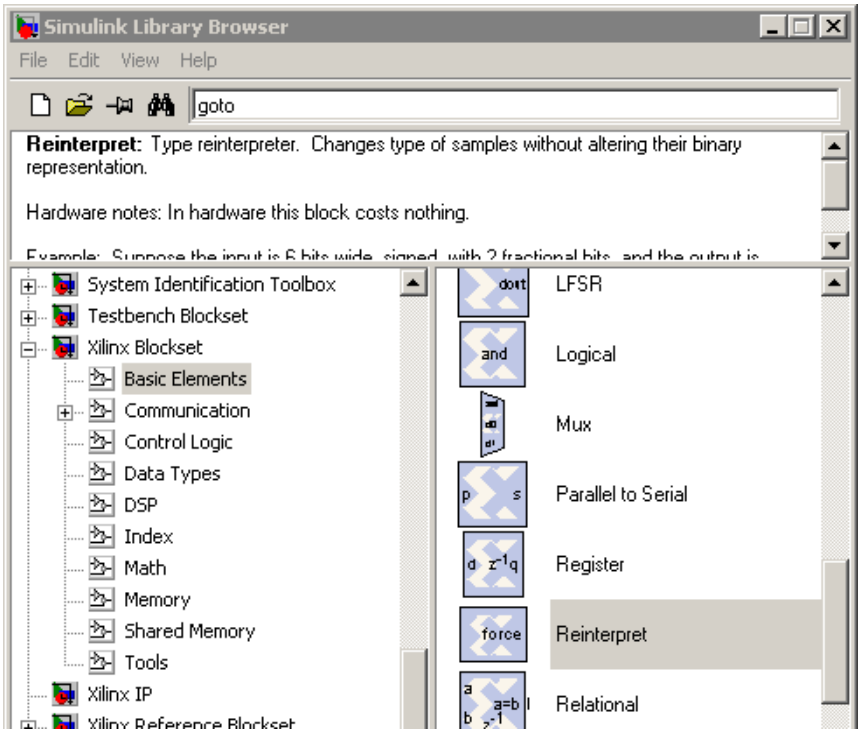
- Check the **Provide Enable Port** option

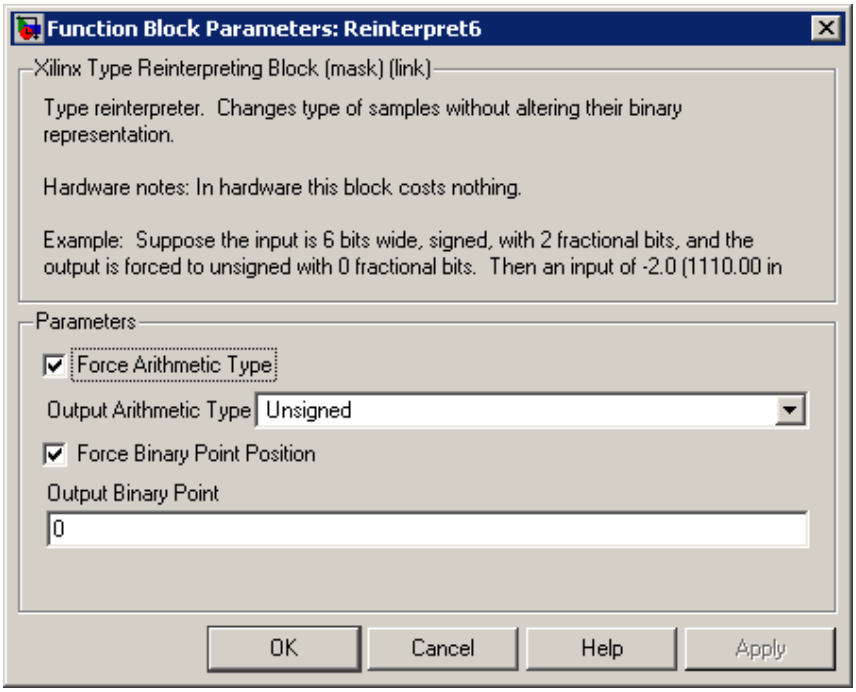
Once *OK* is clicked the block will have an *en* input which is the enable port. The Register will only output when the enable port is set to 1. Connect the *en* port to a branched wire from the *sync_out* port on the **Adder_tree** block. Connect a branched wire from the **Delay** block to the *d* port on the **Register** block.



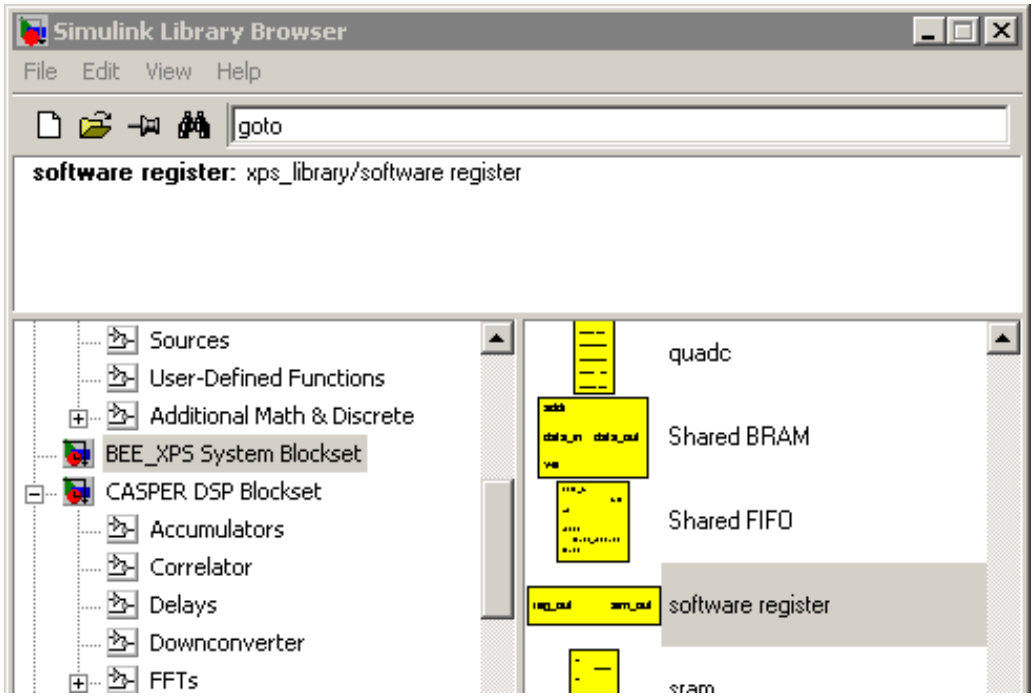
If you will recall we have a 32.14 binary value being placed into our **Register** block. But when we place this value in the software register the output will be a 32.0 number since the software register won't know where there is a binary point. To be on the safe side we should fix the output of the **Register** block to 32.0 before it is put in the software register. There is a **Reinterpret** block which will do this.

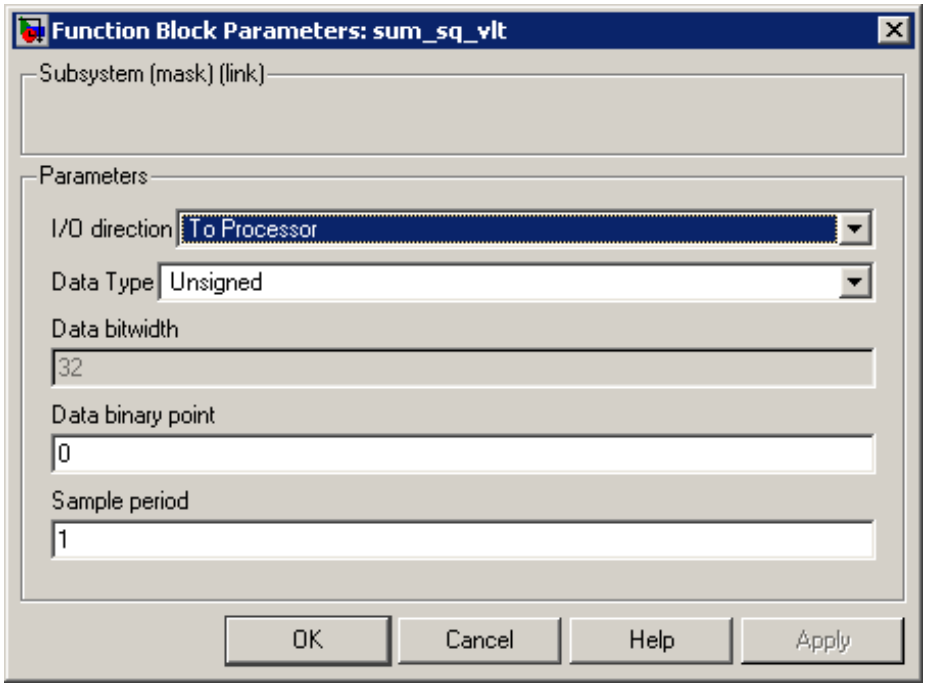
Drop a **Reinterpret** block from **Basic Elements** in the **Xilinx** blockset onto the design, the defaults of the block will set the binary point to 0. Connect the output of the **Register** block to the input of the **Reinterpret** block.





The last block we need is the software register which represents a register which can be read over a network interface. Drop a **Software Register** block from the **BEE_XPS System** blockset onto the design. The default settings for this block are what we want. Connect the output of the **Reinterpret** block to the *reg_out* input of the **Software Register** block. The name of this block is important for when we later interact with the hardware over a network, name the block *sum_sq_vlt*.



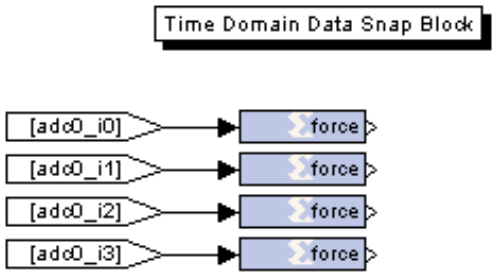


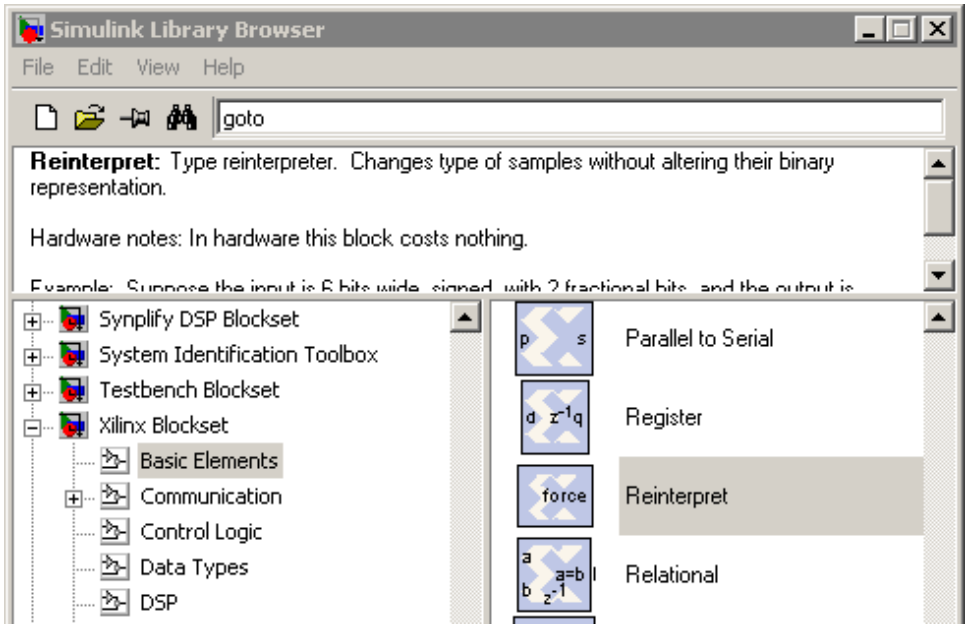
This subsystem is now complete, the last subsystem to design is a time domain data recorder using snap block.

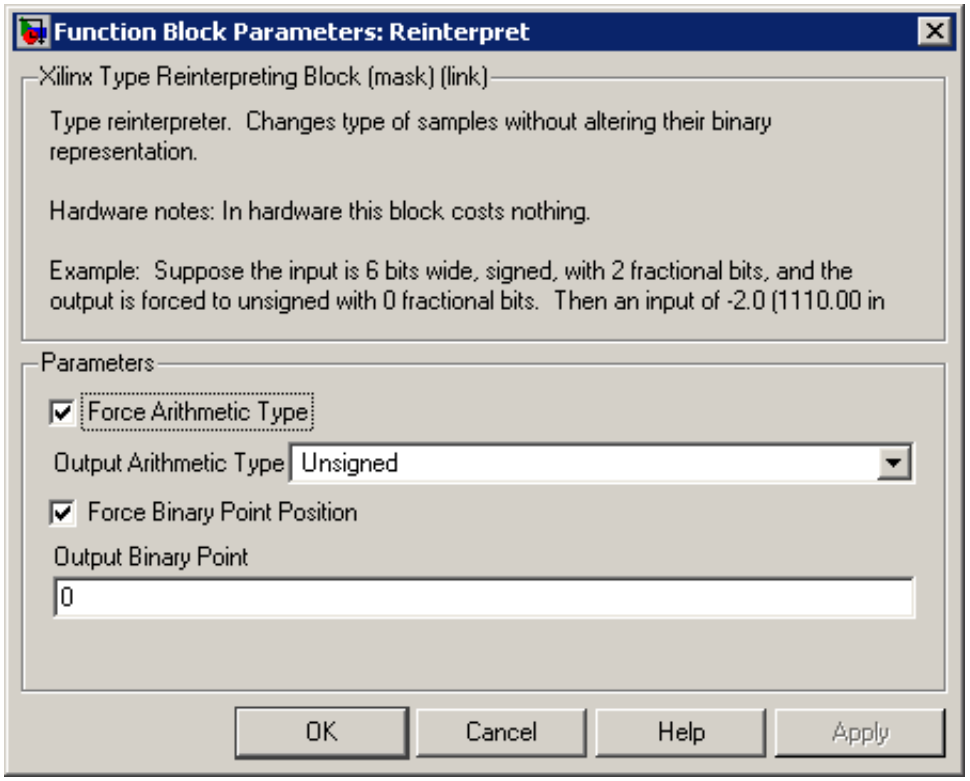
Time Domain Snap Block

Along with generating a summed power value every 2048 clock cycles it would be useful to write those 8192 ADC samples which were used in the computation of the summed power to a piece of memory which can be accessed from another machine. We can do this be writing the values to the BRAM memory.

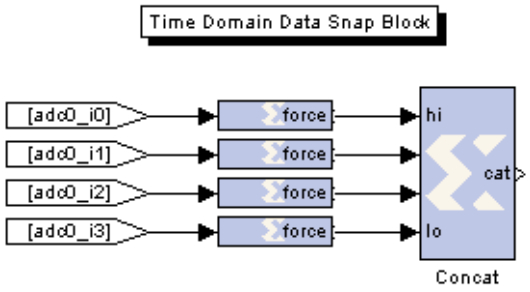
First create another set of ADC sample **From** blocks which are from **Signal Routing** in the **Simulink** blockset .These are 8.7 values and since we will not be manipulating the values, but just recording them to BRAM they should be changed to 8.0 binary values using the **Reinterpret** block from **Basic Element** in the **Xilinx** blockset. Place 4 of these blocks into the design and connect each to the ADC **From** blocks.





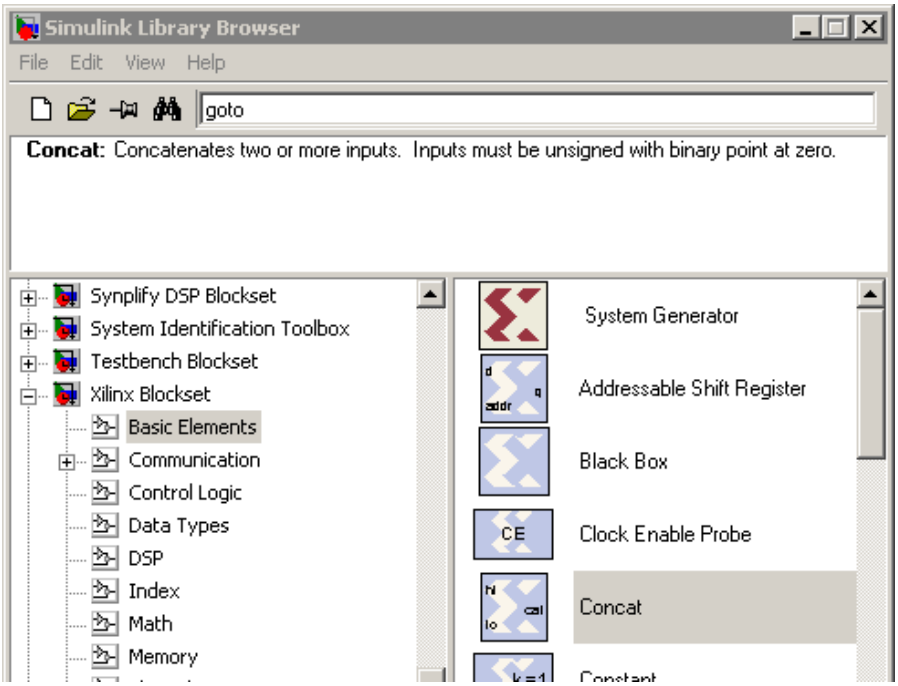


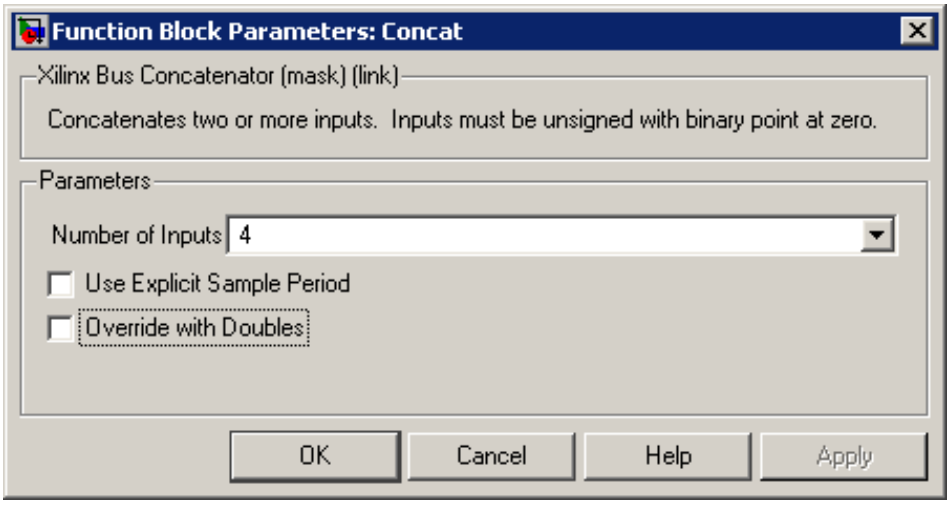
Like the software register we use in the summed power subsystem the BRAM uses 32 bit memory pieces. We have 8 bit data, a simple option is to reinterpret that 8 bit data as 32 bits, but that would be a lot of wasted memory. A better method would be to concatenate 4 8 bit values into a 32 bit value. Then we would only need a BRAM of size 32 by 2048 to store the 8192 values. There is easily enough a **Concatenate** block to do this.



Place a **Concatenate** block from **Basic Elements** in the **Simulink** blockset onto the design and *double-click* on it.

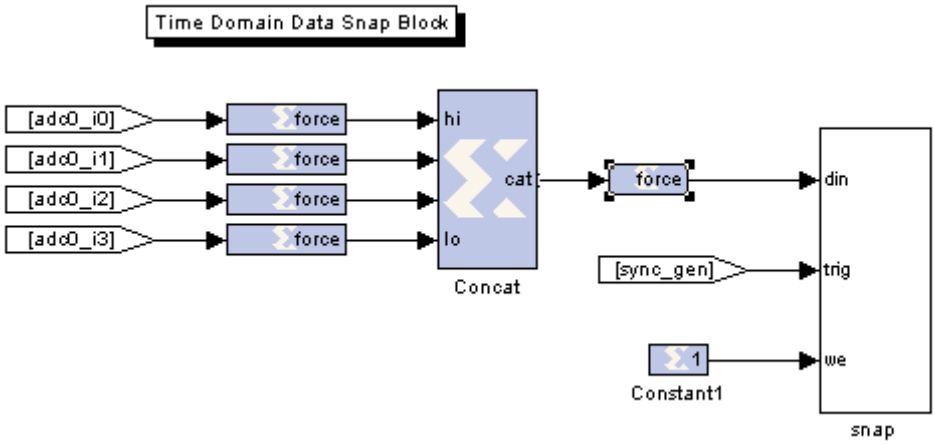
- Set **Number of Inputs** to 4





Connect the outputs of the 4 **Reinterpret** blocks to the inputs of the **Concatenate** block.

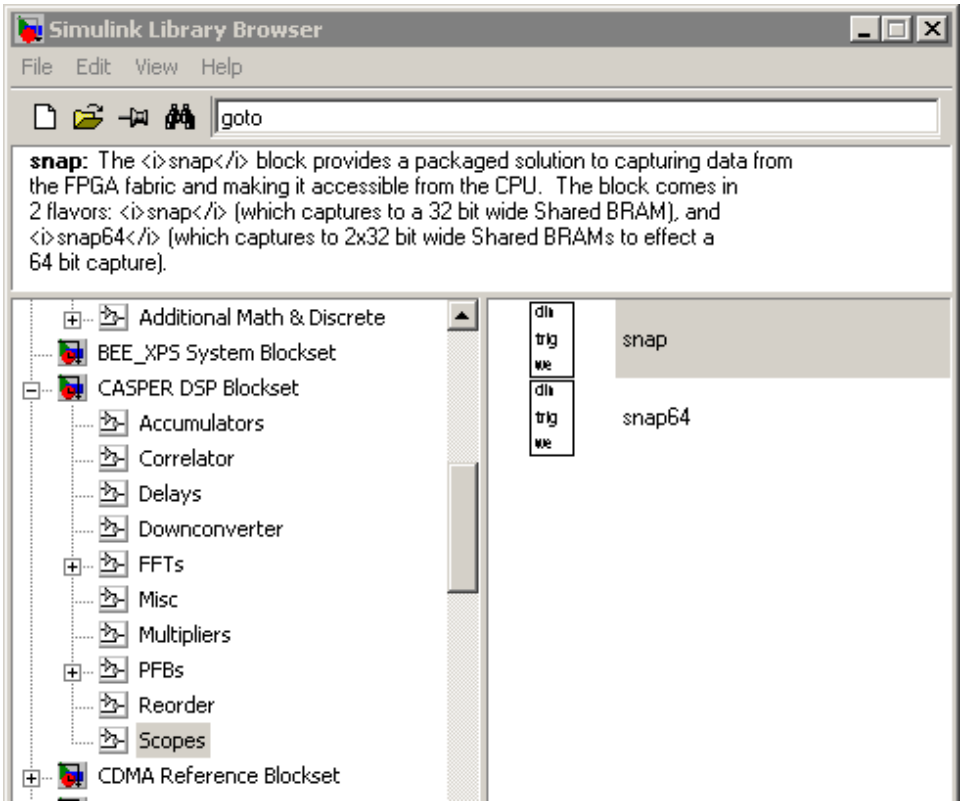
Now that 4 values are packed in to 32 bit values they can be written to BRAM. The easiest way to perform this task is to use the **Snap** block which takes in a 32 bit value and writes it to BRAM.

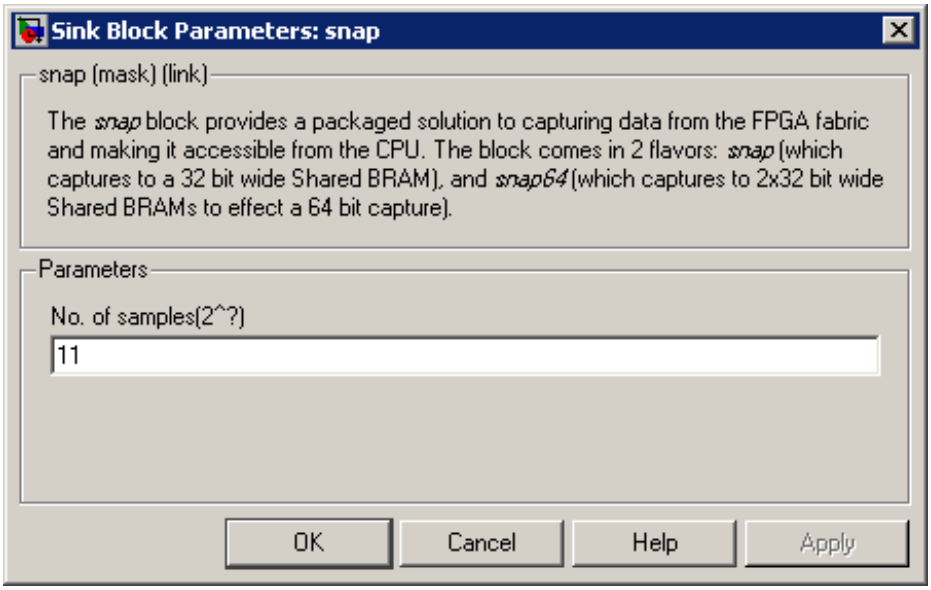


There are 3 inputs to the **Snap** block: *din*, *trig*, and *we*. The *din* input is data input, *trig* triggers the writing of data to the BRAM, and *we* is the write enable option. For this design we want to write 2048 32-bit values starting every time the sync generator sends a sync pulse. Write enable should be set to 1 such that it is always on.

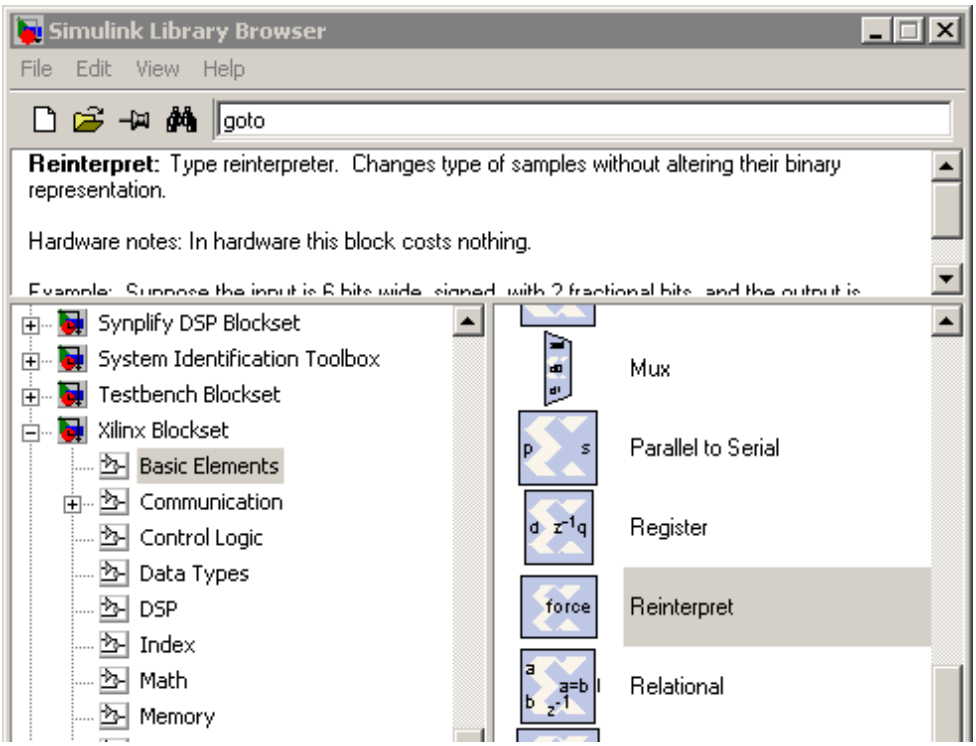
Drop a **Snap** block from **Scopes** in the **CASPER DSP** blockset, and *double-click* on the block.

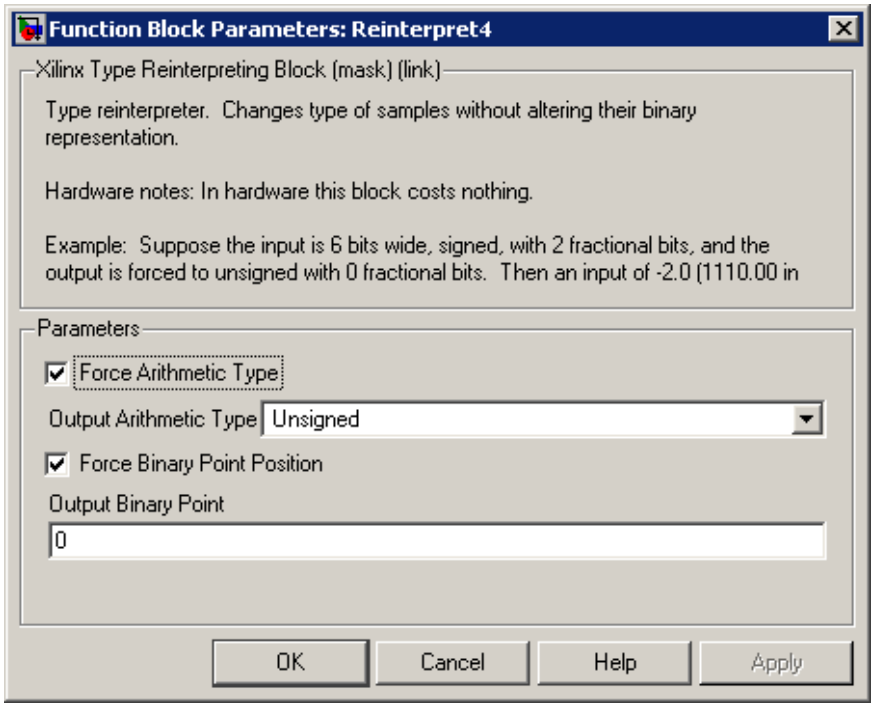
- Set **No. of Samples (2^?)** to 11





After the **Concatenate** block there in order to make sure we are writing 32.0 data to the BRAM we should add another **Reinterpret** block. Connect input of a default **Reinterpret** block from **Basic Elements** in the **Xilinx** blockset to the output of the **Concatenate** block. Connect the output of the **Reinterpret** block to the *din* input of the **Snap** block.

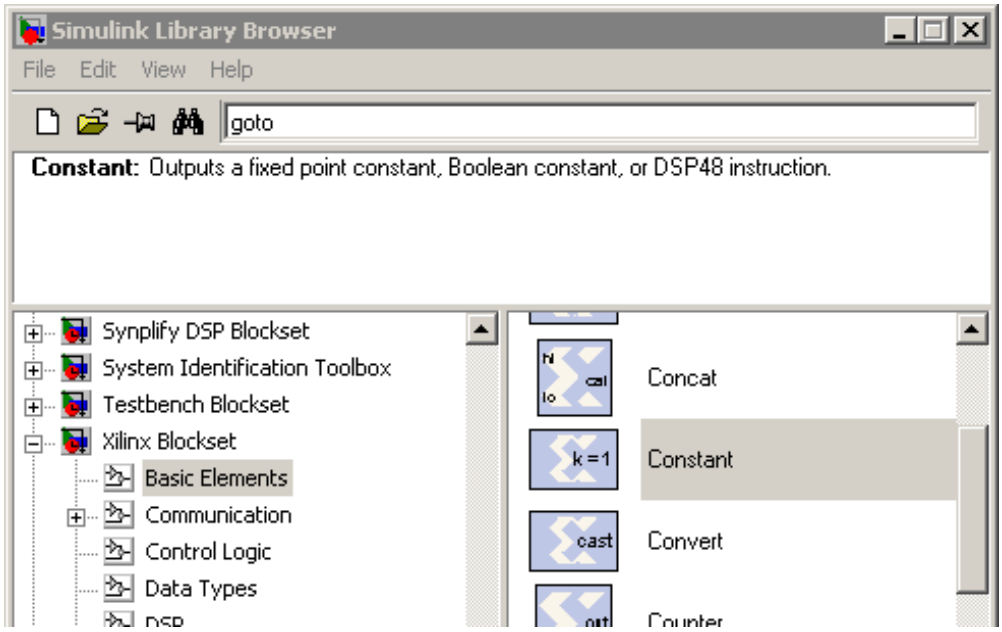


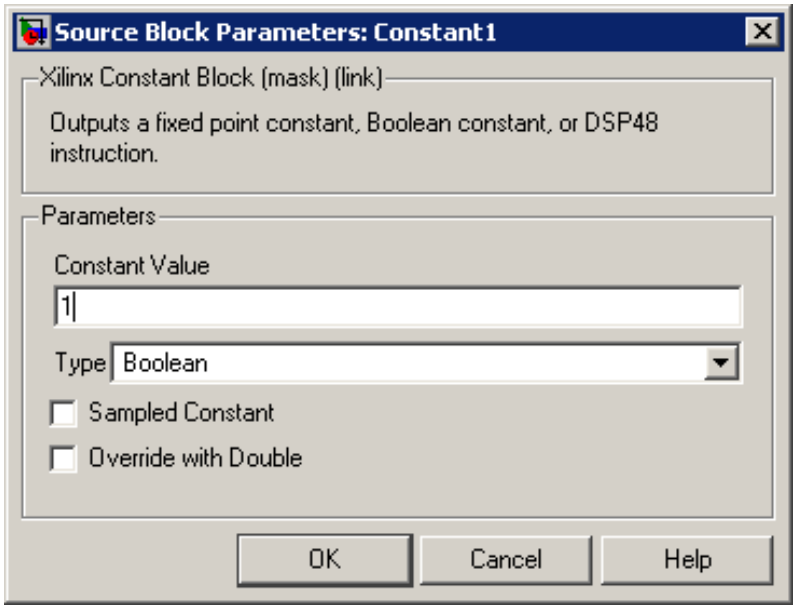


For the *trig* input on the **Snap** block connect a **From** block to with the *sync_gen* tag such that the *trig* input is connected to the sync generator.

Place a **Constant** from **Basic Elements** in the **Xilinx** blockset onto the design and *double-click* on the block:

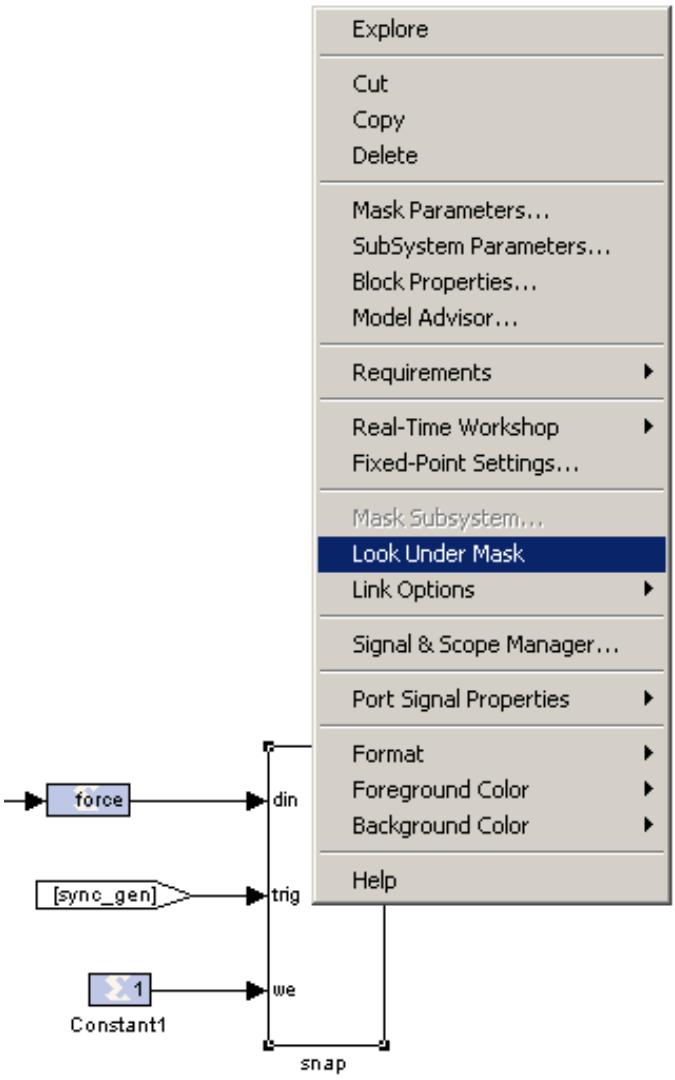
- Set **Constant Value** to *1*
- Select **Type** to be *Boolean*



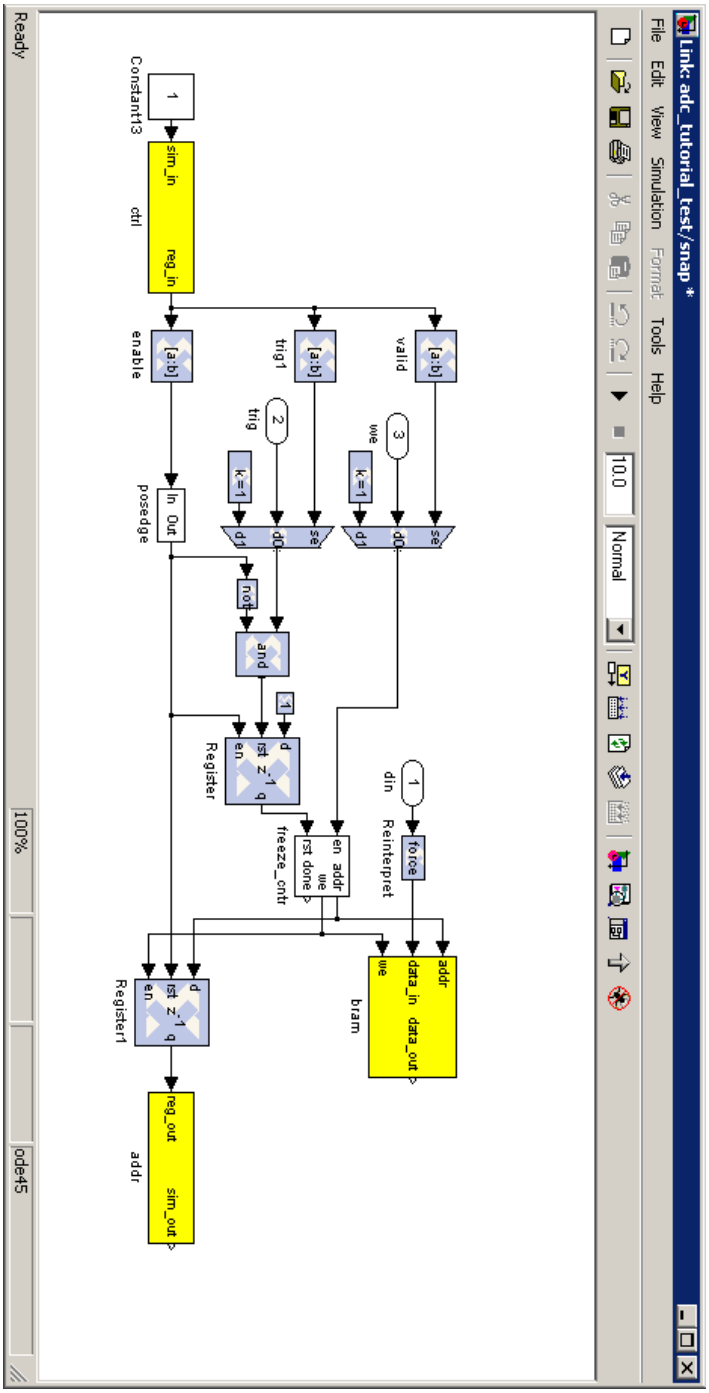


Connect the output of the **Constant** block to the *we* input of the **Snap** block.

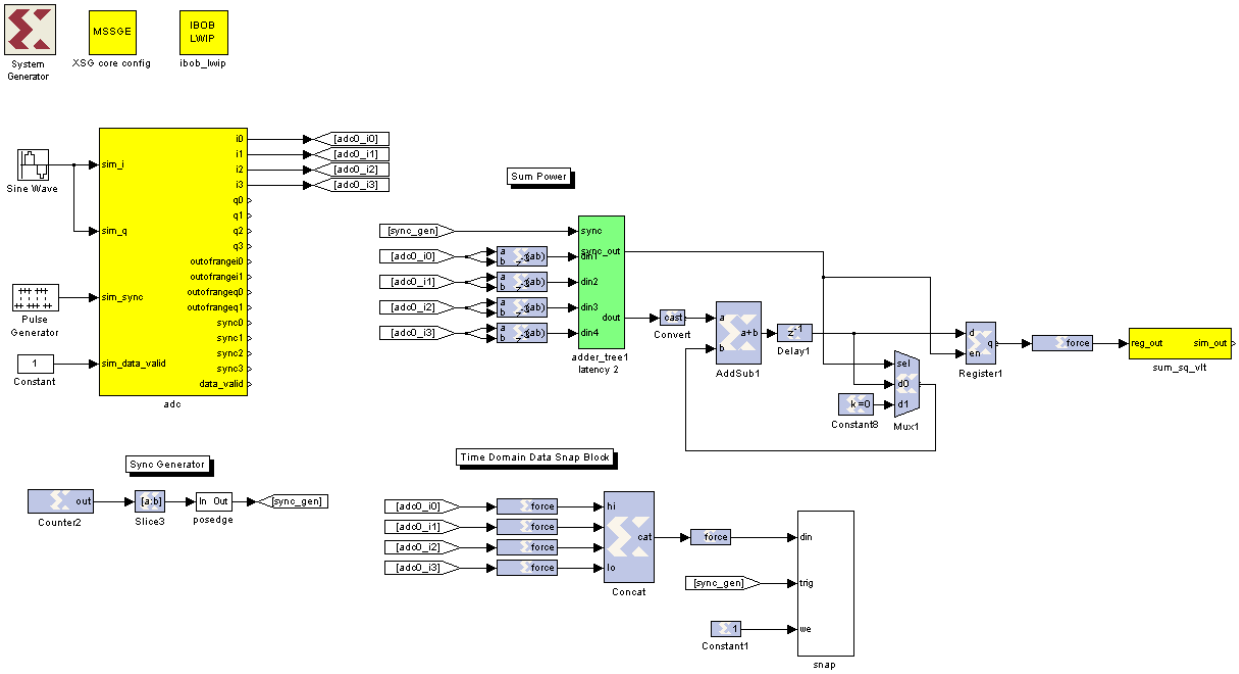
Note on the Snap Block: The **Snap** block is a CASPER block which is made up of simpler blocks which we can take a look at. *Right-click* on the block and select *"Look Under Mask"*.



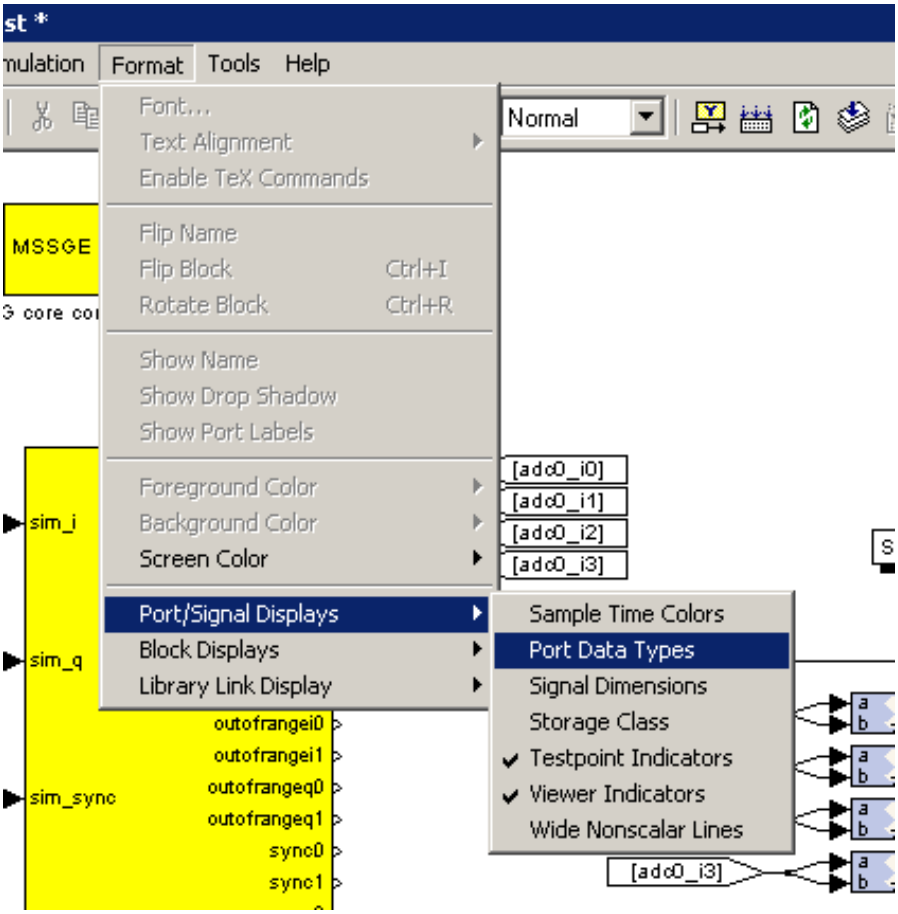
A new window will appear and show the design of the block.

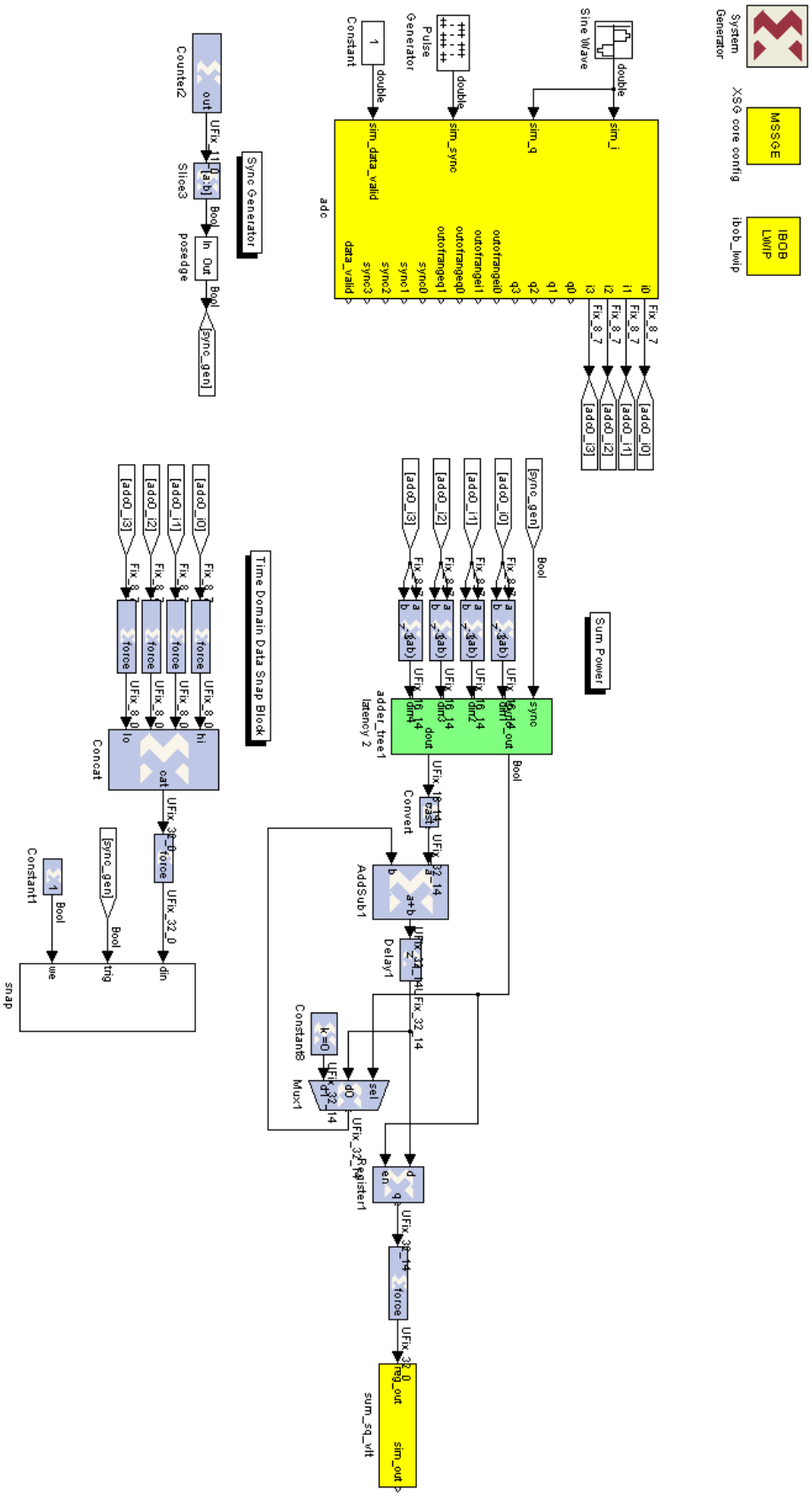


Finally, if all has gone well then we have a completed, but untested design.



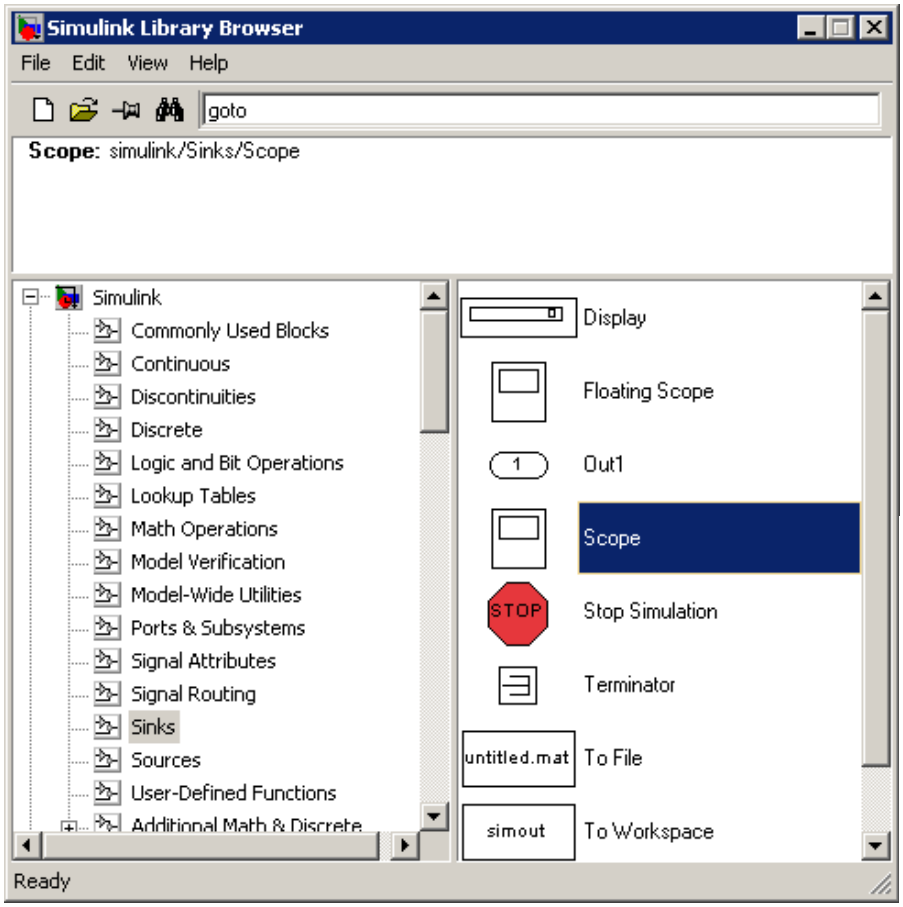
A useful option in seeing the number of bits going from different blocks is to turn on the *Data Port Types* options which can be done by going to the **Format** menu – **Port/Signal Displays** – select *Data Port Types*.

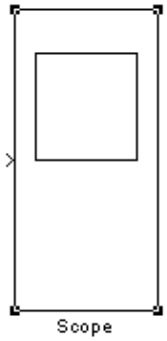



System Generator
XSG core config
MSSGE
IBOB LWIP
ibob_lwip
Sine Wave
double
sim_i
sim_q
Pulse Generator
sim_sync
Constant
sim_data_valid
adc
i0
i1
i2
i3
q0
q1
q2
q3
outofrange0
outofrange1
outofrange2
outofrange3
sync0
sync1
sync2
sync3
data_valid
Fix_8_7
add0_i0
add0_i1
add0_i2
add0_i3
Sum Power
sync_gen
Bool
adder_tree1
latency 2
sync
dout
Convert
AddSub1
Delay1
Constant8
Mux1
Register1
sum_sq_vlt
sim_out
Time Domain Data Snap Block
Concat
Constant1
snap
Counter2
Slice3
Sync Generator
posedge
sync_gen

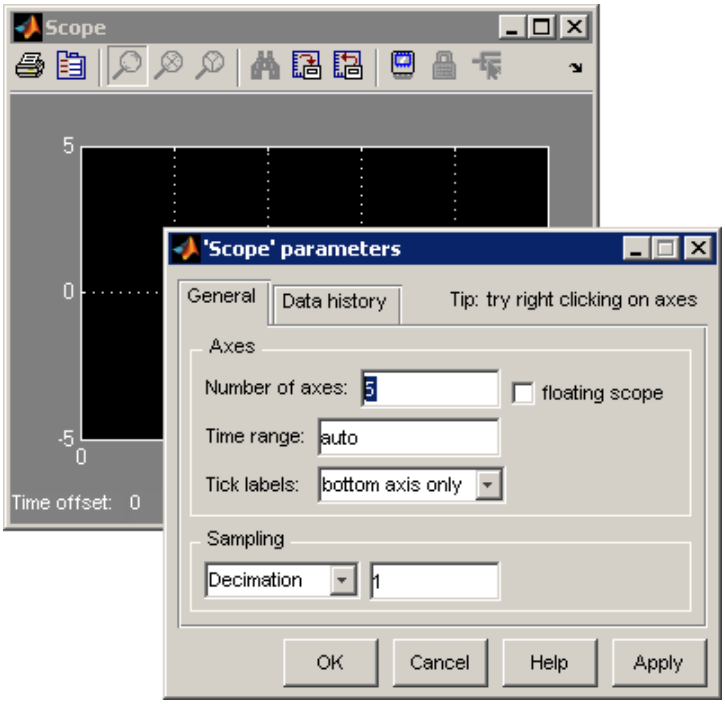
Test Simulations

Before compiling a design and placing the bitstream on a board it is useful to simulate the design and look at how the output matches with what is expected. Simulations also find logical errors and make sure blocks will work properly with their inputs. Simulink allows a design to simulated for a number of clock cycles. It should be noted that the larger and more complex the design the longer Simulink takes to simulate it. When we simulate a design there are various points along the design that we would like to see. For this we use the **Scope** block which is in **Sinks** in the **Simulink** blockset.

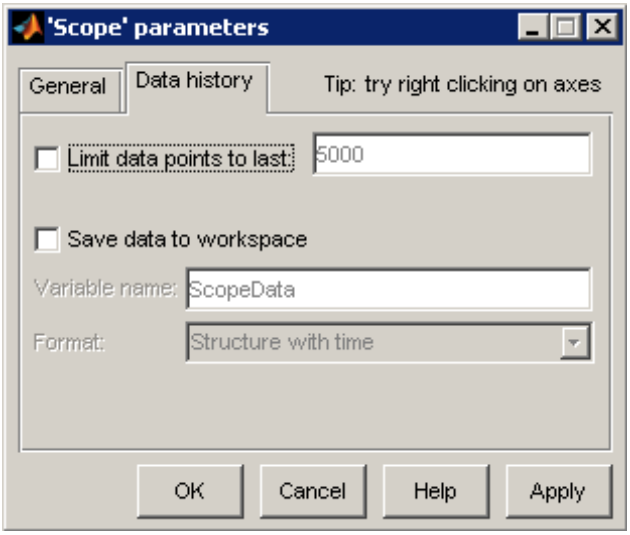




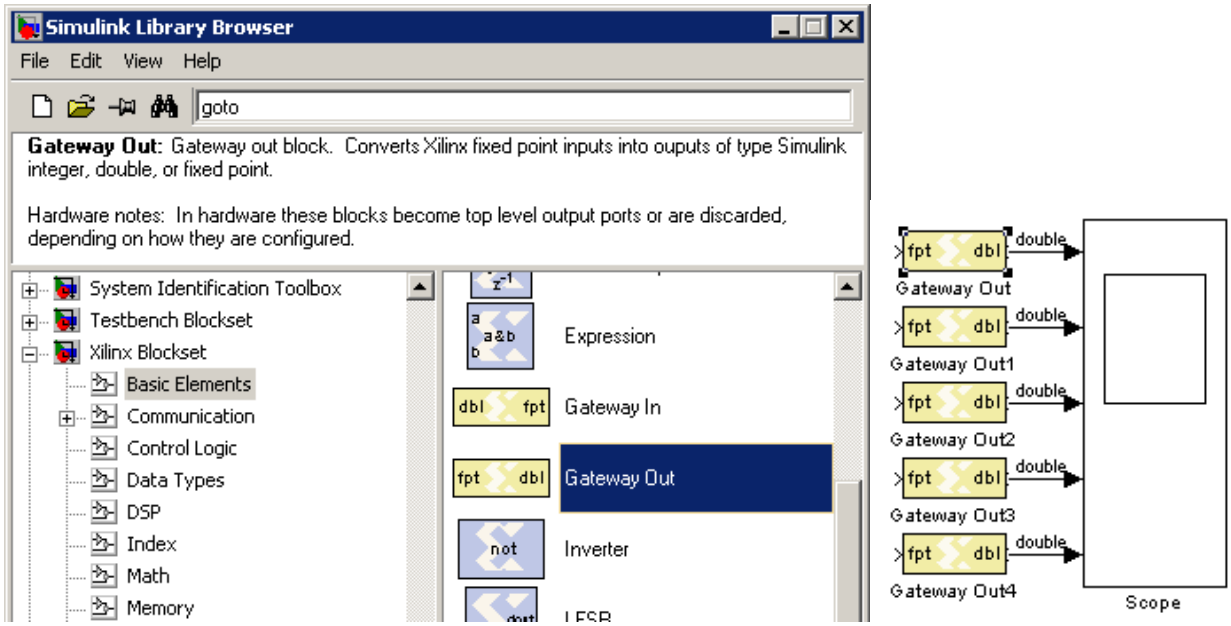
When a **Scope** block is first placed into a design there is only one input, this can be changed by *double-clicking* on the block. The scope display will appear and *clicking* on the *Parameters* button (second on the upper left) will bring up a new window. Then under the heading *General* the setting *Number of Axes* sets the number of inputs. For this simulation we will use 5 inputs.



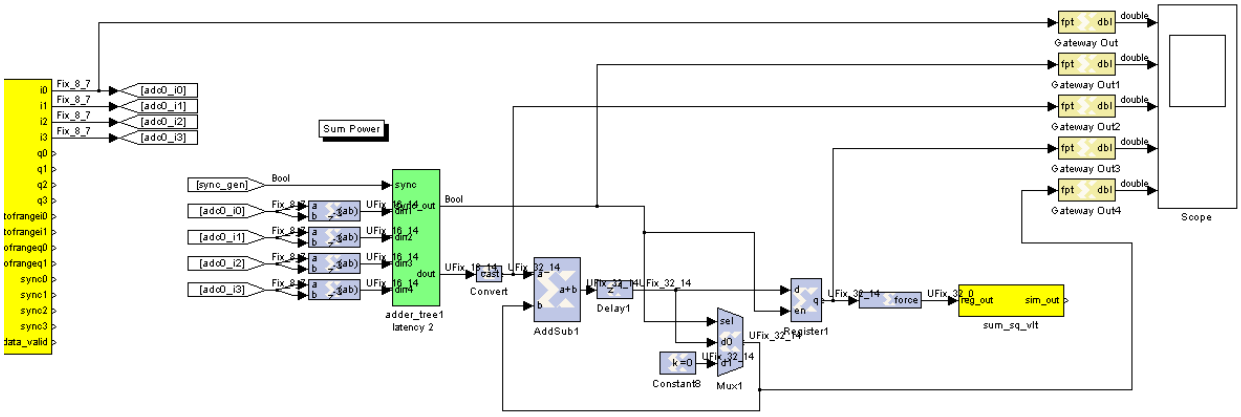
The default scope history to remember is only 5000 cycles, we will be looking at more then 5000 cycles. In the *Data History* tab uncheck the *Limit Data Points to Last* option.



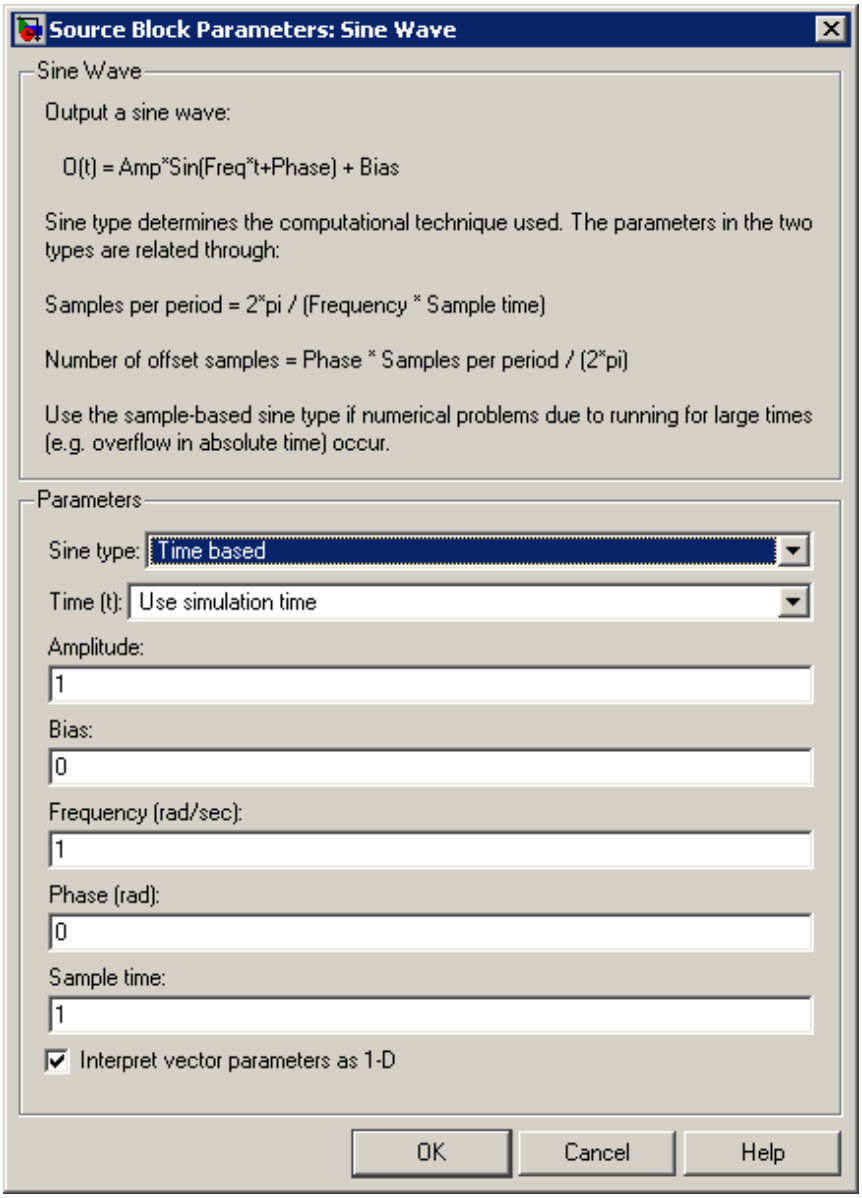
Now that our **Scope** block has 5 inputs we need one more block to get meaningful data out onto the scope display. The values going from block to block have different bit lengths and different meanings. These values should be converted to meaningful integer or float values for the scope. There is a **Gateway Out** block which will do this. Places a **Gateway Out** block from **Basic Elements** in the **Xilinx** blockset onto the design and duplicate the block for each **Scope** block input and connect the output of the **Gateway Out** block to the input of the **Scope** block.



The inputs on the **Scope** block can be hooked up to any wire in the design, feel free to pick some wires to look at. You want to pick wires which will have meaningful outputs and help to determine that the design is simulating properly. Below is an image of the wiring I used to check the design.

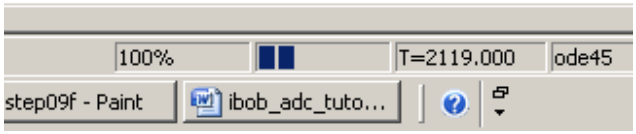


The input to the **ADC** block will determine the output on the scope, at the beginning of the design we placed a **Sine Wave** block onto the input of the **ADC** block. *Doulble-click* on the **Sine Wav** block and make sure the settings are the same as the settings below, then click *OK*.

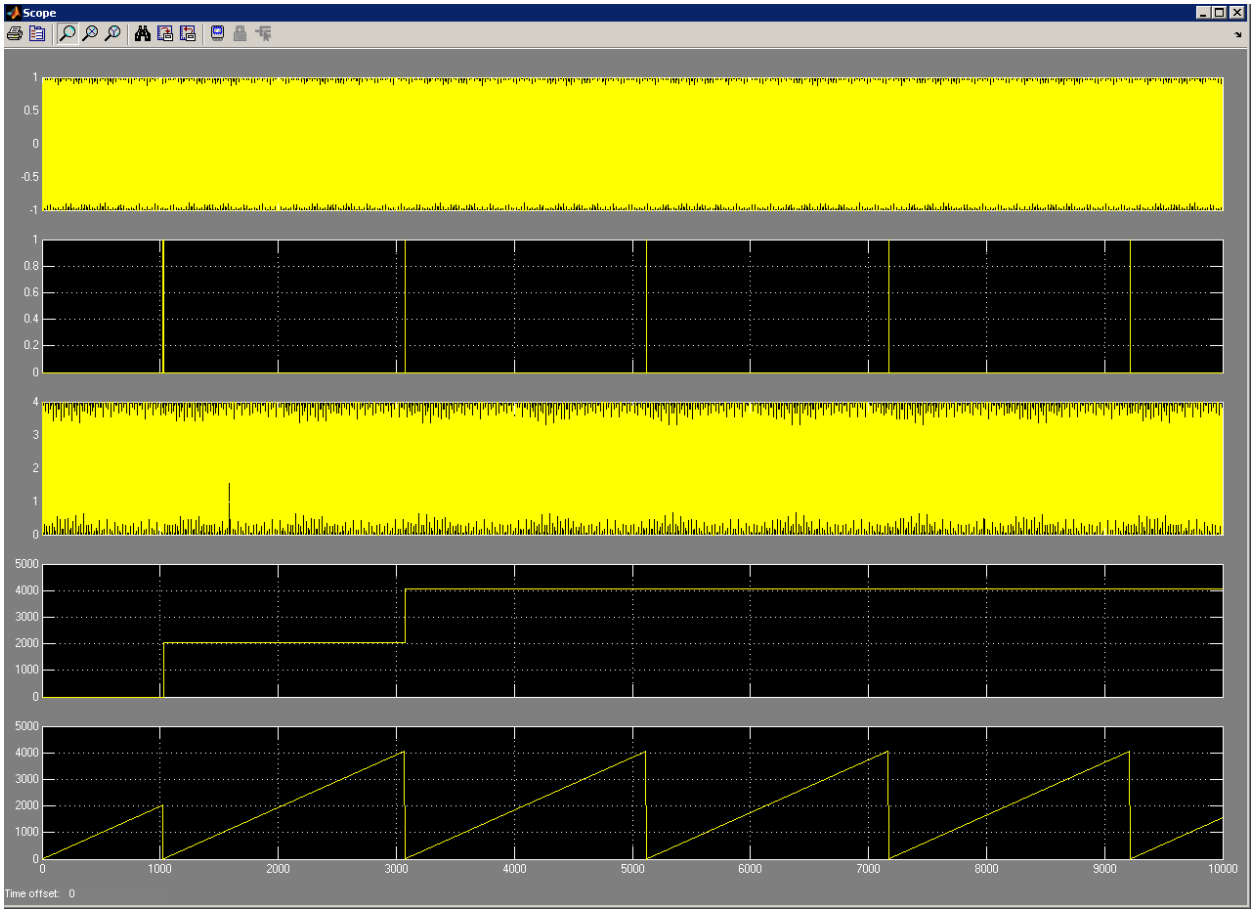


The **Scope** block will display the output of one of the ADC samples, the sync generator, the value of the 4 summed samples before the accumulator, the value written to the software register, and the value in the accumulator. Now we want to simulate the design for many clock cycles. Since the design uses a 2048 accumulator we want at least that many clocks simulated. The first few clock cycles could have undefined output while the system settles. If we do 10000 clock cycles that should give the system time to settle and show a few iterations of the accumulator. At the top of the design window there is a box with a **Play** button to the left of it. This box sets the number of cycles, put in 10000 and press the **Play** button.

Once the simulation has begun it will take a little time to finish. The progress of the simulation is displayed at the bottom of the window.

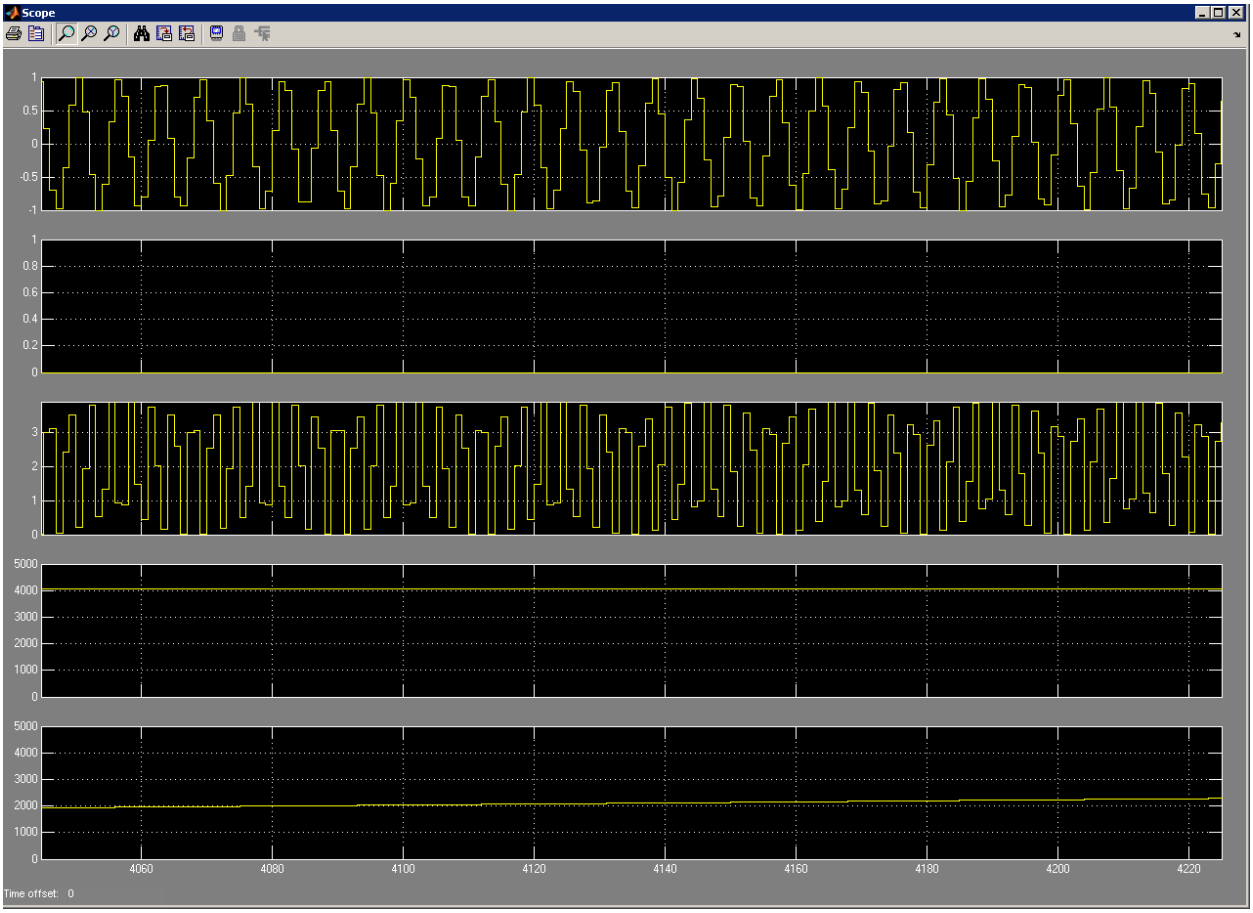


Once the simulation is complete then we can look at the output by *double-clicking* on the **Scope** block. Enlarge the new window that appears and be sure to *click* on the **Auto Scale** button(pair of Binoculars at the top of the window). The output should look something like the image below.



The top window shows an ADC sample which should be a sine wave, we can zoom in on a portion to see that it is a sampled sine wave ranging -1 to 1. The next window is the sync generator which is triggering every 2048 clock cycles. The next window shows the four summed squared samples which should have a range 0 to 4. The fourth window shows the value at the software register. That value should be about the same since the input sine wave isn't changing. The beginning of the window is stepped because the first accumulation isn't a full 2048 accumulation, this is why we give the system time to settle. The last window is the value of the accumulator. For 2048 clocks the value always increases then is reset to 0. The simulation looks good and the results are what we expected.

To zoom into a section of the simulation *click-and-drag* a box over the section to view.



Now that our simulations look good we are ready to compile the design and test it out on an iBOB.

From here there are a few steps which we can work on the further:

FFT block/simulation: add an FFT portion to the design, and simulate the design

BEE_XPS: run BEE_XPS to generate a bit file

Loading design onto an FPGA: physically load a design onto an iBOB

Telnet to iBOB: login to the iBOB and look at the software registers and brams

Revision History

Date	Revision	Changes
March 23, 2009	v1.0	Initial revision.