XILINX®

XAPP264 (v1.0) November 26, 2002

Building OPB Slave Peripherals using System Generator for DSP

Author: Jonathan Ballagh, Eric Keller, James Hwang, Phil James-Roxby

Summary

The inclusion of embedded processor cores in Xilinx FPGAs opens new doors for high-throughput digital signal processing applications. System Generator for DSP is a high-level modeling environment for designing custom DSP data paths with performance and efficiency comparable to hand-crafted designs. Because System Generator for DSP is tightly integrated with the Simulink® and MATLAB® tools from The Mathworks, Inc., FPGA designs are implemented by users in a familiar setting without being overly concerned with underlying hardware details.

A model can be extended to create a CoreConnect® On-Chip Peripheral Bus (OPB) compatible peripheral using the libraries provided in System Generator for DSP. These peripherals are used in conjunction with the MicroBlaze™ and PowerPC™ processor cores, bringing unprecedented throughput and control to DSP embedded systems designers.

This application note shows how to model a slave OPB peripheral in the System Generator for DSP and how to include the peripheral in an embedded systems platform compatible with the Xilinx Embedded Development Kit (EDK). As an example, simple System Generator for DSP constructs are used to connect a reloadable DA FIR filter to the OPB. An embedded (PowerPC or MicroBlaze) processor is used to control filter coefficient reloading. Primary attention is paid to connecting the DSP data path and the OPB. To illustrate how a processor might be used to exchange data with the DSP peripheral, the steps needed to incorporate the peripheral in a platform consisting of a processor and UART are described. Similar interface logic built using System Generator makes it straightforward to implement far more sophisticated signal processing peripherals.

Introduction

High-performance DSP data paths modelled in System Generator for DSP (System Generator) can be used as CoreConnect peripherals by extending them with an appropriate interface. The Xilinx BlockSet provides the components necessary to model a DSP peripheral and OPB interface. Although at present (v2.3 release), there are no intrinsic software models for either the PowerPC or MicroBlaze processors, sufficient subsets of PowerPC and MicroBlaze processor functionality, i.e., basic bus transactions, can be modeled within the same environment. This results in a robust simulation and debug environment suitable for DSP embedded systems design. When the software translates the model into hardware, the same vectors used in the Simulink simulation are used as golden test vectors in the hardware test-bench simulation. By ensuring correct peripheral behavior in the Simulink tool, the designer can be confident the peripheral will function correctly in hardware.

MATLAB and Simulink are registered trademarks of The MathWorks, CoreConnect is a registered trademark of IBM.

This application note discusses the techniques needed to extend a System Generator signal processing data path into a slave peripheral for use on the OPB. These techniques are illustrated using a example platform comprised of a PowerPC or MicroBlaze processor, a UART Lite peripheral for communication with a host PC and a reloadable distributed arithmetic (DA) FIR filter DSP peripheral modeled in System Generator. The principles described in this application note, provide a sufficient understanding of the System Generator peripheral modeling process to promote similar techniques for use with other user models. In fact, a significant portion of the bus interface logic used in the example peripheral model is applicable and reusable with other models. This application note assumes the reader is comfortable with System Generator for DSP as well as the Simulink and MATLAB tools. It also assumes the reader has a basic understanding of OPB bus transaction protocols.[1]

Example Platform

The example platform shown in Figure 1 explains how a System Generator model can be extended to become a peripheral. It includes a software peripheral, an embedded processor (either PowerPC or MicroBlaze) for controlling the peripheral, and a UART Lite for bidirectional communication through a serial cable with an external host PC. The primary focus is on the implementation of the peripheral itself.

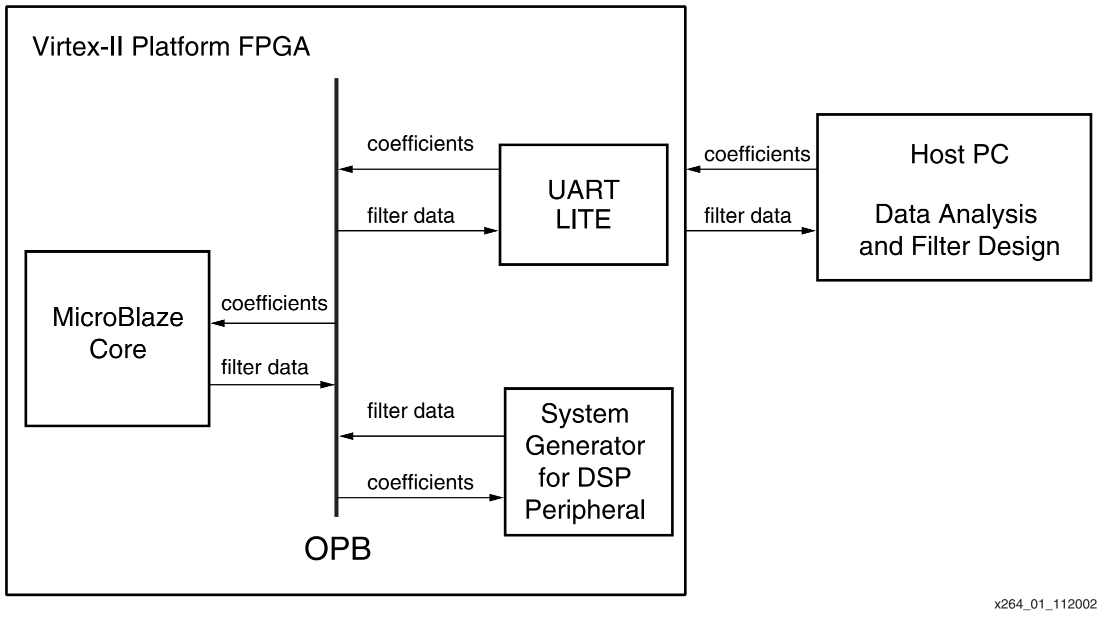


Figure 1: **FPGA Platform: MicroBlaze, UART, and System Generator DSP Peripheral**

The peripheral consists of a System Generator reloadable DA FIR filter augmented with a small amount of control logic. The processor and UART use a serial cable to direct data between the peripheral and a host PC. The PC uses MATLAB to analyze the filter output and design new filters. The PC also initiates filter reloading and transfers new filter coefficients to the processor. Upon receiving new coefficients from the PC, the processor controls the filter reloading from within the FPGA.

The platform operates under two modes: filter reloading and filter frame data transfer. When the filter is not being reloaded, frames of filter output are transferred over the OPB to the processor. From there they are sent to the PC for analysis. On the PC the user can use a MATLAB filter design tool to construct a new filter. After a new filter is constructed, the coefficients are automatically transferred across the serial cable to the UART and then to the embedded processor. Upon receiving the coefficients, the processor transfers the coefficients to the peripheral.

DSP Data Path

System Generator is ideal for modeling high-performance custom signal processing data paths. In particular, the ease of modeling filtering applications in the software, makes them useful, instructive examples. To extend a data path into an OPB peripheral, an example DSP data path is used. It incorporates a reloadable FIR filter block from the Xilinx BlockSet. Included with the filter is a small amount of control logic to manage coefficient reloading, adjust data rates, and control filter output frame buffering.

A reloadable DA FIR filter block lies at the heart of the data path. The block supports parameterization of coefficient precision, coefficient binary point, number of taps, and filter oversampling rate. The block used in the example datapath is configured with 32 taps, 12-bit coefficient precision, and reloadable coefficients (Figure 2).

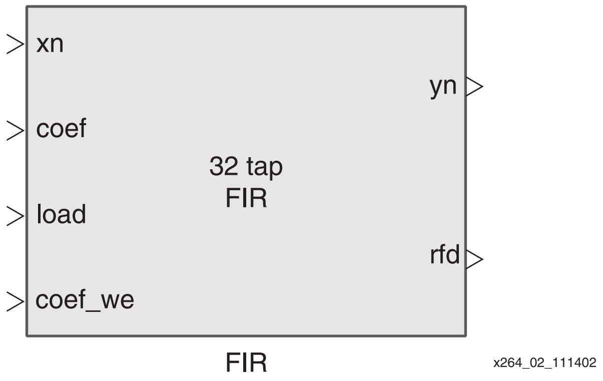


Figure 2: **DA FIR Filter Block from the Xilinx DSP BlockSet**

Operation of the filter block is straightforward. When the filter is not being reloaded, input values drive the *xn* port and filter output values drive the *yn* port. Filter reloading is initiated with a pulse on the load port, *load*. During reload the *rfd* port outputs zeros to indicate the filter is busy. Following the load pulse, new coefficients are written to the *coef* port. Asserting *coef_we* identifies the current value on the *coef* port as valid. After all coefficients are written, the filter comes back online some number of cycles later and resumes processing data. The block signals when coefficient reloading is complete by reasserting the signal driven by *rfd*. For a detailed description of the block, please refer to the **DA FIR filter data sheet**[2].

The filter block is augmented with control logic to allow the data path to communicate with the memory-mapped interface of the peripheral. The data path is implemented in the subsystem shown in Figure 3.

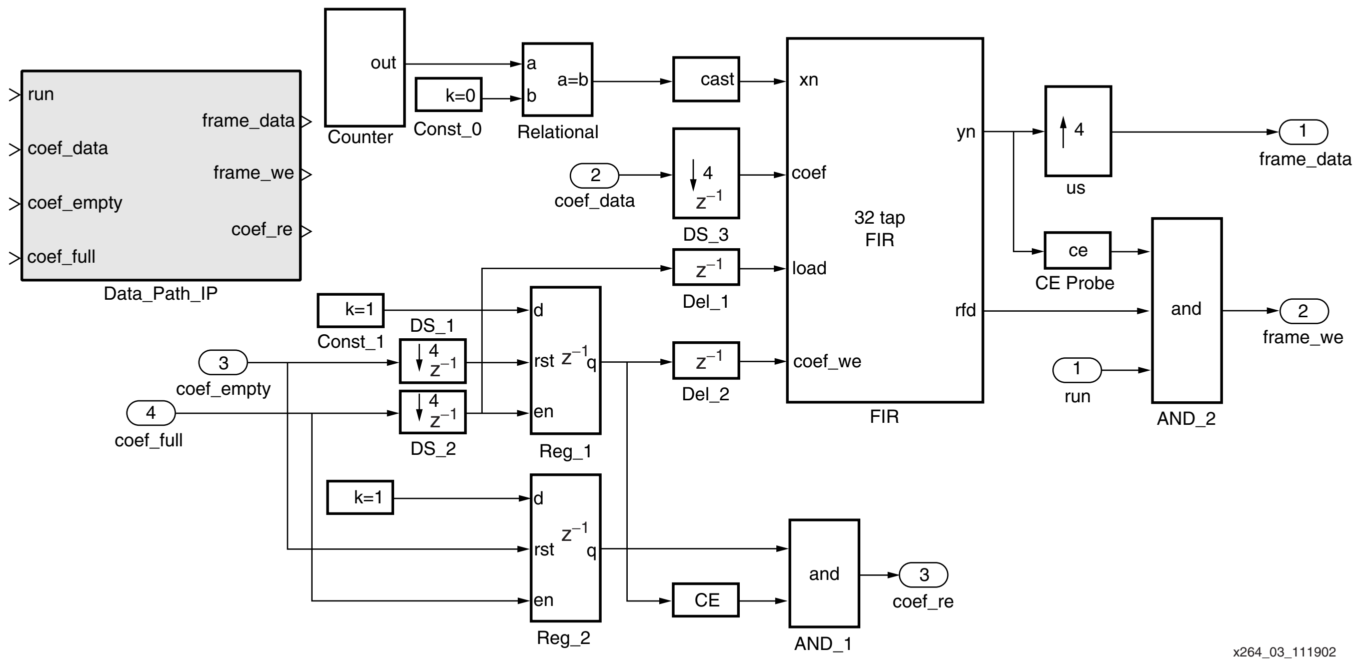


Figure 3: **Example DSP Data Path Subsystem**

The control logic enables the following in the data path:

- The data path monitors the status of a 1-bit *run* control register in the memory map interface of the peripheral. The value of this register is driven to the subsystem through the port labeled *run* in Figure 3. When the register is set to "1", contiguous filter output values are written to a FIFO residing in the memory map interface. The FIFO write-enable signal is driven by the *frame_we* port of the data path. When the register is set to "0", no values are written to the FIFO. This control register allows the processor to manage data flow from the peripheral to the bus. A full FIFO constitutes one frame of filter output data.
- The data input port of the filter is driven with an impulse train. The impulse train is generated using a counter/comparator pair (blocks "Counter" and "Relational" in Figure 3) to produce a pulse each time the counter rolls over. The maximum count value is chosen to be larger than the number of filter taps. The cast block converts the Boolean (1-bit) output of the relational block into a 12-bit input value driving the data input port of the filter.
- New filter coefficients written to the peripheral by the processor are stored in a second memory mapped FIFO. It is the responsibility of the data path to monitor the coefficient FIFO signals driven on input ports *coef_empty* and *coef_full*. When the FIFO is full, indicating all coefficients have been written, the data path initiates a filter reload sequence and issues read requests to the FIFO to obtain the new coefficients. The coefficient FIFO read request is driven on the *coef_re* output port.

To conserve hardware, the filter is configured to oversample at a rate of four. The oversampled filter runs at the system rate (i.e., the same rate as the OPB clock), and therefore the filter data rate is four-cycles per sample. To compensate for the rate change, up and down samplers are used (Figure 3 blocks "US", "DS_1," "DS_2," and "DS_3") at places where the data path connects to the bus. Clock enable probes extract the clock enable pulses used in multi-rate designs, and are used to ensure the FIFO Read/Write transactions align to the filter input sample frame.

Extending the Data Path into a Peripheral

A System Generator data path can be extended into a CoreConnect peripheral through a custom interface constructed using the Xilinx BlockSet. A typical peripheral model requires the following components in addition to the DSP data path.

- Interface to the OPB signals
- Address decoding logic
- A memory mapped register interface to the I/O ports of the data path
- Logic to manage bus transaction handshaking

To make the peripheral as modular as possible, each of the above components are encapsulated in their own Simulink subsystem. Using subsystems also allows each component to be designed and debugged individually, and then to be added to a library for future reuse. A general case System Generator peripheral consisting of subsystems is shown in Figure 4. The following sections focus on the implementation of each subsystem with an example using the reloadable DA FIR filter data path.

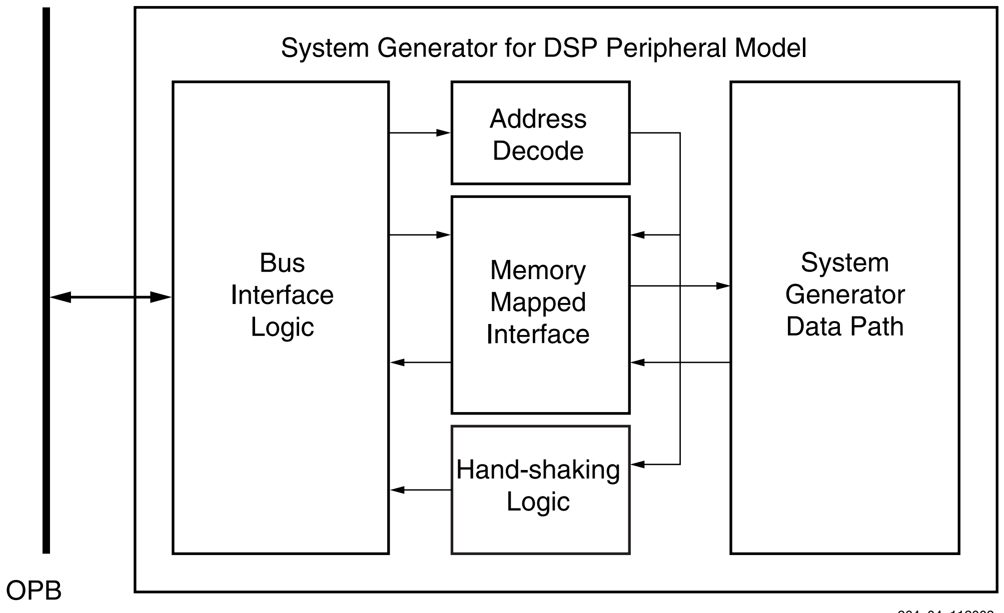


Figure 4: **A Peripheral Modeled in System Generator for DSP**

Bus Interfacing

Bus interface logic is needed to bridge the gap between the I/O ports on the peripheral and the OPB. A Simulink subsystem is a natural parking place for this logic. The benefits of encapsulating the bus interface logic in a subsystem are two-fold. First, placing this logic in a subsystem results in a convenient abstraction of the OPB. Users can easily tap-off signals as needed from the bus interface subsystem. Second, coupling Xilinx input gateway and output gateway with the subsystem logic ensures the necessary ports are instantiated on the top-level peripheral VHDL when the model is translated into hardware.

The names given to the gateway blocks reflect the corresponding OPB signal names. Following this guideline allows the designer to easily identify and associate OPB signals in the microprocessor peripheral description (MPD)[3] file with the corresponding top-level ports on the VHDL model description.

Separating the bus interface logic into two subsystems, one for signals driven to the peripheral by the OPB, and one for signals driven by the peripheral to the OPB, results in a more natural depiction of the left-to-right data flow within the model. This is done in the example by implementing the interface logic in two separate subsystems, *OPB2IP_IF* and *IP2OPB_IF*. *OPB2IP_IF* contains the interfacing of signals driven by the OPB to the peripheral. *IP2OPB_IF* connects signals driven by the peripheral to the OPB. Wherever possible, register these signals to improve timing.

The interface and subsystem logic for the *OPB2IP_IF* component is shown in Figure 5. A best practice design registers the signals read from the OPB. These registers can be removed if the peripheral timing constraints can be relaxed. Each register in the subsystem has an explicit reset port exposed. Routing the *OPB_rst* signal to the reset port of each register ensures the contents of these registers are reset to an initial value if the OPB reset is asserted.

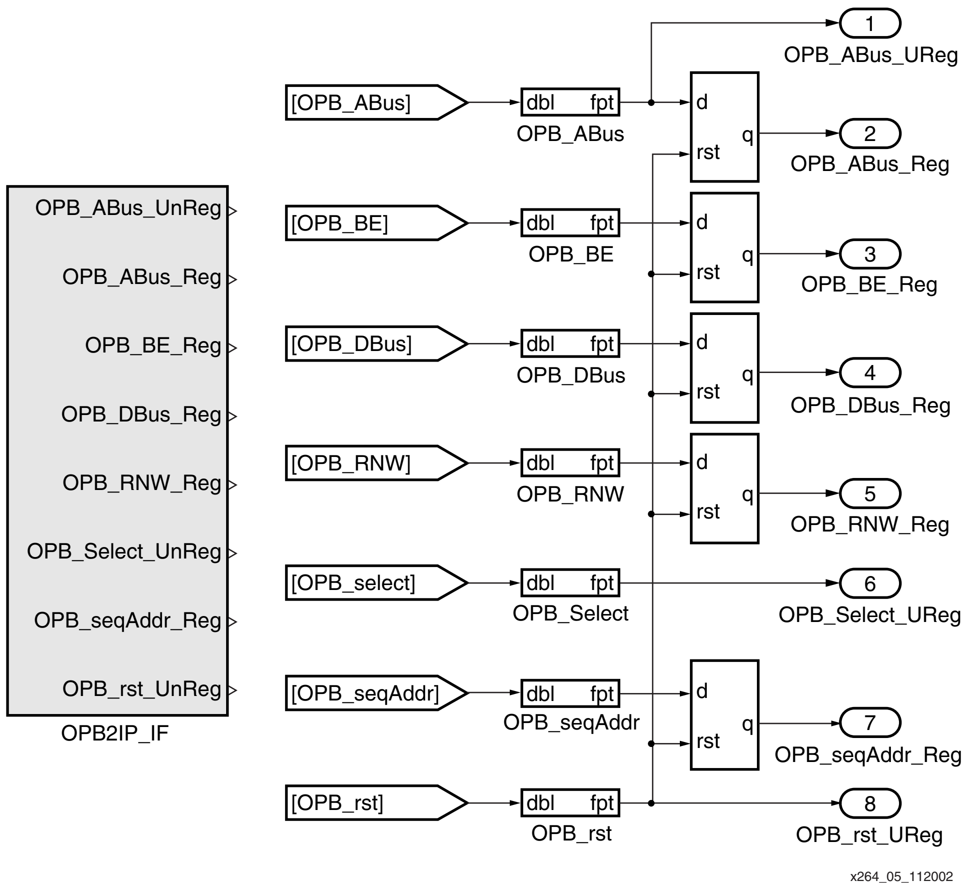


Figure 5: **OPB2IP_IF Subsystem**

Of note is the use of global from blocks as sources for the input gateways. Global from blocks allow gateway blocks to be driven without needing explicit ports on the subsystem interface. This has advantages for simulation, as shown when processor code is encapsulated into a separate processor model subsystem. During simulation, the processor subsystem drives these from blocks using global goto blocks.

The *IP2OPB_IF* subsystem is shown in Figure 6. All output signals are registered before being written to the OPB. Again, these register blocks can be removed if peripheral timing is relaxed. Global goto blocks follow the output gateways and allow the processor subsystem to monitor the output signals of the peripheral without explicit wiring. The reset port on the *SGP_DBus* register is driven by the registered acknowledge signal. This wiring ensures the peripheral data output register resets to zero on the cycle immediately following the assertion of the acknowledge signal. This satisfies the requirement to have the peripheral drive zeros to the OPB when the acknowledge is Low. The terminated OPB signals are not used in this example.

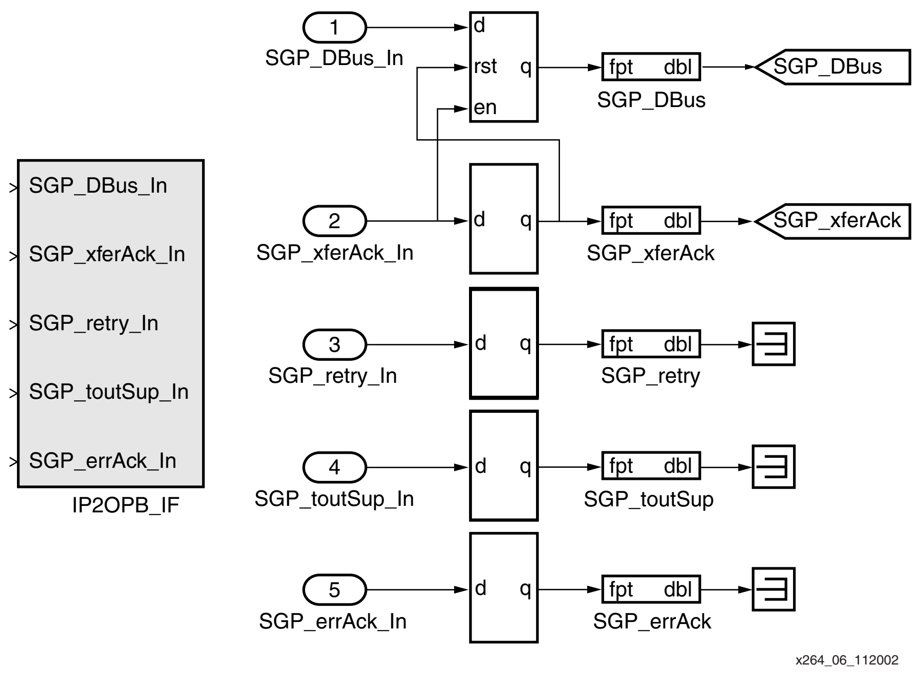


Figure 6: **IP2OPB_IF Subsystem**

The *IP2OPB_IF* and *OPB2IP_IF* subsystems are placed in the top level of the peripheral hierarchy. Every gateway is named after a corresponding OPB port and is assigned a matching width.

Address Decoding

When a processor (or other OPB master) attempts to read or write to a peripheral, it writes an address to the bus. It is the responsibility of the peripheral to decode the address and decide if the current address value is within the memory mapped allocation space of the peripheral. The OPB master indicates a valid address value by asserting the *OPB_select* signal. The peripheral needs only to decode the current address when *OPB_select* is High.

In this example, the address decoding subsystem is implemented with behavior matching the *p_select.vhd* [4] component distributed with the Xilinx EDK. For reuse, the subsystem is made as generic as possible. The subsystem and subsystem logic are shown in Figure 7. The *p_select* subsystem has two input ports, *addr* and *a_valid*. Port *addr* is driven by the bus address signal, *OPB_ABus*. The *a_valid* port of the subsystem is driven by the *OPB_select* signal from the bus. The *ps* output port drives the peripheral select signal for the model.

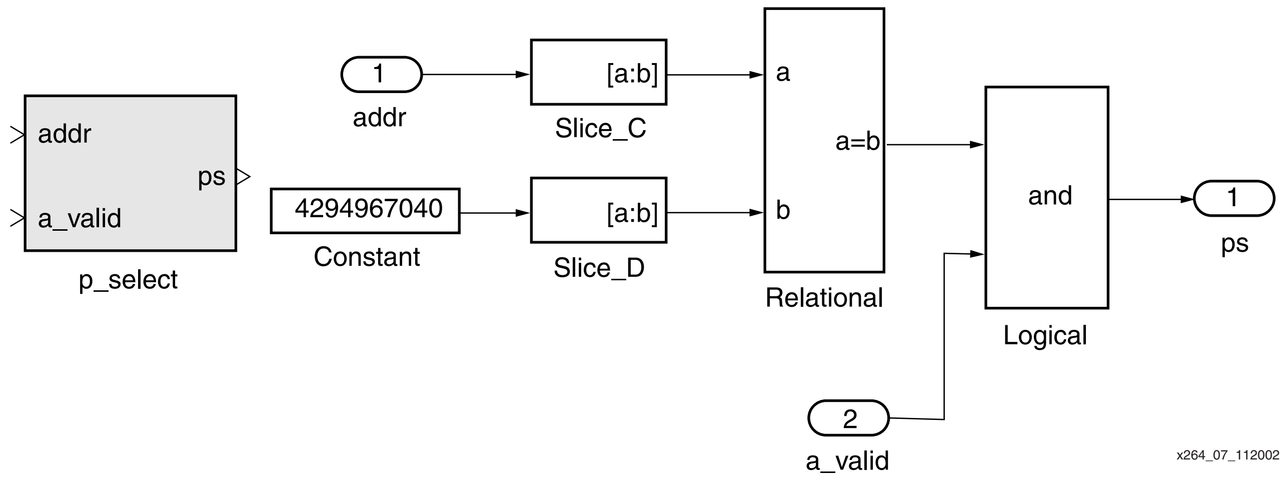
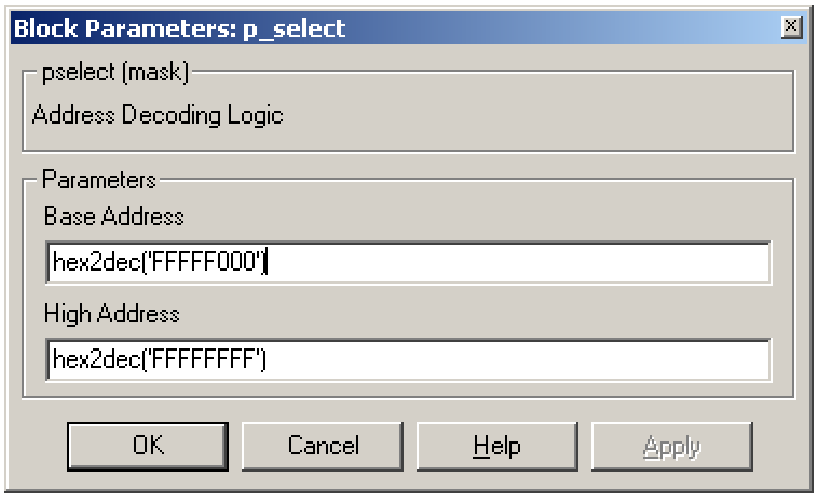


Figure 7: ***p_select*** **Address Decoding Subsystem**

Two slice blocks, *Slice_C* and *Slice_D* (Figure 7) extract the relevant bits of the address signal. The slice blocks are configured with a range defined as an offset from the MSB. The constant block stores the entire base address of the peripheral. The relational block tests for equality between the outputs of the two slice blocks. Finally, a logical block configured to perform an *and* operation ensures that the peripheral select output *ps* is only asserted when the address is valid, as indicated by the *a_valid* signal. Note, the *p_select* subsystem implementation assumes the memory map allocation range is an even power of two.

The usefulness of the *p_select* subsystem is further extended by converting it into a masked subsystem. The subsystem is parameterized in terms of the desired base and high address values for the peripheral model (Figure 8). The base and high address values are passed to mask parameters *C_BASE* and *C_HIGH* respectively.


Block Parameters: p_select

pselect (mask)

Address Decoding Logic

Parameters

Base Address

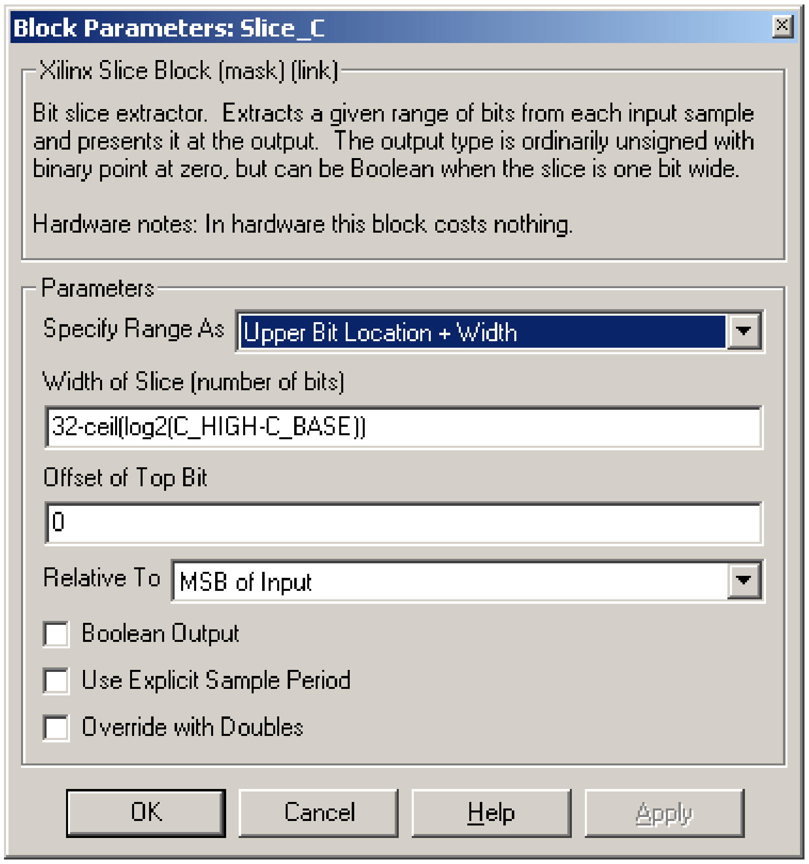
hex2dec('FFFFF000')

High Address

hex2dec('FFFFFFFF')

Figure 8: **Mask parameterization GUI for the *p_select* subsystem**

The close integration of System Generator with MATLAB allows blocks to be parameterized using MATLAB expressions. This flexibility allows the constant and slice blocks to be parameterized using the *C_BASE* and *C_HIGH* parameters. The slice blocks are identically parameterized and the corresponding mask GUI is shown in Figure 9.


Block Parameters: Slice_C

Xilinx Slice Block (mask) (link)

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.

Parameters

Specify Range As: Upper Bit Location + Width

Width of Slice (number of bits): 32-ceil(log2(C_HIGH-C_BASE))

Offset of Top Bit: 0

Relative To: MSB of Input

☐ Boolean Output

☐ Use Explicit Sample Period

☐ Override with Doubles

Figure 9: **Mask Parameterization GUI for the *Slice_C* Block**

Generating the Acknowledge

A slave peripheral either read or written to on the OPB must generate an acknowledge pulse once it has completed the transaction. This acknowledge must be accompanied by valid peripheral output data during a read. This pulse is driven to the *SGP_xferACK* signal of the OPB.

In the example peripheral, each read and write has a fixed and equal latency. Although a state machine is an equally valid alternative, this example uses a register to produce the pulse. This is the technique used in the tutorial "Designing Custom OPB Slave Peripherals for MicroBlaze"[4] to generate the acknowledge. The input to the register is driven by the peripheral select with extra logic to ensure the register resets on the cycle following its assertion.

Two additional registers introduce a two-cycle latency in the acknowledge thereby aligning the pulse with the peripheral output data. The three register outputs are fed back and are used to set the register to zero if any register output is a one. This logic is necessary because the peripheral select signal remains High for several cycles. The acknowledge signal, however, must only be asserted for a single cycle. The logic needed to generate the acknowledge pulse is shown in Figure 10.

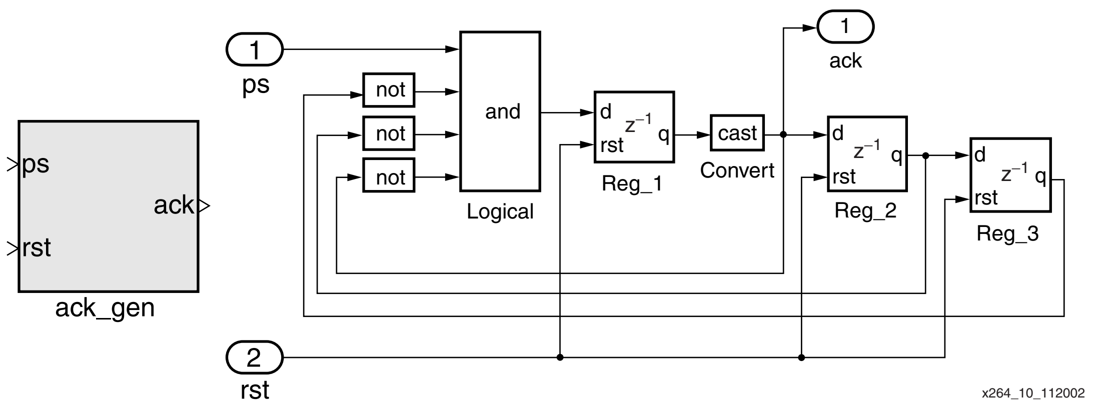


Figure 10: **Acknowledgement Generation Subsystem**

Defining a Memory Mapped I/O Interface

The peripheral communication interface with the OPB is defined in this section. This is typically realized through a memory mapped I/O interface where each port on the example data path is assigned an offset from the base address of the peripheral. There are five I/O ports of interest in the design's data path. The ports are assigned the offset values shown in Table 1.

Table 1: **I/O mapping for the DA FIR Filter Peripheral**

Signal	Description	Transfer	Offset
out_0_re	Data	Read	0x0
out_1_re	Buffer Full	Read	0x4
out_2_re	Buffer Empty	Read	0x8
in_0_we	Run/Stop	Write	0xC
in_1_we	Coefficient	Write	0x10

Additional decoding logic is included in the peripheral to generate the enable signals for each memory mapped register/FIFO component. The subsystem used to generate these enable signals is shown in Figure 11. A slice block extracts the relevant bits from the address signal. In the example peripheral, the addresses are aligned to the full 32-bit word boundaries. Therefore, ignore the two least significant bits of the address signal as they are not needed for data steering. The *OPB_BE* signal is not used in the model. The *OPB_rnw* signal and peripheral acknowledge signals are concatenated with the extracted address bits. The resulting signal drives the first input port of a series of comparators. A constant block drives the second input port of each comparator. The constant value is derived using the offset value of the memory mapped element along with the read/write status, and assumes an asserted acknowledge. The enable signals can now be wired to the enable ports of their respective memory mapped components.

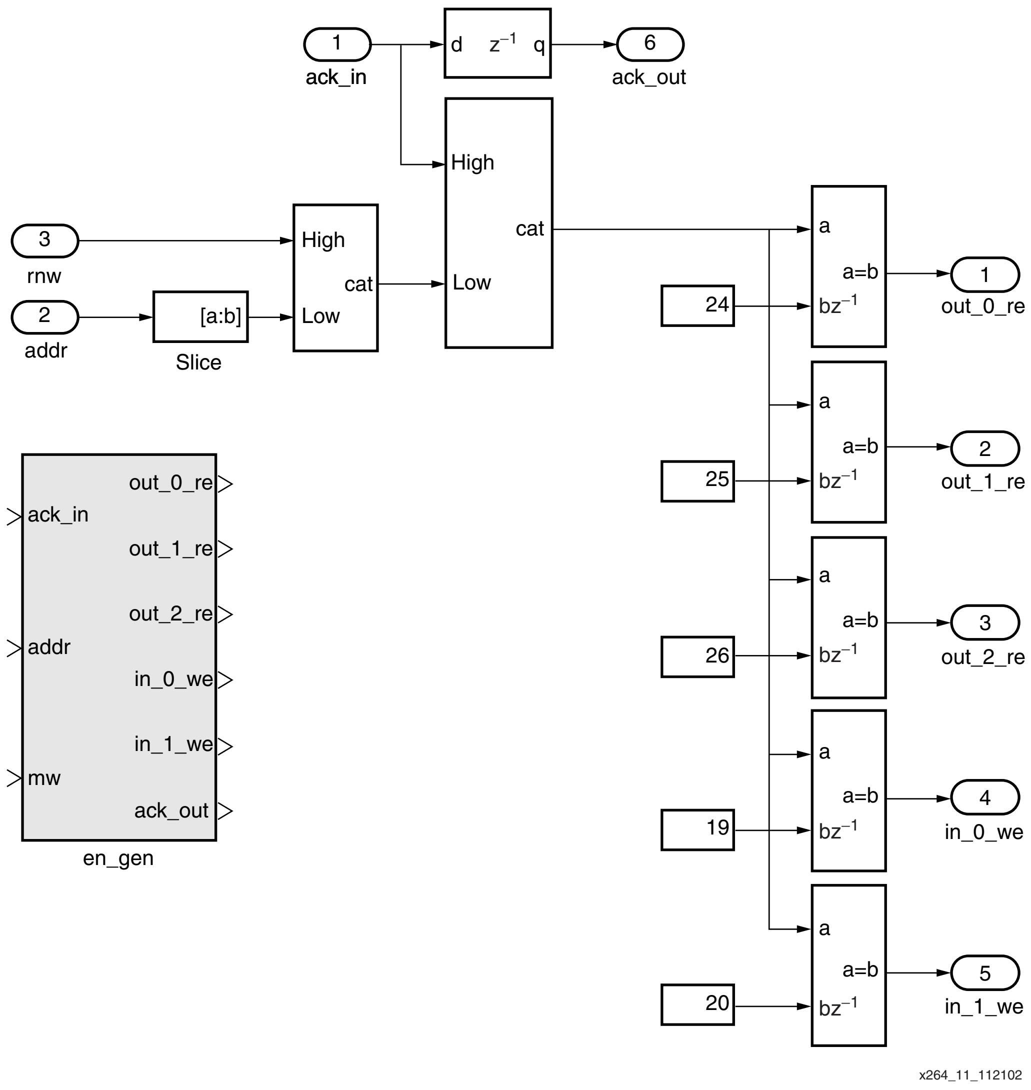


Figure 11: **Enable Generation Subsystem**

Having assigned peripheral I/O ports to addresses, a memory mapped interface is constructed using the Xilinx BlockSet. The memory map interface is partitioned into two subsystems, one for the peripheral inputs and the other for the peripheral outputs. The peripheral input memory map is considered first.

A standard memory-mapped input interface is comprised of register and FIFO blocks; both are naturally modeled and available in the Xilinx BlockSet. The input memory map for the example peripheral is implemented using a register block for the *run* control register and a FIFO block for filter coefficient buffering. Both the register and FIFO data inputs are driven by the OPB data input signal. Slice blocks are placed on both data input signals before the block inputs. For the *run* control, only a single bit for the control is required. This eliminates the need to use a 32-bit register to store the bus data. Instead, the slice block extracts the LSB from the data bus and generates a Boolean output signal. Likewise, the slice block for the FIFO extracts the 12 bits needed to store each filter coefficient. However, the precision produced by the slice block is incompatible with the filter coefficient precision required by the DA FIR filter block. The slice block generates an unsigned 12-bit number with zero factional bits. The parameterization of the DA FIR filter block requires signed 12-bit values with 11 fractional bits. To make this conversion, a force block is placed immediately after the *Slice_B* block. The force block does not require additional hardware resources and is only used to allow Simulink to correctly interpret and scale the coefficient data value.

Many System Generator blocks provide explicit enable and reset controls. These ports are mapped to the enable and reset ports in the synchronous hardware elements when the model is translated into hardware. These ports are used in the example memory map to control when the register and FIFO blocks are written to. An explicit enable signal is exposed on the register block and is driven by its respective *we* signal *in_0_we* from the address decoding logic. Similarly, the *we* port of the FIFO block is driven by the corresponding *we* signal, *in_1_we*. The corresponding input memory map subsystem is shown in Figure 12.

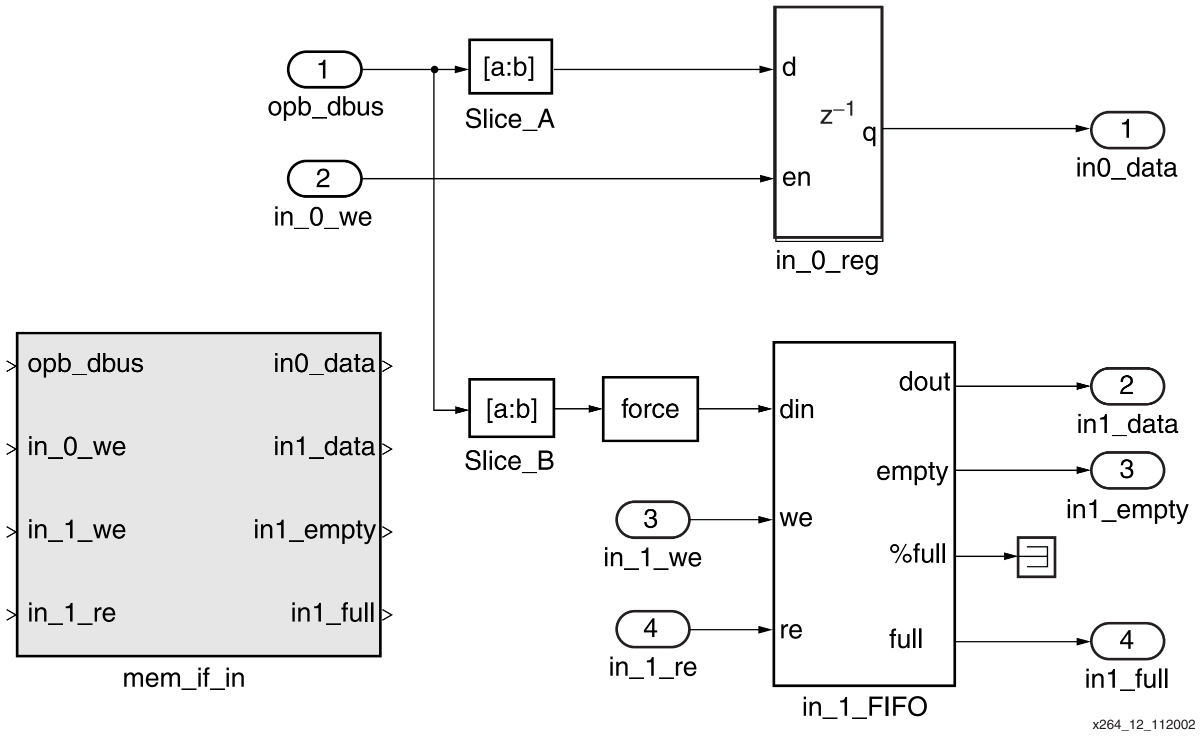


Figure 12: **Example Peripheral Input Memory Map**

The output memory map interface is comprised of a FIFO with multiplexing logic to switch between FIFO output signals. As shown in Table 1, three outputs from the FIFO: data, full, and empty are the predominant concern. In addition, the output memory map must drive zeros to the OPB data output signal if the master is not attempting to read from one of these three signals.

When a read request is issued to the peripheral and the address corresponds to one of these three signals, valid data must be driven to the bus data signal. The bus must have the valid data driven to it in the same cycle as when the acknowledge signal is asserted. The peripheral drives zeros at all other times to avoid bus contention.

The output memory map subsystem and logic are shown in Figure 13. A MUX block configured with four inputs is used to switch between constant zeros and the FIFO outputs. The FIFO outputs are all different widths, however, this is compensated for by using cast blocks to convert the output widths to 32 bits. The input to the subsystem are the three output read enable signals. These signals are concatenated together and drive the input of a ROM block. The ROM block is parameterized to decode the signal and drive the MUX select line with an appropriate value. If none of the read-enable signals are asserted, the MUX selects the constant 32-bit zero input.

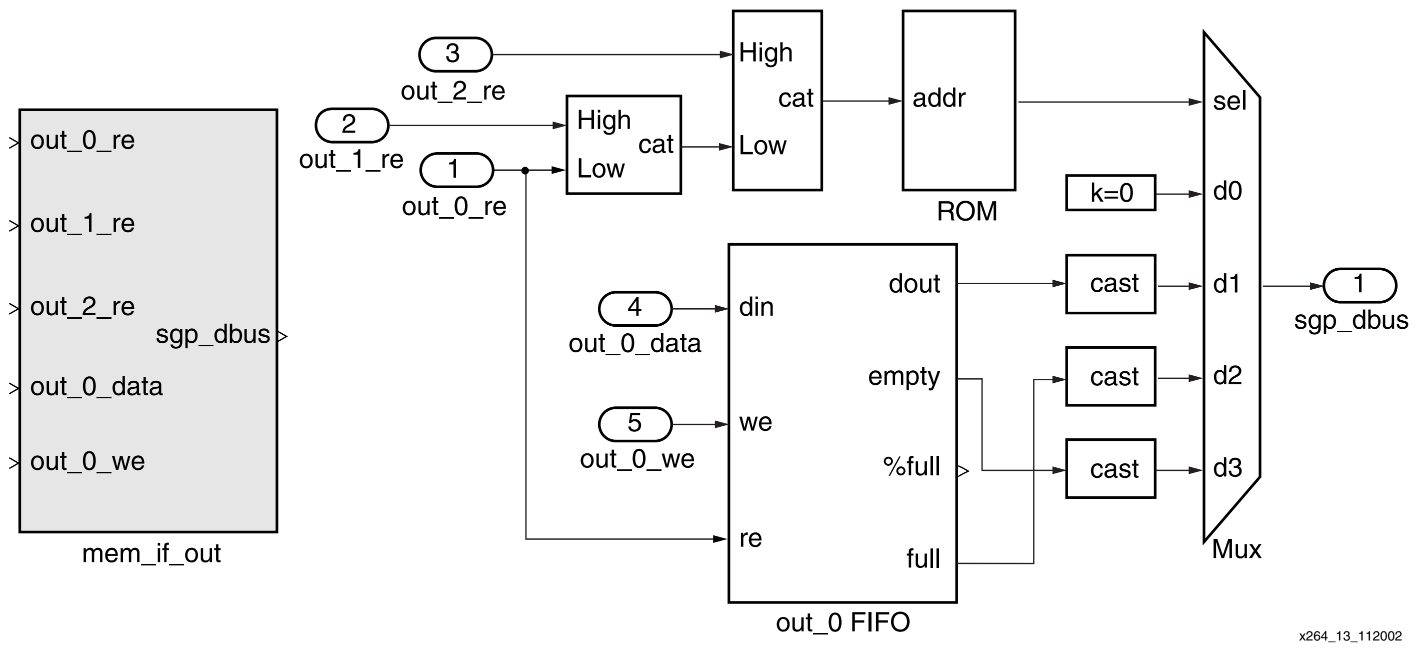


Figure 13: **Example Peripheral Output Memory Map**

Simulating the Peripheral

The complete peripheral implementation is shown in Figure 14.

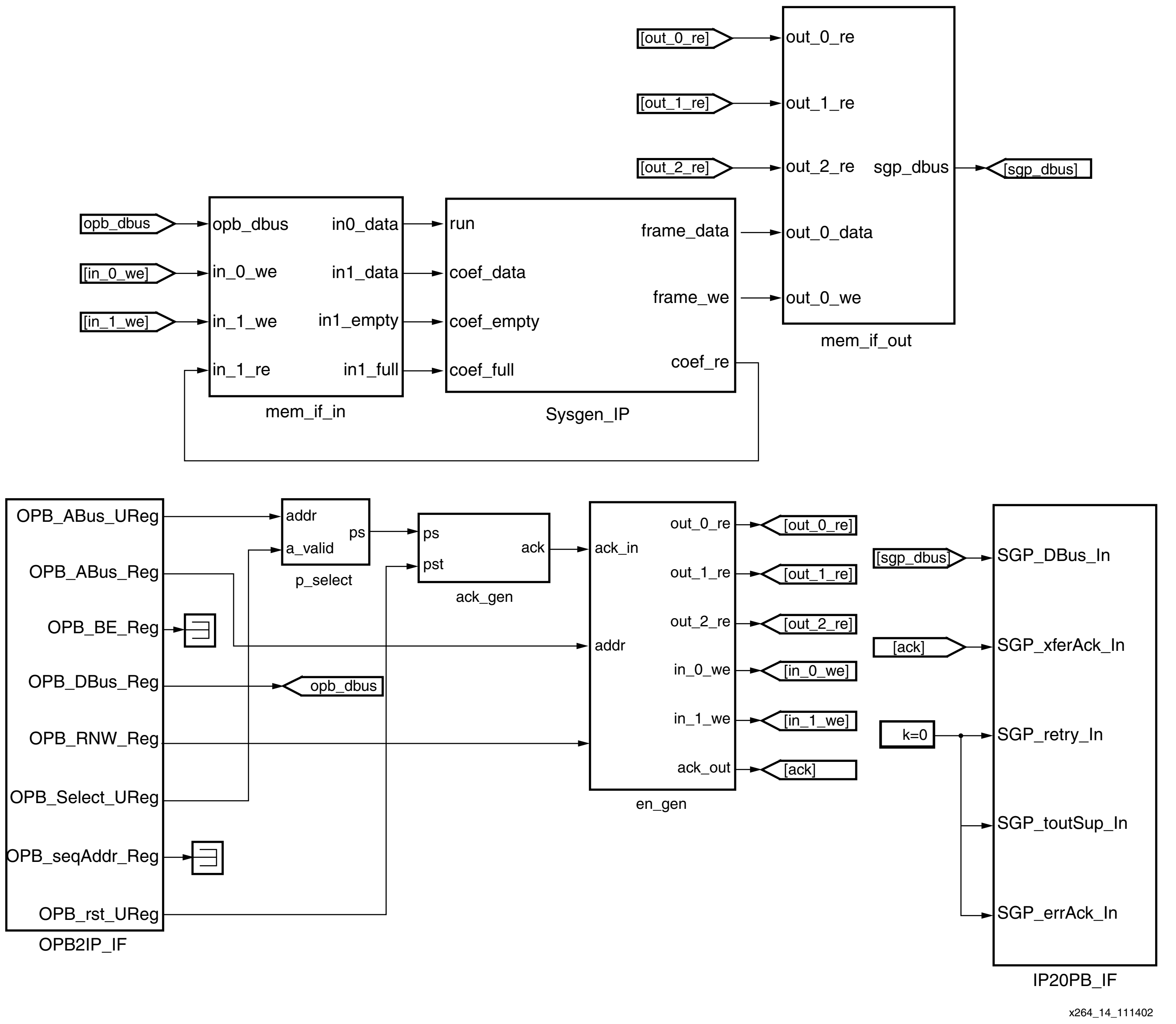


Figure 14: **Example Peripheral Implementation**

Simulink offers a variety of tools for simulating and debugging the peripheral model. These tools can be used by coupling the peripheral model with a bus stimuli model. StateFlow™, an event-driven interactive modeling and simulation tool from MathWorks, can be used as a tool to model basic processor behavior. By simulating subsets of the processor code using state transition diagrams, the user can better visualize peripheral model behavior under realistic stimuli. Simulating subsets of the processor code in Simulink is also advantageous as most analysis tools from existing Simulink libraries can be used in the peripheral debugging process. When the model is translated to hardware, System Generator automatically produces a test bench using the bus simulation test vectors as golden test vectors in the hardware simulation. By running these tests, the hardware representation is both bit and cycle accurate when compared to the behavior of the model.

For the example peripheral, a StateFlow diagram is implemented to model a MicroBlaze code stub. Each StateFlow diagram output drives a corresponding OPB input port of the input bus interface subsystem of the peripheral. Similarly, every StateFlow input is driven by an OPB output port from the output bus interface subsystem of the peripheral. Abstract connections to the bus interface subsystems are realized by the input and outputs of the StateFlow diagram driving or reading global from or goto blocks, respectively. This approach allows encapsulation of the StateFlow diagram, source, and syncs into a single subsystem. Because the input and outputs are wired to global from/goto blocks, bus signals can be tapped off accordingly with additional from/goto blocks. The model monitors the bus signals driven by the peripheral via global from blocks.

The tags on the global from/goto blocks match the from/goto tags found in bus interface blocks, *OPB2IP_IF* and *IP2OPB_IF*. The processor model in Figure 15 accepts a trigger condition; where the triggering is on the rising edge of the clock. A clock probe block extracts the system clock and drives the trigger port of the StateFlow diagram. The resulting StateFlow model with sample state transitions modeling a processor code stub are shown in Figure 15.

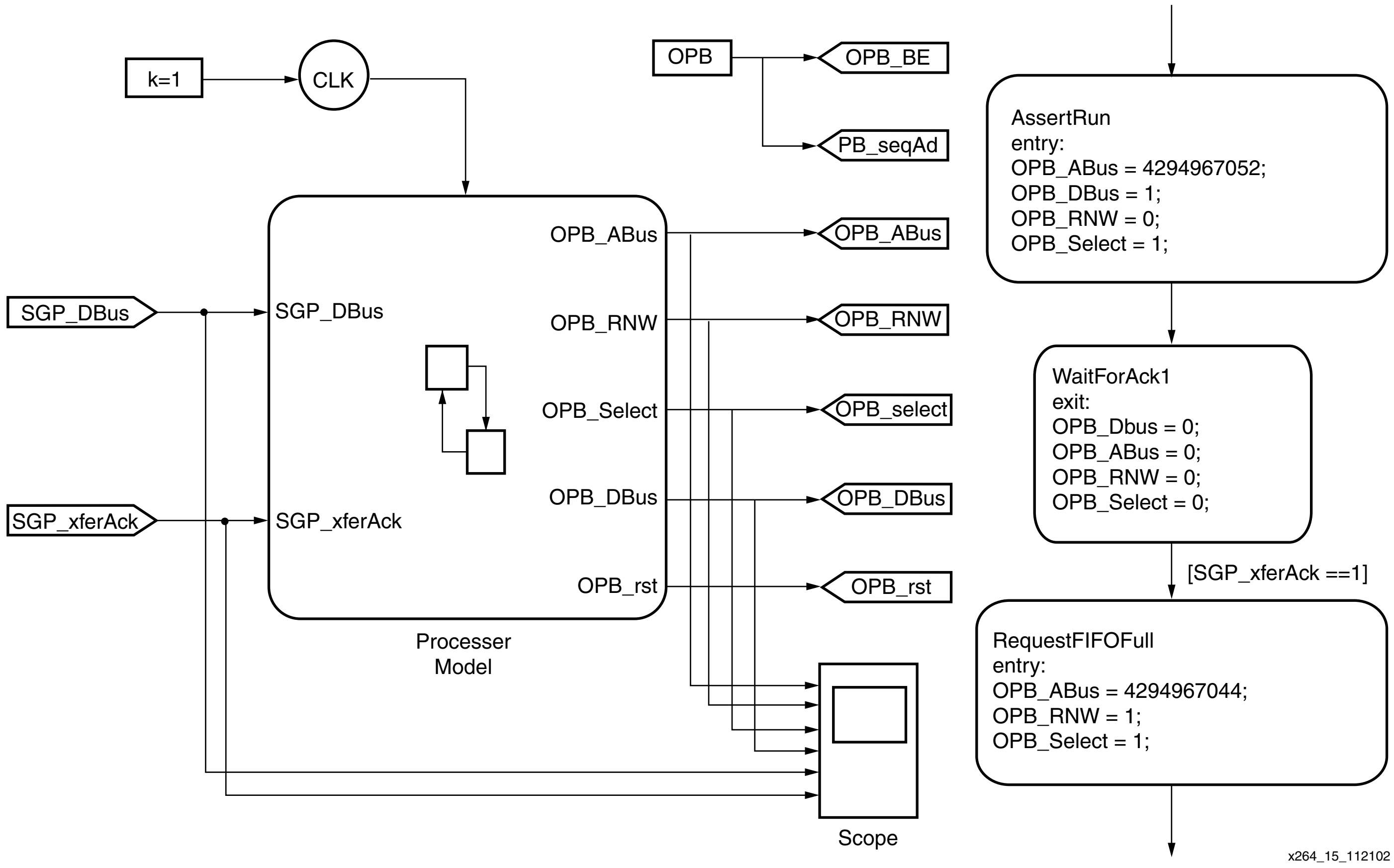


Figure 15: **StateFlow Diagram Block with Example State Transitions**

Included with each state transition in the diagram is a set of signal assignments producing a corresponding bus transaction (Figure 15). Using StateFlow allows easy, reproducible behavior of a processor code stub. In this case, the stub is focused solely on testing the functionality of the DSP peripheral, and not on the other components in the platform. The model is used only during simulation and is not translated in the hardware implementation.

Including the Peripheral in a Platform

Although System Generator produces all VHDL and netlist files necessary for the peripheral implementation, there are some modifications to make to the VHDL before it can be incorporated in a platform. In addition to making these modifications, the user must generate the required MPD, Peripheral Analyze Order (PAO), and Black-Box Definition (BBD) data files used to describe the peripheral.

There are several inconstancies between the VHDL signal formats produced by System Generator and the signal formats expected by the Platform Generator. Observe these differences and compensate for them by either producing an additional wrapper around the top-level to express the signals in the proper format or by adjusting the top-level System Generator VHDL accordingly. From a reusability standpoint, it is better to generate a top-level wrapper. The required changes are as follows:

- 1-bit wide System Generator VHDL signals are expressed using *std_logic_vector* types instead of *std_logic*. The Platform Generator expects *std_logic* signal types.
- If the peripheral model does not use a signal from the OPB, the signal will be optimized away and will not show up on the top-level VHDL port declaration. These ports must be explicitly re-added to the top-level VHDL file.
- System Generator defines bit ordering of the signal using little-Endian notation while the PowerPC and MicroBlaze operate under the assumption of big-Endian ordering. The signal ordering on the top-level VHDL ports must be reversed.

The PAO file lists all HDL files needed for synthesis of the peripheral. Refer to the Embedded System Tools Guide[3] for information about the PAO format. System Generator produces a file similar to the PAO file and with minor tweaking, this file can be converted into the correct PAO format. The *xst* project file contains a list of all VHDL files needed for synthesis. For example, the first five lines of the *xst_opb_sgp_filter.prj* file created by the software are shown in this sample VHDL file list from XST project file:

```
const_pkg.vhd
conv_pkg.vhd
clock_pkg.vhd
synth_reg.vhd
synth_reg_w_init.vhd
```

With only a few changes easily realized in a text editor, the project file can be converted into a PAO file. The corresponding five sample lines are from the PAO file:

```
lib opb_sgp_filter_v1_00_a const_pkg
lib opb_sgp_filter_v1_00_a conv_pkg
lib opb_sgp_filter_v1_00_a clock_pkg
lib opb_sgp_filter_v1_00_a synth_reg
lib opb_sgp_filter_v1_00_a synth_reg_w_init
```

Many System Generator blocks are mapped to handcrafted IP cores [5] when they are translated into hardware. Each core is distributed in EDIF netlist form. Copy every file with a *.edn* extension from the model's implementation directory into the *netlist* directory of the peripheral. For the EDK to copy the files over during implementation, the user must specify each netlist in the BBD file. The BBD file for the example peripheral follows.

```
##########################################################################
##
## Copyright (c) 2002 Xilinx, Inc.  All rights reserved.
##
## opb_sgp_filter_v2_0_0.bbd
##
## Black-Box Definition file
##
##########################################################################

FILES
opb_sgp_filter_xlcounter_free_x_0_core.edn,
opb_sgp_filter_xlfifo_x_0_core.edn, opb_sgp_filter_xlfifo_x_1_core.edn,
opb_sgp_filter_xlfir_reloadable_1ch_x_0_core.edn,
opb_sgp_filter_xlmux_x_0_core.edn,
opb_sgp_filter_xlrelational_x_7_core.edn,
opb_sgp_filter_xlsprom_dist_x_0_core.edn
```

In the MPD file, the STYLE parameter must be set to MIXED. Setting this parameter to MIXED informs the Platform Generator that the design has both HDL and optimized hardware netlist files. The MPD file for the reloadable filter peripheral follows.

```
##########################################################################
##
## Copyright (c) 2002 Xilinx, Inc.  All rights reserved.
##
## opb_sgp_filter_v2_0_0.mpd
##
## Microprocessor Peripheral Definition file
##
##########################################################################

PARAMETER VERSION = 2.0.0
BEGIN opb_sgp_filter, IPTYPE=PERIPHERAL, IMP_NETLIST=TRUE, STYLE=MIXED

BUS_INTERFACE BUS=SOPB, BUS_STD=OPB, BUS_TYPE=SLAVE

# Global ports
PORT clk = OPB_Clk, DIR=IN, SIGIS=CLK, BUS=SOPB
PORT opb2ip_if_opb_rst = OPB_Rst, DIR=IN, BUS=SOPB

# OPB signals
PORT opb2ip_if_opb_abus = OPB_ABus, DIR=IN, VEC=[0:31], BUS=SOPB
PORT opb2ip_if_opb_be = OPB_BE, DIR=IN, VEC=[0:3], BUS=SOPB
PORT opb2ip_if_opb_rnw = OPB_RNW, DIR=IN, BUS=SOPB
PORT opb2ip_if_opb_select = OPB_select, DIR=IN, BUS=SOPB
PORT opb2ip_if_opb_seqaddr = OPB_seqAddr, DIR=IN, BUS=SOPB
PORT opb2ip_if_opb_dbus = OPB_DBus, DIR=IN, VEC=[0:31], BUS=SOPB
PORT ip2opb_if_sgp_dbus = Sl_DBus, DIR=OUT, VEC=[0:31], BUS=SOPB
PORT ip2opb_if_sgp_errack = Sl_errAck, DIR=OUT, BUS=SOPB
PORT ip2opb_if_sgp_retry = Sl_retry, DIR=OUT, BUS=SOPB
PORT ip2opb_if_sgp_toutsup = Sl_toutSup, DIR=OUT, BUS=SOPB
PORT ip2opb_if_sgp_xferack = Sl_xferAck, DIR=OUT, BUS=SOPB

END
```

References

1. IBM, Inc. On-Chip Peripheral Bus: Architecture Specifications Version 2.1, http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/9A7AFA74DAD200D087256AB30005F0C8
2. Xilinx, Inc., Distributed Arithmetic FIR Filter V7.0, Product Specification. http://www.xilinx.com/ipcenter/catalog/logicore/docs/da_fir.pdf
3. Xilinx, Inc., Embedded System Tools Guide: Embedded Development Kit, EDK (v3.1 EA) September 24, 2002.
4. Xilinx, Inc., Tutorial: Designing Custom OPB Slave Peripherals for MicroBlaze, February 8, 2002. http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/opb_tutorial.pdf
5. Xilinx, Inc., Xilinx Core Generator System, http://www.xilinx.com/products/logiccore/coregen/index.htm.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/26/02	1.0	Initial Xilinx release.