

PROGRAMMING STREAMING FPGA APPLICATIONS USING BLOCK DIAGRAMS IN SIMULINK

Brian C. Richards
Chen Chang
John Wawrzynek
Robert W. Brodersen
University of California–Berkeley

Although a system designer can use hardware description languages, such as VHDL (Chapter 6) and Verilog to program FPGAs, the algorithm developer typically uses higher-level descriptions to refine an algorithm. As a result, an algorithm described in a language such as Matlab or C is frequently reentered by hand by the system designer, after which the two descriptions must be verified and refined manually. This can be time consuming.

To avoid reentering a design when translating from a high-level simulation language to HDL, the algorithm developer can describe a system from the beginning using block diagrams in Matlab Simulink [1]. Other block diagram environments can be used in a similar way, but the tight integration of Simulink with the widely used Matlab simulation environment allows developers to use familiar data analysis tools to study the resulting designs. With Simulink, a single design description can be prepared by the algorithm developer and refined jointly with the system architect using a common design environment.

The single design entry is enabled by a library of Simulink operator primitives that have a direct mapping to HDL, using matching Simulink and HDL models that are cycle accurate and bit accurate between both domains. Examples and compilation environments include System Generator from Xilinx [2], Synplify DSP from Synplicity [3], and the HDL Coder from The Mathworks [1]. Using such a library, nearly any synchronous multirate system can be described, with high confidence that the result can be mapped to an FPGA given adequate resources.

In this chapter, a high-performance image-processing system is described using Simulink and mapped to an FPGA-based platform using a design flow built around the Xilinx System Generator tools. The system implements edge detection in real time on a digitized video stream and produces a corresponding video stream labeling the edges. The edges can then be viewed on a high-resolution monitor. This design demonstrates how to describe a high-performance parallel

datapath, implement control subsystems, and interface to external devices, including embedded processors.

8.1 DESIGNING HIGH-PERFORMANCE DATAPATHS USING STREAM-BASED OPERATORS

Within Simulink we employ a Synchronous Dataflow computational model (SDF), [described in the Synchronous Dataflow, Section 5.1.3, of Chapter 5](#). Each operator is executed once per clock cycle, consuming input values and producing new output values once per clock tick. This discipline is well suited for stream-based design, encouraging both the algorithm designer and the system architect to describe efficient datapaths with minimal idle operations.

Clock signals and corresponding clock enable signals do not appear in the Simulink block diagrams using the System Generator libraries, but are automatically generated when an FPGA design is compiled. To support multirate systems, the System Generator library includes up-sample and down-sample blocks to mark the boundaries of different clock domains. When compiled to an FPGA, clock enable signals for each clock domain are automatically generated.

All System Generator components offer compile time parameters, allowing the designer to control data types and refine the behavior of the block. Hierarchical blocks, or *subsystems* in Simulink, can also have user-defined parameters, called *mask parameters*. These can be included in block property expressions within that subsystem to provide a means of generating a variety of behaviors from a single Simulink description. Typical mask parameters include data type and precision specification and block latency to control pipeline stage insertion. For more advanced library development efforts, the mask parameters can be used by a Matlab program to create a custom schematic at compile time.

The System Generator library supports fixed-point or Boolean data types for mapping to FPGAs. Fixed-point data types include signed and unsigned values, with bit width and decimal point location as parameters. In most cases, the output data types are inferred automatically at compile time, although many blocks offer parameters to define them explicitly.

Pipeline operators are explicitly placed into a design either by inserting delay blocks or by defining a delay parameter in selected functional blocks. Although the designer is responsible for balancing pipeline operators, libraries of high-level components have been developed and reused to hide pipeline-balancing details from the algorithm developer.

The Simulink approach allows us to describe highly concurrent SDF systems where many operators—perhaps the entire dataflow path—can operate simultaneously. With modern FPGAs, it is possible to implement these systems with thousands of simultaneous operators running at the system clock rate with little or no control logic, allowing complex, high performance algorithms to be implemented.

8.2 AN IMAGE-PROCESSING DESIGN DRIVER

The goal of the edge detection design driver is to generate a binary bit mask from a video source operating at up to a 200 MHz pixel rate, identifying where likely edges are in an image. The raw color video is read from a neighboring FPGA over a parallel link, and the image intensity is then calculated, after which two 3×3 convolutional Sobel operator filters identify horizontal and vertical edges; the sum of their absolute values indicates the relative strength of a feature edge in an image. A runtime programmable gain (variable multiplier) followed by an adjustable threshold maps the resulting pixel stream to binary levels to indicate if a given pixel is labeled as an edge of a visible feature. The resulting video mask is then optionally mixed with the original color image and displayed on a monitor.

Before designing the datapaths in the edge detection system, the data and control specification for the video stream sources and sinks must be defined. By convention, stream-based architectures are implemented by pairing data samples with corresponding control tags and maintaining this pairing through the architecture. For this example, the video data streams may have varying data types as the signals are processed whereas the control tags are synchronization signals that track the pipeline delays in the video stream. The input video stream and output display stream represent color pixel data using 16 bits—5 bits for red, 6 bits for green, and 5 bits for blue unsigned pixel intensity values. Intermediate values might represent video data as 8-bit grayscale intensity values or as 1-bit threshold detection mask values.

As the data streams flow through the signal-processing datapath, the operators execute at a constant 100 MHz sample rate, with varying pipeline delays through the system. The data, however, may arrive at less than 100 MHz, requiring a corresponding `enable` signal (see the discussion of data presence subsection in Chapter 5, Section 5.2.1) to tag valid data. Additionally, `hsync`, `vsync`, and `msync` signals are defined to be true for the first pixel of each row, frame, and movie sequence, respectively, allowing a large variety of video stream formats to be supported by the same design.

Once a streaming format has been specified, library components can be developed that forward a video stream through a variety of operators to create higher-level functions while maintaining valid, pipeline-delayed synchronization signals. For blocks with a pipeline latency that is determined by mask parameters, the synchronization signals must also be delayed based on the mask parameters so that the resulting synchronization signals match the processed data stream.

8.2.1 Converting RGB Video to Grayscale

The first step in this example is to generate a grayscale video stream from the RGB input data. The data is converted to intensity using the NTSC RGB-to-Y matrix:

$$Y = 0.3 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$$

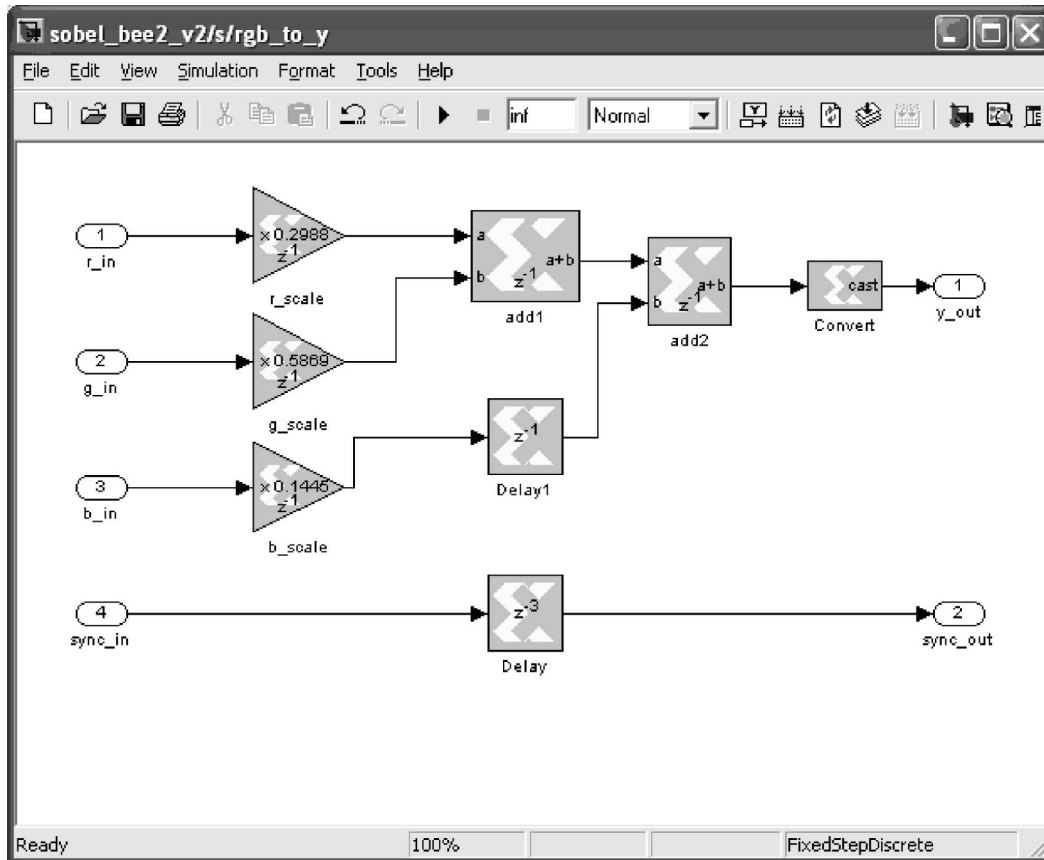


FIGURE 8.1 ■ An RGB-to-Y (intensity) Simulink diagram.

This formula is implemented explicitly as a block diagram, shown in Figure 8.1, using constant gain blocks followed by adders. The constant multiplication values are defined as floating-point values and are converted to fixed-point according to mask parameters in the gain model. This allows the precision of the multiplication to be defined separately from the gain, leaving the synthesis tools to choose an implementation. The scaled results are then summed with an explicit adder tree.

Note that if the first adder introduces a latency of $adder_delay$ clock cycles, the b input to the second adder, $add2$, must also be delayed by $adder_delay$ cycles to maintain the cycle alignment of the RGB data. Both the `Delay1` block and the `add1` block have a subsystem mask parameter defining the delay that the block will introduce, provided by the mask parameter `dialogue` as shown in Figure 8.2. Similarly, the synchronization signals must be delayed by three cycles corresponding to one cycle for the gain blocks, one cycle for the first adder,

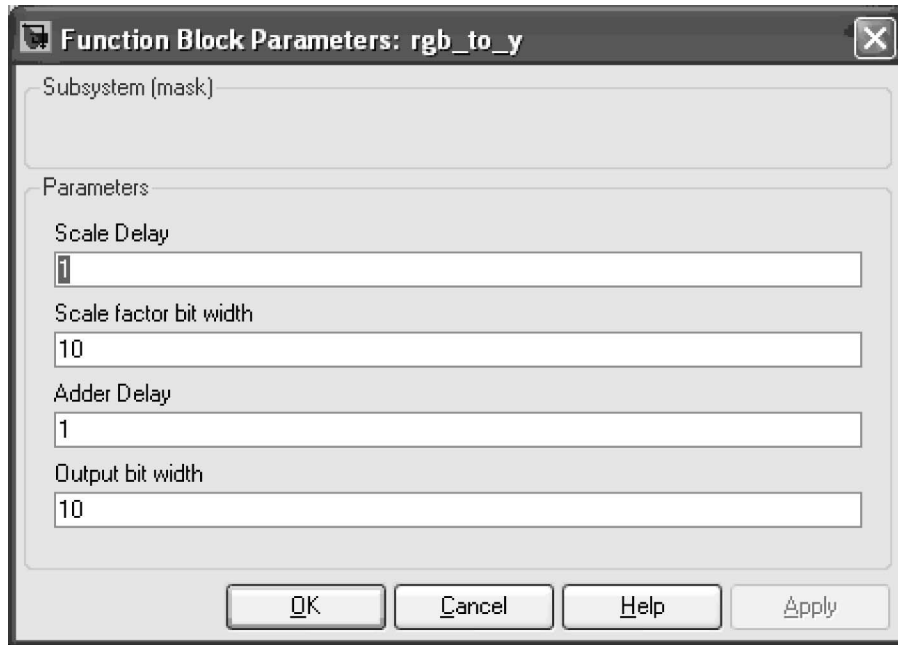


FIGURE 8.2 ■ A dialog describing mask parameters for the `rgb_to_y` block.

and one cycle for the second adder. By designing subsystems with configurable delays and data precision parameters, library components can be developed to encourage reuse of design elements.

8.2.2 Two-dimensional Video Filtering

The next major block following the RGB-to-grayscale conversion is the edge detection filter itself (Figure 8.3), consisting of two pixel row delay lines, two 3×3 kernels, and a simplified magnitude detector. The delay lines store the two rows of pixels preceding the current row of video data, providing three streams of vertically aligned pixels that are connected to the two 3×3 filters—the first one detecting horizontal edges and the second detecting vertical edges. These filters produce two signed fixed-point streams of pixel values, approximating the edge gradients in the source video image.

On every clock cycle, two 3×3 convolution kernels must be calculated, requiring several parallel operators. The operators implement the following convolution kernels:

Sobel X Gradient:	<table style="border-collapse: collapse; text-align: center;"> <tr><td>-1</td><td>0</td><td>+1</td></tr> <tr><td>-2</td><td>0</td><td>+2</td></tr> <tr><td>-1</td><td>0</td><td>+1</td></tr> </table>	-1	0	+1	-2	0	+2	-1	0	+1	Sobel Y Gradient:	<table style="border-collapse: collapse; text-align: center;"> <tr><td>+1</td><td>+2</td><td>+1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>-1</td><td>-2</td><td>-1</td></tr> </table>	+1	+2	+1	0	0	0	-1	-2	-1
-1	0	+1																			
-2	0	+2																			
-1	0	+1																			
+1	+2	+1																			
0	0	0																			
-1	-2	-1																			

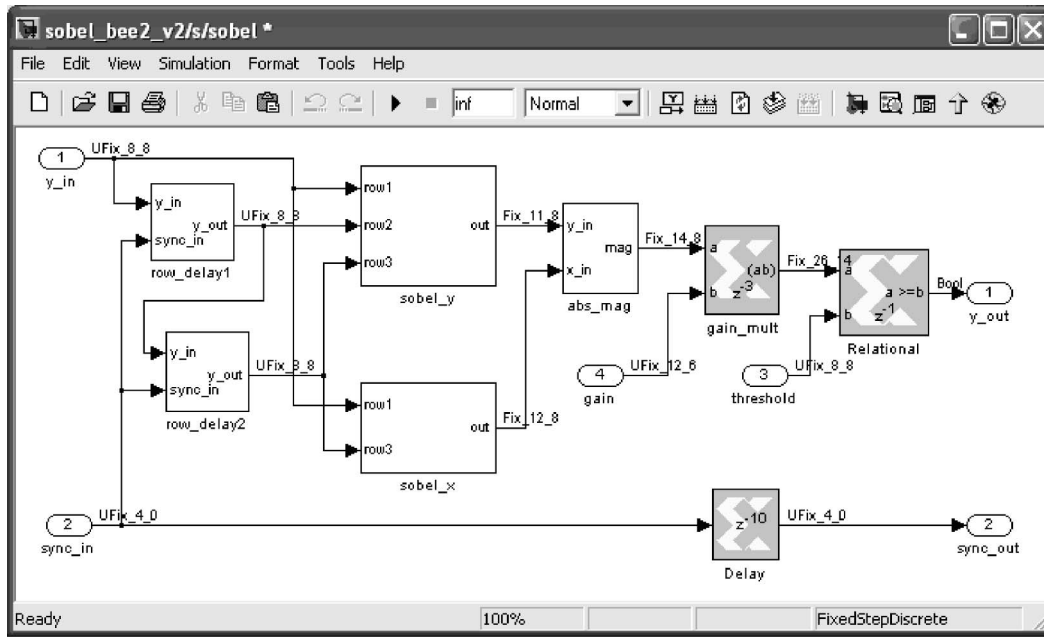


FIGURE 8.3 ■ The Sobel edge detection filter, processing an 8-bit video datastream to produce a stream of Boolean values indicating edges in the image.

To support arbitrary kernels, the designer can choose to implement the Sobel operators using constant multiplier or gain blocks followed by a tree of adders. For this example, the subcircuits for the x - and y -gradient operators are hand-optimized so that the nonzero multipliers for both convolution kernels are implemented with a single hardwired shift operation using a power-of-2 scale block. The results are then summed explicitly, using a tree of add or subtract operators, as shown in Figures 8.4 and 8.5.

Note that the interconnect in Figures 8.4 and 8.5 is shown with the data types displayed. For the most part, these are assigned automatically, with the input data types propagated and the output data types and bit widths inferred to avoid overflow or underflow of signed and unsigned data types. The bit widths can be coerced to different data types and widths using casting or reinterpret blocks, and by selecting saturation, truncation, and wraparound options available to several of the operator blocks. The designer must exercise care to verify that such adjustments to a design do not change the behavior of the algorithm.

Through these Simulink features a high-level algorithm designer can directly explore the impact of such data type manipulation on a particular algorithm.

Once the horizontal and vertical intensity gradients are calculated for the neighborhood around a given pixel, the likelihood that the pixel is near the boundary of a feature can be calculated. To label a pixel as a likely edge of a feature in the image, the magnitude of the gradients is approximated and the resulting nonnegative value is scaled and compared to a given threshold. The

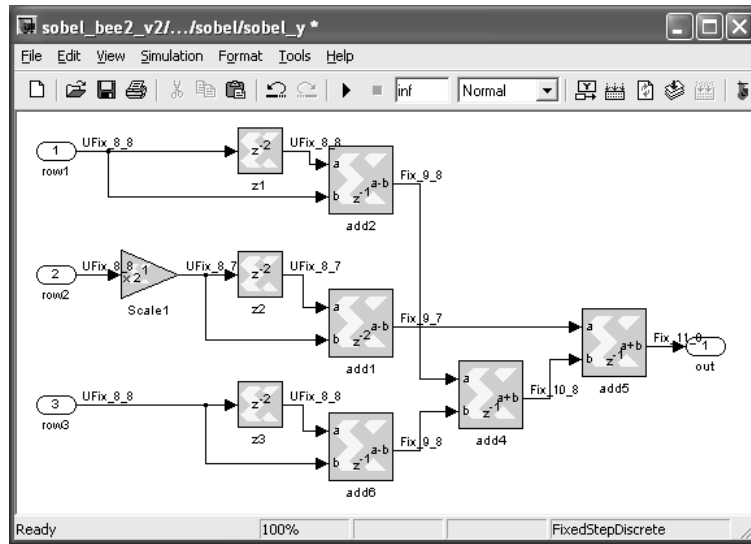


FIGURE 8.4 ■ The `sobel_y` block for estimating the horizontal gradient in the source image.

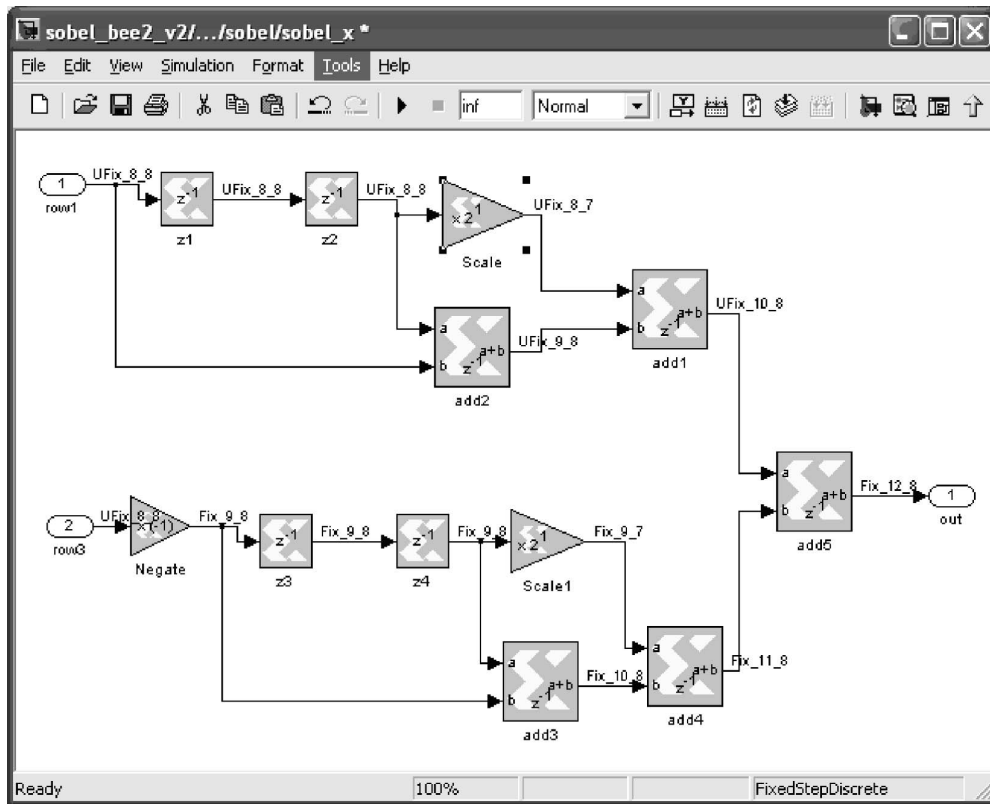


FIGURE 8.5 ■ The `sobel_x` block for estimating the vertical gradient in the source image.

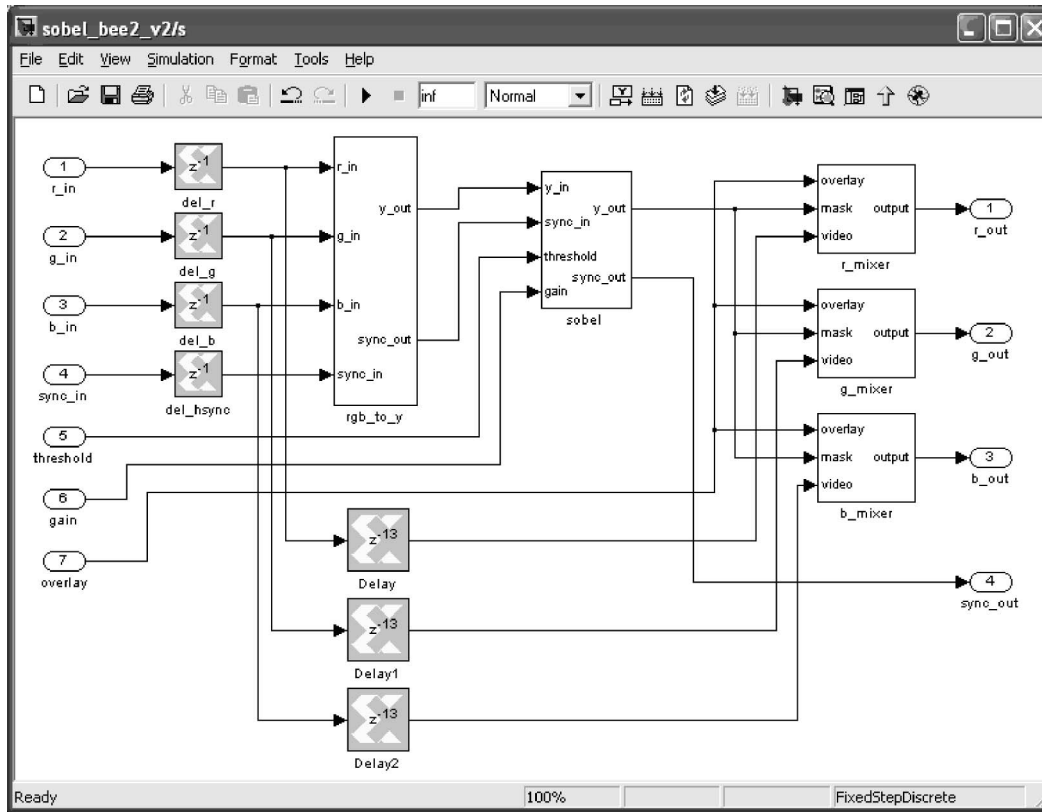


FIGURE 8.7 ■ The main filtering subsystem, with RGB-to-Y, Sobel, and mixer blocks.

color data fed straight through to the red, green, and blue mixers is delayed. The delay, 13 clock cycles in this case, corresponds to the pipeline delay through both the `rgb_to_y` block and the Sobel edge detection filter itself. This is to ensure that the background original image data is aligned with the corresponding pixel results from the filter. The sync signals are also delayed, but this is propagated through the filtering blocks and does not require additional delays.

8.2.3 Mapping the Video Filter to the BEE2 FPGA Platform

Our design, up to this point, is platform independent—any Xilinx component supported by the System Generator commercial design flow can be targeted. The next step is to map the design to the BEE2 platform—a multiple-FPGA design, developed at UC Berkeley [4], that contains memory to store a stream of video data and an HDMI interface to output that data to a high-resolution monitor.

For the Sobel edge detection design, some ports are for video data streams and others are for control over runtime parameters. The three user-controllable inputs to the filtering subsystem, `threshold`, `gain`, and `overlay`, are connected

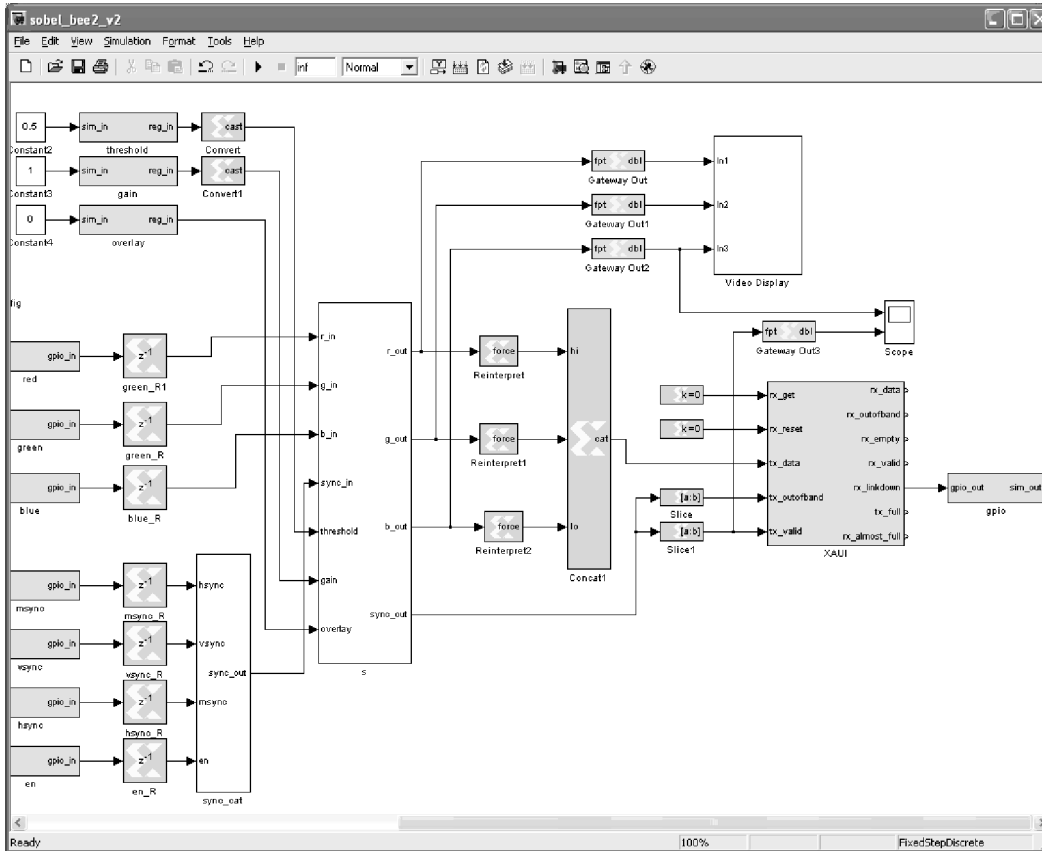


FIGURE 8.9 ■ The output section of the top-level testbench, with a 10G XAUI interface block.

description to be simulated with a software testbench to verify basic functionality before mapping the design to hardware. They also allow the designer to assign the FPGA pin locations for the final configuration files.

The BEE Platform Studio (BPS) [5] provides additional I/O blocks that allow the designer to select pin locations symbolically, choosing pins that are hardwired to other FPGAs, LEDs, and external connections on the platform. The designer is only required to select a platform by setting BPS block parameters, and does not need to keep track of I/O pin locations. This feature allows the designer to experiment with architectural tradeoffs without becoming a hardware expert.

In addition to the basic I/O abstractions, the BPS allows high-performance or analogue I/O devices to be designed into a system using high-level abstractions. For the video-testing example, a 10 Gbit XAUI I/O block is used to output the color video stream to platform-specific external interfaces. The designer selects the port to be used on the actual platform from a pulldown menu of available names, hiding most implementation details.

A third category of platform-specific I/O enables communication with embedded microprocessors, such as the Xilinx MicroBlaze soft processor core or the embedded PowerPC available on several FPGAs. Rather than describe the details of the microprocessor subsystem, the designer simply selects which processor on a given platform will be used and a preconfigured platform-specific microprocessor subsystem is then generated and included in the FPGA configuration files. For the video filter example, three microprocessor registers are assigned and connected to the threshold, gain, and overlap inputs to the filter using general-purpose I/O (GPIO) blocks. When the BPS design flow is run, these CPU register blocks are mapped to general-purpose I/O registers on the selected platform, and C header files are created to define the memory addresses for the registers.

8.3 SPECIFYING CONTROL IN SIMULINK

On the one hand, Simulink is well suited to describing highly pipelined stream-based systems with minimal control overhead, such as the video with synchronization signals described in the earlier video filter example. These designs assume that each dataflow operator is essentially running in parallel, at the full clock rate. On the other hand, control tasks, such as state machines, tend to be inherently sequential and can be more challenging to describe efficiently in Simulink. Approaches to describing control include the following:

- Counters, registers, and logic to describe controllers
- Matlab M-code descriptions of control blocks
- VHDL or Verilog hand-coded or compiled descriptions
- Embedded microprocessors

To explore the design of control along with a stream-based datapath, consider the implementation of a synchronous delay line based on a single-port memory. The approach described here is to alternate between writing two data samples and reading two data samples on consecutive clock cycles. A simpler design could be implemented using dual-port memory on an FPGA, but the one we are using allows custom SOC designs to use higher-density single-port memory blocks.

8.3.1 Explicit Controller Design with Simulink Blocks

The complete synchronous delay line is shown in Figure 8.10. The control in this case is designed around a counter block, where the least significant bit selects between the two words read or written from the memory on a given cycle and the upper counter bits determine the memory address. In addition to the counter, control-related blocks include *slice* blocks to select bit fields and Boolean *logic* blocks. For this design, the block diagram is effective for describing control, but minor changes to the controller can require substantial redesign.

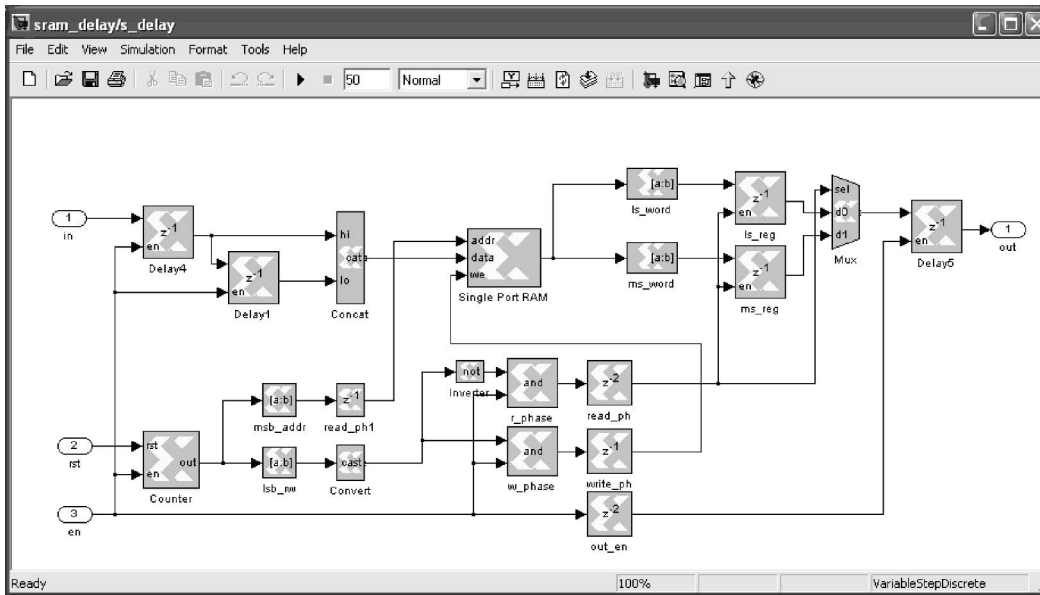


FIGURE 8.10 ■ A simple datapath with associated explicit control.

8.3.2 Controller Design Using the Matlab M Language

For a more symbolic description of the synchronous delay line controller, the designer can use the Matlab “M” language to define the behavior of a block, with the same controller described previously written as a Matlab function. Consider the code in Listing 8.1 that is saved in the file `sram_delay_cntl.m`.

Listing 8.1 ■ The delay line controller described with the Matlab function `sramdelaycntl.m`.

```
function [addr, we, sel] = sramdelaycntl(rst, en, counterbits, countermax)
% sramdelaycntl -- MCode implementation block.
% Author: Brian Richards, 11/16/2005, U. C. Berkeley
%
% The following Function Parameter Bindings should be declared in
% the MCode block Parameters (sample integer values are given):
%   {'counterbits', 9, 'countermax', 5}

% Define all registers as persistent variables.
persistent count,
    count = xlstate(0, { xlUnsigned, counterbits, 0});
persistent addrreg,
    addrreg = xlstate(0, { xlUnsigned, counterbits-1, 0});
persistent wereg,   wereg = xlstate(0, { xlBoolean});
persistent selreg1, selreg1 = xlstate(0, { xlBoolean});
persistent selreg2, selreg2 = xlstate(0, { xlBoolean});

% Delay the counter output, and split the lsb from
% the upper bits.
```

```

addr = addrreg;
addrreg = xlslice(count, counterbits-1, 1);
countlsb = xfix({ xlBoolean}, xlslice(count, 0, 0));
% Write-enable logic
we = wereg;
wereg = countlsb & en;

% MSB-LSB select logic
sel = selreg2;
selreg2 = selreg1;
selreg1 = ~countlsb & en;

% Update the address counter:
if (rst | (en & (count == countermax)))
    count = 0;
elseif (en)
    count = count + 1;
else
    count = count;
end

```

To add the preceding controller to a design, the Xilinx M-code block can be dragged from the Simulink library browser and added to the subsystem. A [dialog box](#) then asks the designer to select the file containing the M source code, and the block `sram_delay_cnt1` is automatically created and added to the system (see Figure 8.11).

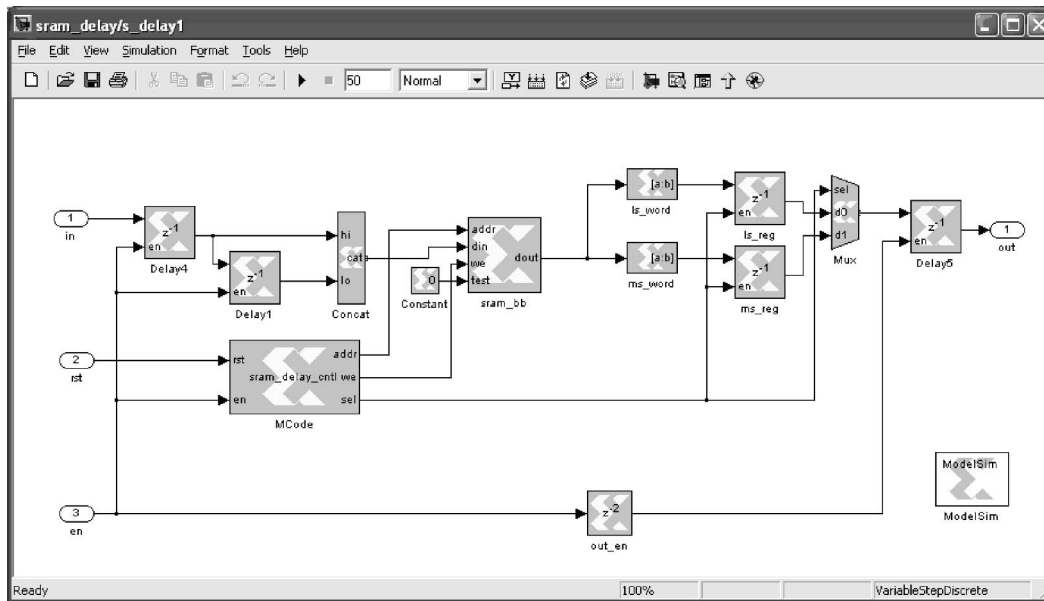


FIGURE 8.11 ■ A simple datapath using a controller described in Matlab code.

There are several advantages to using the M code description compared to its explicit block diagram equivalent. First, large, complex state machines can be described and documented efficiently using the sequential M language. Second, the resulting design will typically run faster in Simulink because many fine-grained blocks are replaced by a single block. Third, the design is mapped to an FPGA by generating an equivalent VHDL RTL description and synthesizing the resulting controller; the synthesis tools can produce different results depending on power, area, and speed constraints, and can optimize for different FPGA families.

8.3.3 Controller Design Using VHDL or Verilog

As in the M language approach just described, a controller can also be described with a *black box* containing VHDL or Verilog source code. This approach can be used for both control and datapath subsystems and has the benefit of allowing IP to be included in a Simulink design.

The VHDL or Verilog subsystems must be written according to design conventions to ensure that the subsystem can be mapped to hardware. Clocks and enables, for example, do not appear on the generated Simulink block, but must be defined in pairs (e.g., `clk_sg`, `ce_sg`) for each implied data rate in the system. Simulink designs that use these VHDL or Verilog subsystems can be verified by cosimulation between Simulink and an external HDL simulator, such as Modelsim [6]. Ultimately, the same description can be mapped to hardware, assuming that the hardware description is synthesizable.

8.3.4 Controller Design Using Embedded Microprocessors

The most elaborate controller for an FPGA is the embedded microprocessor. In this case, control can be defined by running compiled or interpreted programs on the microprocessor. On the BEE2 platform, a tiny shell can be used interactively to control datapath settings, or a custom C-based program can be built using automatically generated header files to symbolically reference hardware devices.

A controller implemented using an embedded microprocessor is often much slower than the associated datapath hardware, perhaps taking several clock cycles to change control parameters. This is useful for adjusting parameters that do not change frequently, such as threshold, gain, and overlay in the Sobel filter. The BEE Platform Studio design flow uses the Xilinx Embedded Development Kit (EDK) to generate a controller running a command line shell, which allows the user to read and modify configuration registers and memory blocks within the FPGA design. Depending on the platform, this controller can be accessed via a serial port, a network connection, or another interface port.

The same embedded controller can also serve as a source or sink for low-bandwidth data streams. An example of a user-friendly interface to such a source or sink is a set of Linux 4.2 kernel extensions developed as part of the BEE operating system, BORPH [7]. BORPH defines the notion of a hardware process, where a bit file and associated interface information is encapsulated in an

executable `.bof` file. When launched from the Linux command line, a software process is started that programs and then communicates with the embedded processor on a selected FPGA. To the end user, hardware sources and sinks in Simulink are mapped to Linux files or pipes, including standard input and standard output. These file interfaces can then be accessed as software streams to read from or write to a stream-based FPGA design for debugging purposes or for applications with low-bandwidth continuous data streams.

8.4 COMPONENT REUSE: LIBRARIES OF SIMPLE AND COMPLEX SUBSYSTEMS

In the previous sections, low-level primitives were described for implementing simple datapath and control subsystems and mapping them to FPGAs. To make this methodology attractive to the algorithm developer and system architect, all of these capabilities are combined to create reusable library components, which can be parameterized for a variety of applications; many of them have been tested in a variety of applications.

8.4.1 Signal-processing Primitives

One example of a rich library developed for the BPS is the Astronomy library, which was co-developed by UC Berkeley and the Space Sciences Laboratory [8,9] for use in a variety of high-performance radio astronomy applications. In its simplest form, this library comprises a variety of complex-valued operators based on Xilinx System Generator real-valued primitives. These blocks are implemented as Simulink subsystems with optional parameters defining latency or data type constraints.

8.4.2 Tiled Subsystems

To enable the development of more sophisticated library components, Simulink supports the use of Matlab M language programs to create or modify the schematic within a subsystem based on parameters passed to the block. With the Simulink Mask Editor, initialization code can be added to a subsystem to place other Simulink blocks and to add interconnect to define a broad range of implementations for a single library component.

Figure 8.12 illustrates an example of a tiled cell, the `biplex_core` FFT block, which accepts several implementation parameters. The first parameters define the size and precision of the FFT operator, followed by the quantization behavior (truncation or rounding) and the overflow behavior of adders (saturation or wrapping). The pipeline latencies of addition and multiplication operators are also user selectable within the subsystem.

Automatically tiled library components can conditionally use different subsystems, and can have multiple tiling dimensions. An alternative to the stream-based `biplex_core` block shown in Figure 8.13, a parallel FFT implementation,

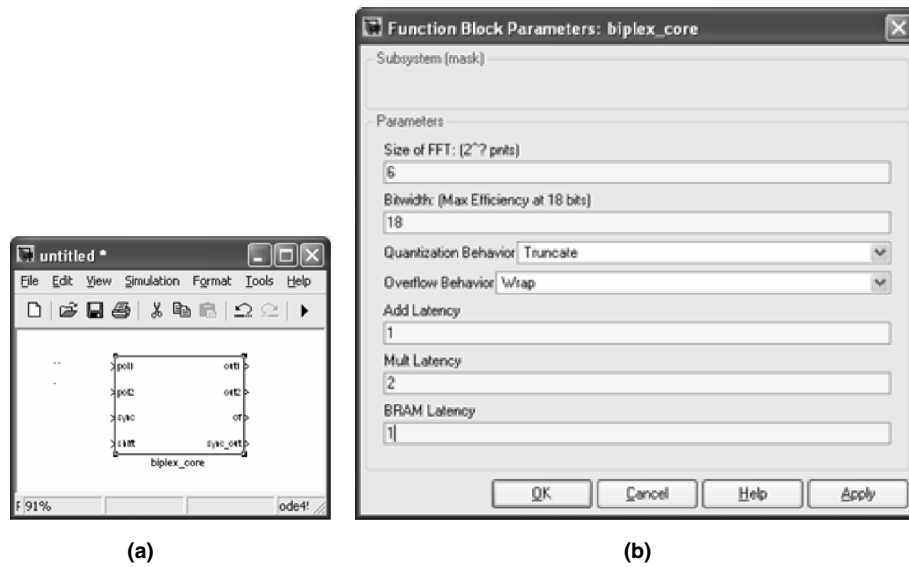


FIGURE 8.12 ■ The `biplex_core` dual-channel FFT block (a), with the parameter dialog box (b).

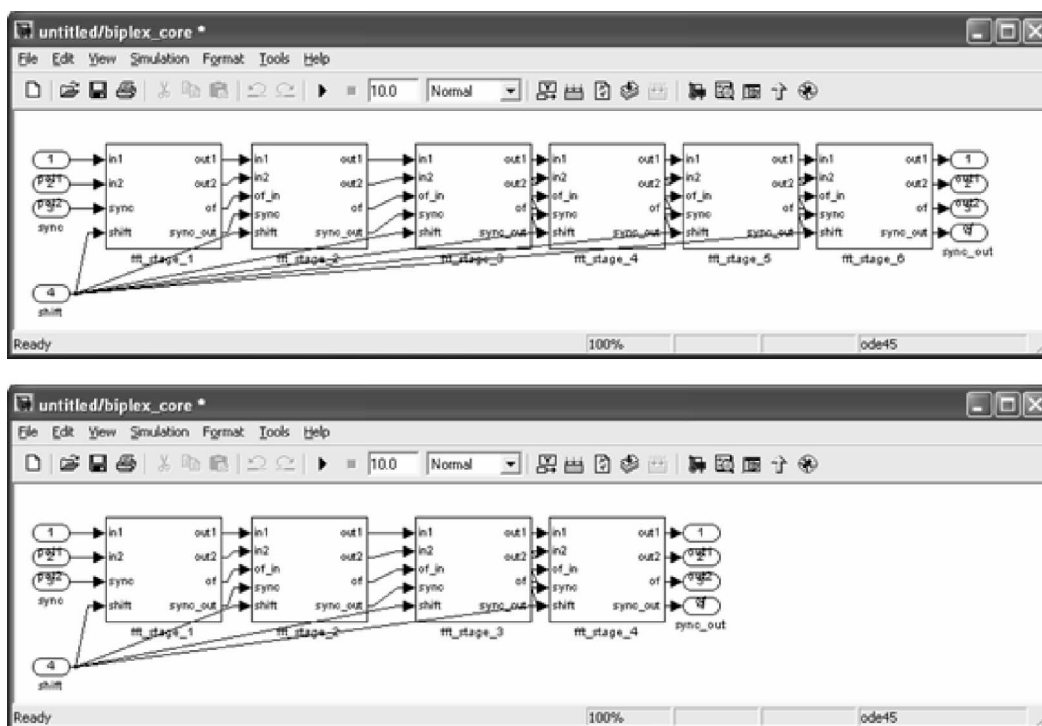


FIGURE 8.13 ■ Two versions of the model schematic for the `biplex_core` library component, with the size of the FFT set to 6 (2^6) and 4 (2^4). The schematic changes dynamically as the parameter is adjusted.

is also available, where the number of I/O ports changes with the FFT size parameter. An 8-input, 8-output version is illustrated in Figure 8.14. The parallel FFT tiles butterfly subsystems in two dimensions and includes parameterized pipeline registers so that the designer can explore speed versus pipeline latency tradeoffs.

In addition to the FFT, other commonly used high-level components include a poly-phase filter bank (PFB), data delay and reordering blocks, adder trees, correlator functions, and FIR filter implementations. Combining these platform-independent subsystems with the BPS I/O and processor interface library described in Section 8.2.3, an algorithm designer can take an active roll in the architectural development of high-performance stream-based signal-processing applications.

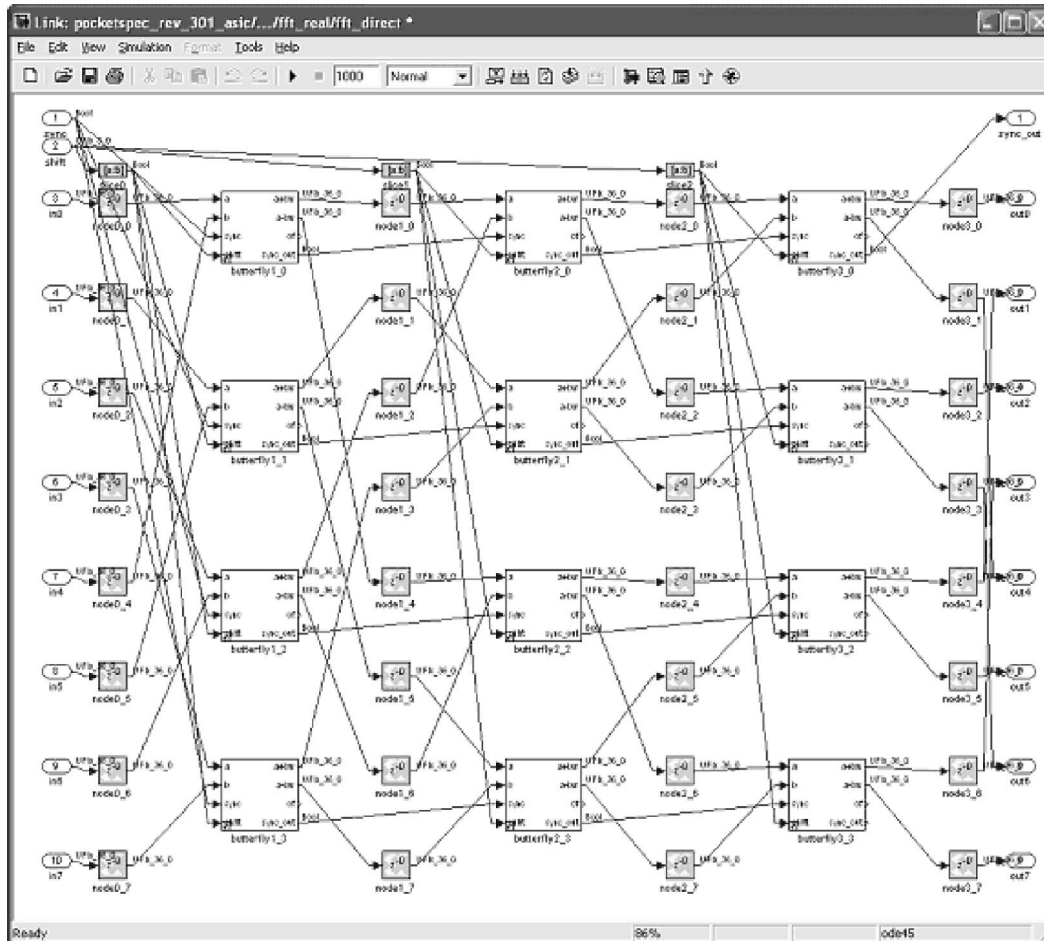
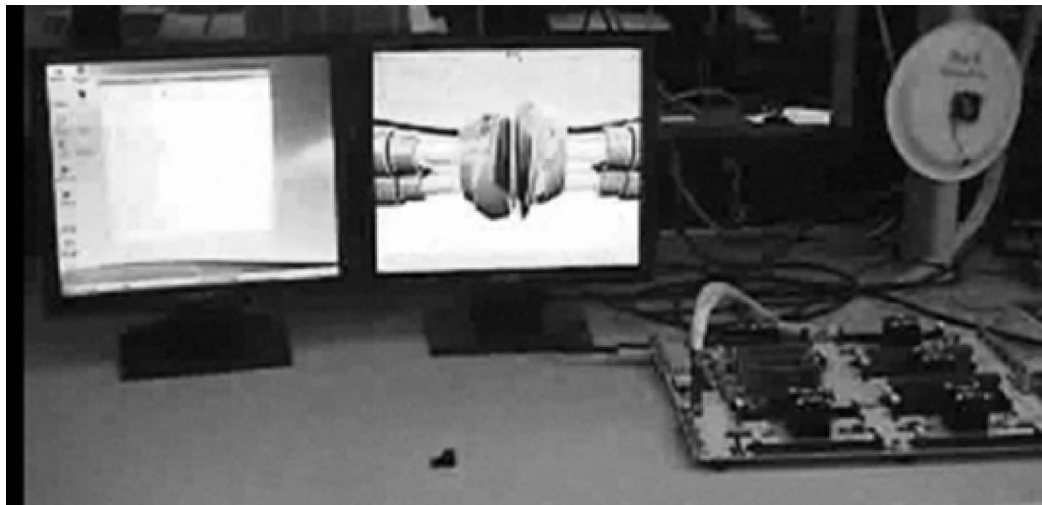


FIGURE 8.14 ■ An automatically generated 8-channel parallel FFT from the `fft_direct` library component.

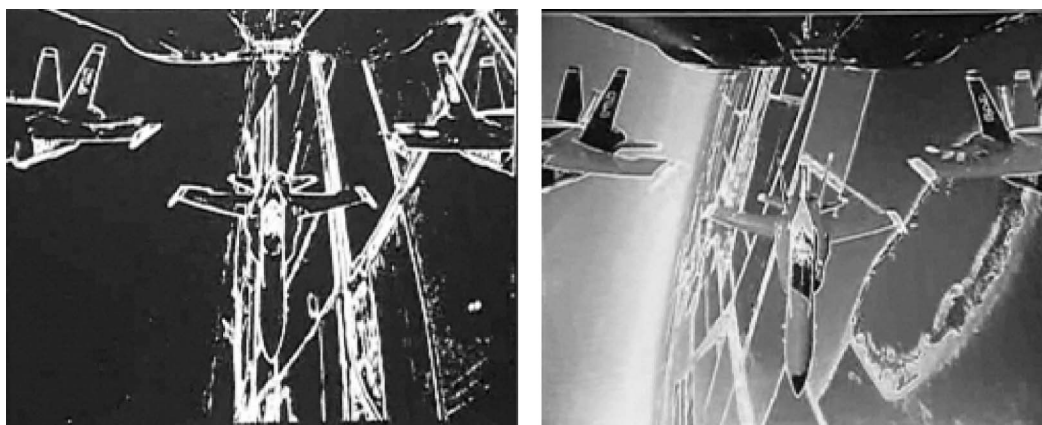
8.5 SUMMARY

This chapter described the use of Simulink as a common design framework for both algorithm and architecture development, with an automated path to program FPGA platforms. This capability, combined with a rich library of high-performance parameterized stream-based DSP components, allows new applications to be developed and tested quickly.

The real-time Sobel video edge detection described in this chapter runs on the BEE2 platform, shown in Figure 8.15, which has a dedicated LCD monitor



(a)



(b)

(c)

FIGURE 8.15 ■ (a) The Sobel edge detection filter running on the BEE2, showing the BEE2 console and video output on two LCD displays, with (b, c) two examples of edge detection results based on interactive user configuration from the console.

connected to it. Two filtered video samples are shown, with edges displayed with and without the original source color video image.

For more information on the BPS and related software, visit <http://bee2.eecs.berkeley.edu>, and for examples of high-performance stream-based library components, see the Casper Project [9].

Acknowledgments This work was funded in part by C2S2, the MARCO Focus Center for Circuit and System Solutions, under MARCO contract 2003-CT-888, and by Berkeley Wireless Research Center (BWRC) member companies (bwrc.eecs.berkeley.edu). The BEE Platform Studio development was done jointly with the Casper group at the Space Sciences Laboratory (ssl.berkeley.edu/casper).

References

- [1] <http://www.mathworks.com>.
- [2] <http://www.xilinx.com>.
- [3] <http://www.synplicity.com>.
- [4] C. Chang, J. Wawrzynek, R. W. Brodersen. BEE2: A high-end reconfigurable computing system. *IEEE Design and Test of Computers* 22(2), March/April 2005.
- [5] C. Chang. Design and Applications of a Reconfigurable Computing System for High Performance Digital Signal Processing. Ph.D. thesis, University of California at Berkeley, 2005.
- [6] <http://www.mentor.com>
- [7] K. Camera, H. K.-H. So, R. W. Brodersen. An integrated debugging environment for reprogrammable hardware systems. *Sixth International Symposium on Automated and Analysis-Driven Debugging*, September, 2005.
- [8] A. Parsont et al. PetaOp/Second FPGA signal processing for SETI and radio astronomy. *Asilomar Conference on Signals, Systems, and Computers*, November 2006.
- [9] <http://seti.eecs.berkeley.edu/casper>.