

NEW CONFIGURATION APPROACHES CAN LEAD TO EASIER SYSTEM DESIGNS, BENEFITING A RANGE OF APPLICATIONS. REDUCING THE OVERALL PIN COUNT, INTERFACE COMPLEXITY, AND RESOURCE USAGE ALSO ENABLES FPGAs TO BE A FLEXIBLE DIGITAL-SIGNAL-PROCESSING ALTERNATIVE TO DSP ARCHITECTURES.

Creating quasistatic, parameterized FPGA designs

Many of today's designs use RAM-based FPGAs (field-programmable gate arrays), either as prototyping vehicles before moving designs to ASICs or as production platforms in low-volume applications. Even in high-volume consumer-electronics applications, you might decide to use FPGAs as the production platform; their all-important in-the-field-reconfigurability enables you to improve product returns by providing new features in the form of potentially revenue-generating updates. FPGAs also enable a shorter time to market, because you can start hardware design before you complete firmware.

FPGAs can also help you reduce hardware diversity and maximize your company's product portfolio. A range of products might differ only in their firmware, minimizing hardware-development costs. To successfully employ FPGAs in consumer-electronics devices, though, you'll need to efficiently use every possible on-chip resource. One way you can make your designs more efficient is by eliminating the control port, a circuit often considered fundamental to FPGAs, but not eliminating its flexibility.

Whatever the application, your FPGAs always need an interface to the outside world, on either a board or a system level. You can split this interface into at least two parts: the actual data ports, which move data into and out of the component, and the control ports. In many designs, the control port is active only immediately after power-up to set system parameters inside the FPGA; it remains idle for the duration of the device's up-time. Although this approach is flexible, it results in inefficient logic use that increases cost.

If your design belongs to this quasistatic category, you may be better off using a more efficient approach that completely eliminates the control port, thereby making gates and I/O pins available for other uses, and still retain the same functions and flexibility of the design you originally had in mind.

Because RAM-based FPGAs require reprogram-

ming each time you switch on their power, you must always use them in combination with a design-specific configuration file containing the information that gives the FPGA its desired functions. Usually, this configuration file remains unchanged for the duration of the product's lifetime. Occasionally, however, new product firmware releases update the configuration file, bringing along new features, fixing annoying bugs, or both. In general, however, the ability to reconfigure the FPGA a virtually infinite number of times is left largely unexploited. You could even consider configuration a nuisance; it increases system cost because the configuration file requires extra storage space, and it takes time. But why not turn this nuisance into a virtue?

Theoretically, you could make a collection of configuration files for a range of settings, store them in nonvolatile memory, and pick the one that best suits your requirements at any given time. In this way, you can forget about the control port. Storing a separate configuration file for all possible control vectors, though, is not an option. Doing so would rapidly result in highly inefficient and therefore unacceptable use of costly and scarce embedded memory. You need to develop a more sophisticated plan to be able to remove the control port from the design.

If you know the structure of the configuration file and the meaning of its contents, you can alter those contents to change the design's behavior and eliminate the control port. You need to store less information, provided that you approach the alteration intelligently. This approach is similar to a firmware update, but the firmware changes every time you modify one or more design parameters. How do you change those parameters, once the control port is gone?

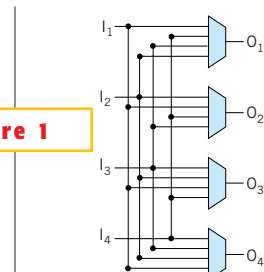


Figure 1

A parameterized barrel shifter is an efficient LUT-based structure.

The answer lies in the small LUTs (look-up tables) that are scattered over the FPGA die. They contain 2^N bits of data and have N inputs—usually, four—to select what bit appears at the output. An interesting aspect of LUTs is that their contents undergo an initialization process upon configuration, using a user-defined or—better yet—engineer-defined value that the configuration file contains. If you can figure out which configuration bits correspond to which LUT, you have all the information you need to start building fully parameterized design elements.

The best way to explain the approach is to start with an example. Certain signal-processing applications require a constant adder or subtractor to, for example, compensate for an offset early in the signal-processing chain. Because that compensation value normally changes slowly with time, your design need not consume logic in the implementation of an always-available control port. Furthermore, getting rid of the control port would result in a smaller and possibly even faster system, with no compromise in flexibility—exactly what you’re after. Indeed, if you store the offset value in a couple of LUTs, you can still change the value by altering the right bits in the configuration file.

The procedure boils down to transforming the new settings in a series of bit positions and bit values and then patching the original configuration file. The only information you have to store is the location of those LUT bits in the configuration stream. One drawback of the scheme is that you need a system processor to do the necessary translation and patching tasks that generate your new configuration file, so using EEPROM-based systems is not an option. However, because most systems include a host processor to do system housekeeping, and you need its processing power only when the system is not fully operational (because you haven’t yet configured your FPGA), the additional cost is minimal.

Constant coefficient multipliers implemented in distributed arithmetic can also benefit from this approach. You

```

LISTING 1—TOP-LEVEL VHDL
entity top is
  Port ( input : in std_logic_vector(3 downto 0);
        output : out std_logic);
end top;

architecture Behavioral of top is
  component RAM16X1S
    port (
      D, WCLK, A0, A1, A2, A3, WE : in std_logic;
      O : out std_logic);
  end component;

  attribute INIT : string;
  attribute INIT of inst_LUT4 : label is "X0001";

begin
  inst_LUT4 : RAM16X1S
    port map (
      O => output,
      A0 => input(0),
      A1 => input(1),
      A2 => input(2),
      A3 => input(3),
      D => '0',
      WCLK => '0',
      WE => '0');
end Behavioral;
    
```

```

LISTING 2—PLACEMENT INFORMATION

s inst_lut4 RAM16 c 1
  pref CLB_R1C1.S1
  place CLB_R1C1.S1.G
  p d I 24
  p wclk I 24
  p a0 I 19
  p a1 I 18
  p a2 I 20
  p a3 I 17
  p we I 24
  p o O 16
    
```

would normally be unable to change the coefficient value without going back to the software-design environment, because a distributed arithmetic multiplier uses LUTs. However, once you know the relationship between the LUTs and the configuration bits, the constant suddenly becomes a variable using the approach described earlier. You can then use this multiplier in a range of applications, such as FIR and IIR filters, with the same flexibility in frequency characteristics (references 1 through 3).

A parameterized barrel shifter is an-

other efficient LUT-based structure. By using 4-bit LUTs as multiplexers, you can implement a 4-bit-wide shifter (Figure 1). The number of LUTs you need depends on the shifter’s size, word width, and shift range. Such an approach could be useful, for instance, as a programmable-gain block. Again, merely changing the LUTs’ coefficients results in a different shift distance.

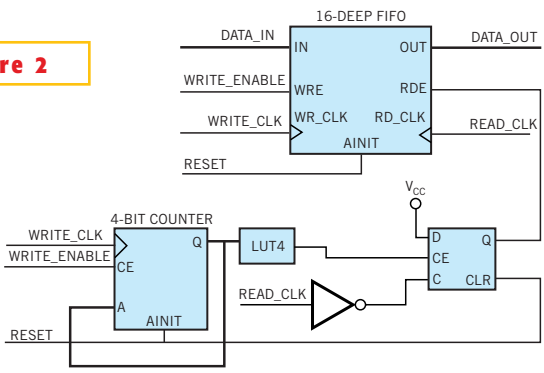
With a variable-length-delay line, a small extra block of logic can provide you with the necessary flexibility to change the delay, and it fits perfectly into the LUT-bit-patching scheme (Figure 2).

As soon as the FIFO starts filling, the counter begins incrementing. When the counter reaches a certain value (for which the bit in the LUT is set to 1), the flip-flop switches and remains in the “1” position, thereby enabling the read port of the FIFO. From that point on, the FIFO remains filled at this level, and you have achieved the desired delay. If you increase the counter width and the number of LUTs in the design, you can set the maximum delay to any value, provided that the FIFO is large enough. Two LUTs and an AND gate are sufficient to support a 255-cycle delay.

By now, it should be clear that the LUT-based approach can be useful. You are still one step away from a workable scheme, though, until you determine the relationship between the LUT contents and the corresponding configuration file. Fortunately, this task is relatively easy. First, you need to find out where the LUTs will reside inside the FPGA. The design environment you are using normally outputs that information, generated during the place-and-route step in the design process. Then, you need to determine which LUT location corresponds to which bits in the configuration file. Ideally, the FPGA vendor provides this information in its product documentation. At least one FPGA vendor, Xilinx, has published everything you need to complete the described procedure.

When such details are unavailable, it’s easy to extract the LUT-to-bit relationship by analyzing a couple of wisely chosen

Figure 2

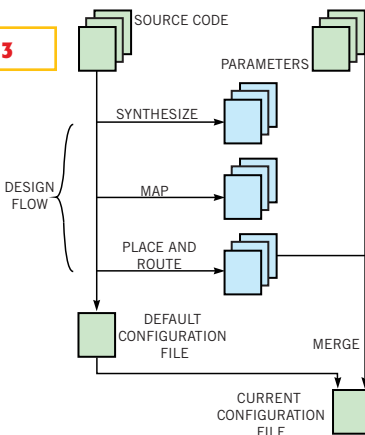


Increasing the counter width and the number of LUTs enables you to set the maximum delay of the variable-delay line to any value.

designs. To be on the safe side, though, be sure to first ask the vendor's permission. Finally, compile the placement and bit-location info into one file, which serves as the blueprint of your design. You now have two files. The first is the original configuration file, and it contains some default settings. The second file contains a list of records of LUT instance names that your design uses, along with their bit positions. From now on, whenever you want to change some or all of the design parameters, a merging tool will combine the new parameters with the two design files into a new configuration file that you can subsequently use to program the FPGA (Figure 3).

The Xilinx ISE 5SP3 environment, with a Virtex XCV50-5PQ240 as the design target, illustrates these steps. The design comprises just one instantiation of the entity RAM16X1S, called inst_LUT4, with its input and output pins routed to five package pins (Listing 1). The entity RAM16X1S, is a primitive, mapped onto one LUT. The end goal is to modify the

Figure 3



A merging tool combines the new parameters and the two design files to form a new configuration file.

contents of this RAM module by changing the appropriate bits in the configuration file. Simple as the design may be, it demonstrates all of the necessary steps of the process.

As indicated, the knowledge you need to get going is the LUT placement. In the

ISE design environment, the Floorplanner provides this information. In this example, it generates a file named top.fnf containing a list of records that describe the primitives used, their location on the die, and their interconnection. From one record, you can see that the instantiated RAM module is using the LUT “G” located in row 1, column 1, slice 1 (Listing 2). The design includes a placement constraint; hence, the “pref” line.

A number of formulas provide the relationship between the LUTs' contents and their positions in the configuration file (references 4 through 6). These formulas take device- and design-specific values, such as FPGA type, LUT location, and bit index, as an input, and generate bit positions as an output. In this example, bit 3 in byte 54,098 of the first configuration frame of the configuration file sets bit 0 of the inst_LUT4 (Table 1).

At this point, you can change the contents of the LUT without ever again using the design environment; you just change the listed bits. Don't forget that you must also recalculate the CRC values

embedded in the configuration file. Because the configuration file has changed, the configuration will fail unless you also update the CRC. This job is straightforward, because Xilinx specifies in its public documents the CRC algorithm that it uses to guarantee file integrity. After this last step, the new configuration file is ready for use.

The novelty of this approach lies in its use of the small LUTs as patchable storage containers, or static-control ports, whose contents you can modify without returning to the design suite. Each time you want to change the parameters, you just set or clear the appropriate bits in the configuration file and generate a new configuration file on the fly. As such, the use of the FPGA's reconfigurability feature extends beyond the well-known concept of firmware updates. The approach imposes no restrictions on the type of structures you want to design. You can build up every parameterized design using LUTs, although some structures will use resources more efficiently than others. Even for less efficient structures, low efficiency

may be a small price to pay, considering the design's inherent flexibility.

The technique is applicable to any FPGA family that uses LUTs as its basic building blocks; it is not limited to one FPGA vendor's product range. The firmware engineer needs to know only how the LUT data relates to the contents of the configuration file. However, bear

in mind that, if you change the wrong bits in the configuration file and correspondingly update the CRC values, you create a syntax-valid file that might destroy parts of the component, for example, due to an internal short circuit. (The vendors' design software employs self-checks to guard against such possibilities.) □

TABLE 1—BIT-POSITION EXAMPLE

Bit	Byte offset	Bit offset within byte
0	54,098	3
1	54,050	3
2	54,002	3
3	53,954	3
4	53,906	3
5	53,858	3
6	53,810	3
7	53,762	3
8	53,714	3
9	53,666	3
10	53,618	3
11	53,570	3
12	53,522	3
13	53,474	3
14	53,426	3
15	53,378	3

AUTHORS' BIOGRAPHIES

Jo Pletinckx is a research assistant of the Fund for Scientific Research—Flanders (Belgium)(FWO—Vlaanderen), preparing for a PhD in electrical engineering focusing on hard-real-time systems.

Rik Vlaminck is a research engineer also preparing for a PhD in electrical engineering, focusing on the emulation of copper-based access network plants.

Jan Vandewege, professor and head of the INTEC_design training laboratory obtained his PhD in electronics from Ghent University in 1978.

You can find the references to this article on the Web version at www.edn.com.