FPGA lookup tables build flexible pattern matchers

Steve Knapp

Peer into the heart of most SRAM-based field-programmable gate arrays (FPGAs), and you'll find a common logic function called a lookup table (LUT). This little bundle of Boolean brilliance is the basic building block responsible for the combinatorial logic in an FPGA design. FPGA development software gathers the gates you draw with a schematic editor or that a logic synthesizer creates from your description, and it packs the resulting function into LUTs. However, the development software sometimes hides the incredible power and flexibility inherent in a LUT, which is capable of much more than just implementing a collection of gates. This article investigates a few possibilities.

Building blocks

First, who makes FPGAs using LUTs? Table 1 shows a list of popular device families, their manufacturers and the number of LUT inputs. Generically, the industry considers these devices coarse-grained FPGA architectures, distinct from fine-grained architectures that employ a much smaller logic function. Some of the listed architectures support only 4-input LUTs. Others combine two 4-input devices with a 2:1 multiplexer in each logic block to make a 5-input LUT. Some devices even support LUTs with as many as six inputs.

FPGA Family	Vendor	3-input LUT	4-input LUT	5-input LUT	6-input LUT
Flex 10K	Altera		•		
AT40K	Atmel	•	•		
Orca	Lucent		•	•	•
VF1	Vantis	•	•	•	•
XC4000	Xilinx		•	•	

Table 1—Various FPGA companies build devices using lookup tables (LUTs) to implement logic.

Second, and perhaps more fundamental, what exactly is a LUT? Consider the example of a 4-input device, which is essentially a 16-bit deep 1-bit wide PROM (Fig 1). Although built with SRAM technology, external circuitry programs it at power-up and usually leaves it alone, so think of it as PROM. The four logic inputs act as address lines to the LUT's PROM, whose contents define the

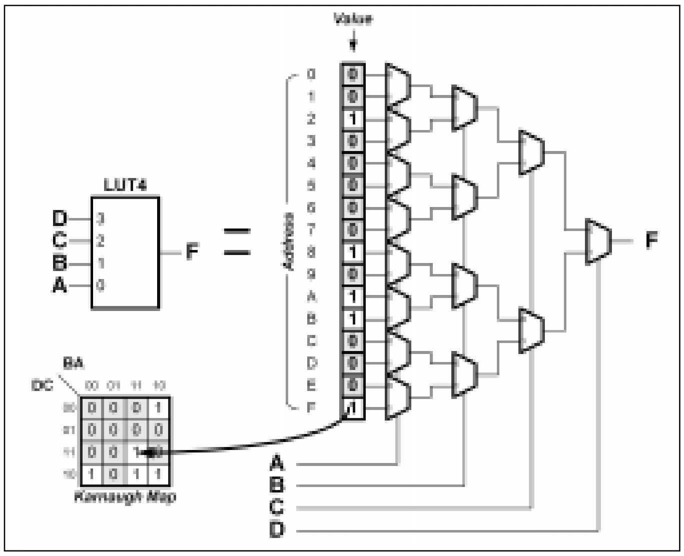

Fig 1—A 4-input lookup table (LUT4) is essentially a 16x1 PROM whose address inputs select the logic function held in the memory's data contents.

Steven K Knapp is the founder and president of OptiMagic Inc (Aptos, CA, www.optimagic.com), a firm that develops intellectual property and design software for programmable logic. Prior to founding this firm he held various applications, engineering and management positions at Xilinx and Intel's former programmable-logic division.

logic function and require 16 memory locations. These locations represent 65,536 possible functions ranging from simple state definitions such as a logic One or Zero to something more complex such as a 4-input XOR. Simply stated, if a function has four or fewer inputs, it fits into a 4-input LUT regardless of its Boolean complexity, assuming no feedback. Fig 2 shows such an example.

Another way to think of a 4-input LUT is as a 4-input Karnaugh map. Each of the device's memory locations maps to one of the Karnaugh map's cells. For designers more familiar with PAL-like devices, think of this same LUT as a marcrocell with 16 product terms, each with four inputs and their complements (Fig 3).

Table 2 shows that a LUT's complexity grows exponentially with the number of inputs. For example, while a 4-input LUT supports 65,536 possible functions, a 5-input supports more than 4 billion possible functions!

If a wider LUT is more flexible and thus more powerful, why not use an even wider one on a device? Wouldn't 8-input LUTs be even better and faster than a 4-input version for implementing logic? Sure, it would be faster for wider functions and support a great many combinations. However, the memory array a LUT requires also grows exponentially with the number of inputs. Growing one from four inputs to five doubles the number of memory bits it needs, which roughly doubles the silicon area. In addition, not every function could optimally use such a large block, resulting in wasted silicon area. Based on both theoretical and empirical data, LUTs with between three to five inputs seem ideal for most applications.

Note that some devices, such as Altera's Flex 10K FPGAs, contain bigger SRAM memory functions, known as embedded array blocks (EABs). When functioning as a lookup table, an EAB performs as a single 10-input LUT or as up to eight different 8-input LUTs with shared inputs.

Another advantage of LUTs is speed. Conceptually, LUTs exhibit a constant delay regardless of the function. Though some subtle differences arise in delay between the LUT inputs, vendors model the delay from all inputs to the output with the same value. Consequently, an inverter, a 4-input XOR or other complex 4-input functions all incur the same logic delay.

	Generically	For 4-input LUT	For 5-input LUT
LUT inputs	n	4	5
PROM bits required	2^n	16	32
Possible functions	2^{2^n}	2^{16} = 65,536	2^{32} = 4,294,967,296

Table 2—A lookup table's complexity grows exponentially with the number of inputs.

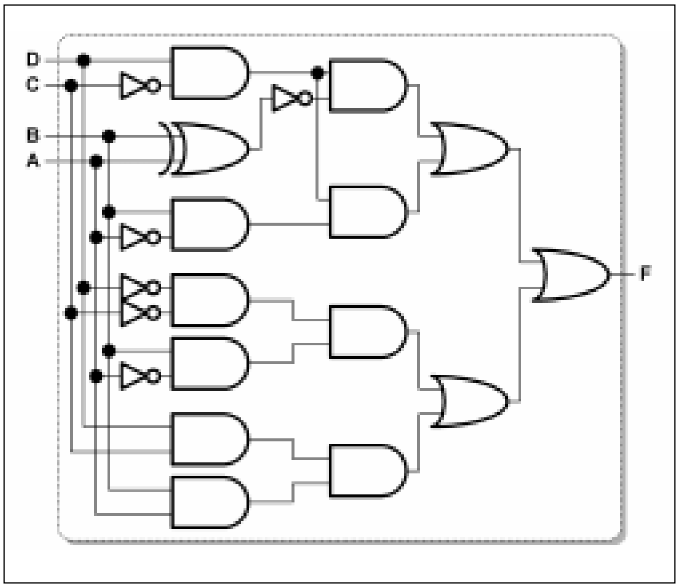

Fig 2—Regardless of logical complexity, a 4-input lookup table can always represent a function with four inputs, provided the function contains no feedback.

Development support

Direct development system support for LUTs varies widely between vendors and sometimes even between device families from one vendor. Every development tool can map a function described in schematics or through synthesis and place it in a LUT. Some, though, allow you to specify LUT contents directly. Consider that Xilinx offers schematic primitives that define a 16x1 PROM (ROM16X1) or a 32x1 PROM (ROM32X1). Recall that a 4-input LUT behaves exactly like a 16x1 PROM. In a schematic environment, you define LUT contents by attaching an attribute to the appropriate ROM symbol. For instance, to define the function in Fig 2 using a PROM symbol, you place a ROM16X1 symbol on the schematic, connect the appropriate inputs and add an attribute to define the PROM's data contents. The Xilinx software expects the initialization string in hexadecimal format, from most-significant bit down to least, and in this example it would be INIT=8D04. In contrast, Lucent doesn't seem to offer a ROM primitive at first glance. However, you can use one of the on-chip RAM primitives and turn it into ROM by disabling the RAM's write-enable input.

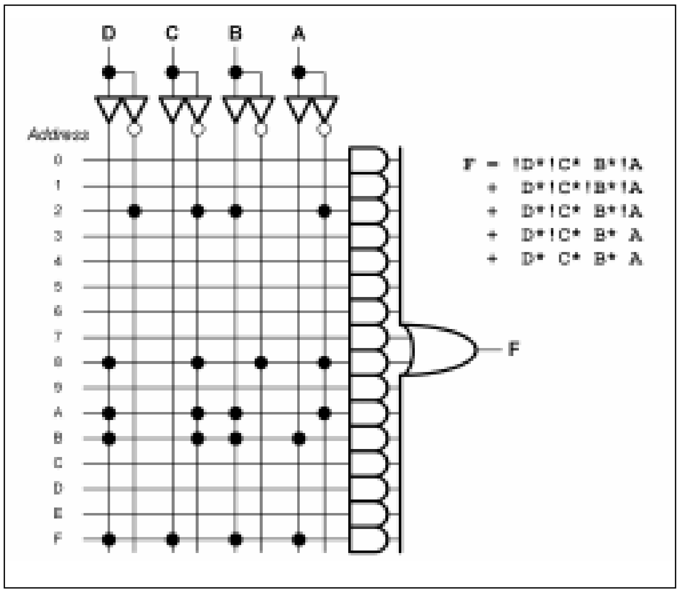

Fig 3—A 4-input lookup table, when presented as a PAL macrocell, appears as 16 product terms, each with four inputs and their complements.

Pattern-matching example

Let's look at a simple example that demonstrates the full power of a LUT. In some digital pattern matching applications you want to find an exact match to a specific string. In other cases, you're interested in also finding a close match. For instance, assume that a circuit must look for the binary pattern 1010. Further, the incoming data might be corrupted or you might want to check for similar patterns. So instead of an exact comparison, check if at least three out of the four bits match.

In other words, you want to perform the function in Fig 4 and Table 3. The process then involves comparing each bit against a corresponding bit in the desired fixed pattern, counting the number of bits that match the pattern and checking if the number of match bits reach the threshold value (three).

Pattern (D, C, B, A)	Number of matching bits	3 or more bits match?	LUT Address	LUT Value
0000	2	No	0	0
0001	1	No	1	0
0010	3	Yes	2	1
0011	2	No	3	0
0100	1	No	4	0
0101	0	No	5	0
0110	2	No	6	0
0111	1	No	7	0
1000	3	Yes	8	1
1001	2	No	9	0
1010	4	Yes	A	1
1011	3	Yes	B	1
1100	2	No	C	0
1101	1	No	D	0
1110	3	Yes	E	1
1111	2	No	F	0

Table 3—Evaluating all 16 possible input values produces the resulting data to implement the digital pattern matcher in a single 4-input LUT.

Because you're matching against a fixed pattern and because the threshold value is fixed, the pattern matcher becomes a function of the four logic inputs. If only four inputs are available, then this entire complex function must fit into a 4-input LUT. If you wish to implement it in a Xilinx XC4000, the easiest way to specify it would be with a ROM16X1 symbol with the initialization property set to INIT=8D04. Alternatively, you could describe the function with the schematic in Fig 2 or enter it through synthesis using the data in Table 3 or the Karnaugh map in Fig 1, all of which show equivalent implementations.

This example demonstrates how to match in input stream to a fixed pattern. What if the target values were variable? A number of possible implementations consume varying numbers of gates, with larger designs allowing faster switches between patterns.

Consuming the largest amount of real estate but permitting the fastest pattern-change time is a pattern

Intentionally blank

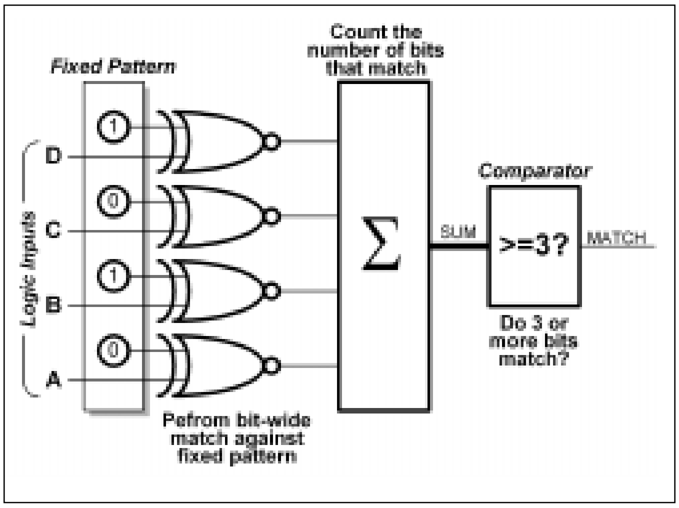

Fig 4—A 4-bit digital pattern matcher fits into one LUT.

matcher with additional inputs for a variable pattern and a variable threshold. With it you can change which pattern the logic looks for very quickly but at the cost of substantially increased logic size.

Another possible solution involves FPGAs with LUTs that can function as RAM. With some additional control logic, you can occasionally update the LUT contents to match a new pattern. While pattern changes aren't as fast as with the previous circuit, this design consumes much less logic.

Or as in Fig 4, you could build multiple circuits with various fixed patterns and thresholds. Whenever an application requires a new set of values, you reprogram the entire FPGA with another design that contains the modified match values. This approach is useful in designs where variable values change very infrequently, but it has the disadvantage that the updating process requires that the FPGA shut down during reprogramming. ***PE&IN***

Editorial Feedback
This article's value to me was:
High—269 Average—270 Low—271