FPGA Implementation Details

Clark N. Taylor
Department of Electrical and Computer Engineering
Brigham Young University
clark.n.taylor@gmail.com

October 22, 2008

1 Introduction

So far, in both ECEn 224 and this course, we have always talked about implementing digital logic in a fairly generic technology, or made broad assumptions about the characteristics of the technology. In these notes, I will discuss one particular technology, Xilinx FPGAs, and we will discuss some of the unique aspects of implementing logic with this technology.

These notes are organized as follows. In Section 2, we discuss how basic boolean formulas are implemented in Xilinx FPGAs. Despite being able to implement any generic logic function using the basic FPGA architecture, Xilinx has made some modifications to make specific classes of logic functions easier to implement in the FPGA fabric. They have made modifications at both a "micro" and "macro" level, which will be discussed in Sections 3 and 4 respectively.

2 Implementing Simple Boolean Formulas in an FPGA

To understand how a Xilinx FPGA implements general logic functions, it is necessary to review the first introduction to boolean functions that was presented in ECEn 224. In ECEn 224, boolean functions were first represented as a truth table. For example, a 3-input `xor` boolean function (F) with inputs A, B, and

C could be represented as

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

. (1)

To represent different boolean functions of the three inputs A, B, and C, the F column is simply modified.

The key insight used to create Xilinx FPGAs is that the truth table described above could also be an initialization table for a memory. For example, if an 8-element memory was available, then there would be three address lines used to address the memory, and for each address a single bit would be stored. Therefore, by tying A, B, and C to the address lines of an 8-element memory and setting the elements of the memory to the F column, we can implement any 3-input boolean function.

The basic building block of Xilinx FPGAs is a LUT, which stands for "lookup table". Each LUT is simply a 16-element memory, which enables any 4-input boolean function to be implemented. For example, to implement a 4-input `or` gate, the memory would be loaded with the 16-bit value 0xFFFE (writing the F bits from most significant to least significant). For an `and` gate, the value 0x8000 would be loaded into the 16-element memory. By setting the bits of the memory, *any* 4-input (or less than 4) boolean function can be implemented.

The second primary building block of Xilinx FPGAs are routing blocks which connect together different components. These routing components are implemented in one of two ways. First, a mux can be used to decide between two inputs, with the select line of the mux tied to an SRAM cell. An example of this type of routing is shown in Figure 1. Associated with every LUT in an FPGA is a register. Usually, the register is used to store the output of the LUT, but it can also be used to store some other value routed into that register. The *configuration bit* (a single bit stored in on-chip memory) is "programmed" into the FPGA to determine which value the register is required to store.

The second way that routing is configured is by "connecting" wires together. While the specific connections allowed between wires can vary significantly from FPGA to FPGA family, the general connection architecture is a "crossbar." An example of a crossbar is shown in Figure 2. The key technique used to create a cross bar is the transistor that is placed between two wires which are crossing. If the gate on the transistor is held low, then there is no connection between the wires. However, when the gate is high, then the two wires are connected. By programming crossbar switches with the appropriate configuration bits, connections between wires are made. In this way, connections between the inputs and outputs of different LUTs can be programmed, enabling the FPGA

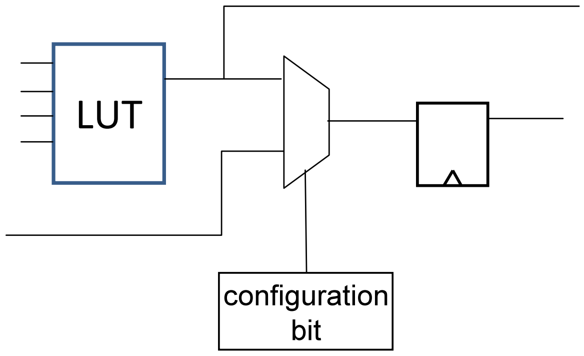

Figure 1: An example of using a mux to route signals in an FPGA. Note that the value stored in the configuration bit what the input to the register will be.

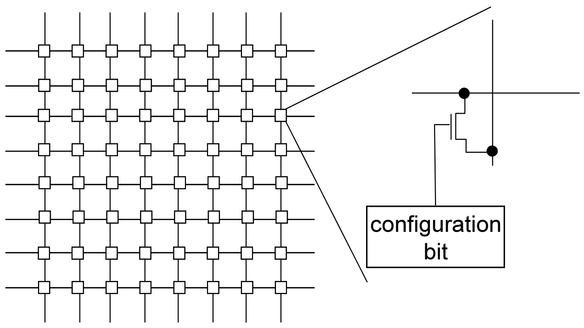

Figure 2: A simple example of a crossbar. The individual connections between wires can be enabled or disabled by appropriately setting the configuration bits.

to implement complex logic functions.

With the basic elements of configurable LUTs and configurable routing now enabled, any generic boolean function can be implemented. In addition, with the ability to register outputs as shown in Figure 1, state machines and other sequential logic can be enabled. Note that for the rest of these notes, we will concentrate on the “basic unit” of the FPGA, shown in Figure 1. To create an FPGA, this unit is replicated multiple times, enabling a large variety of boolean functions to be implemented on the FPGA.

2.1 Impact of Basic FPGA Design

Now let us briefly consider the impact of how the FPGA is designed on simple boolean formula implementations.

First, in the past, we have claimed that different types of boolean functions will have different timing requirements. Because of the LUT-based structure of

FPGAs however, all functions of 4 or less inputs take the same time. Therefore, it will not matter if we are implementing a 4-input XOR, a 4-input OR, a 4-input NOR, or some other random function (e.g., AB'C + A'BCD + ABCD') of 4 inputs. All these functions will take the same time to compute.

Second, it is important to note that the critical path when timing a design in FPGAs is often the routing and not the logic itself. Because the routing has to be configured, every connection between wires in a crossbar leads to another transistor delay. The more connections on a path, the longer it takes to propagate through the path. Similarly, in the actual FPGA hardware design, a significant amount of area (close to 90%) is dedicated to routing signals.

Third, in previous classes we have claimed that, as a general rule, the fewer the inputs to a gate, the faster that gate runs. This is not strictly true on FPGAs, however. Because the basic unit of implementation is a 4-input LUT, any function of up to 4 inputs takes the same amount of time to compute. Therefore, the timing difference between a 2-input boolean function and 4-input boolean function is zero. Once more than 4 inputs are required however, then the amount of hardware can increase dramatically.

To implement a boolean function with more than 4 inputs, factoring out of individual boolean terms is used. For example, for a 5-input function of A, B, C, D, and E, we can rewrite the function as

$$f(A,B,C,D,E) = A \cdot g(B,C,D,E) + A' \cdot h(B,C,D,E), \tag{2}$$

where $g()$ and $h()$ are functions which are derived from factoring out A and A' from $f()$. Therefore, when using FPGAs to implement functions larger than 4 inputs, a set of functions with 4 inputs are derived and placed in LUTs. Another set of LUTs is then used to mux between the outputs of these LUTs, using the remaining inputs as the select lines to that mux. Using this method for implementing logic functions can lead to a rapid increase in hardware required to implement the function. For example, to implement a generic boolean function in an FPGA, the following number of LUTs are required for a given number of inputs:

# inputs	# LUTs
4	1
5	3
6	7
7	15
8	31
9	63
10	127

(3)

As can be observed from the example above, as the number of inputs to a function increases, the number of LUTs increases exponentially. Therefore, while FPGAs *can* implement any boolean function, there are certain types of functions that the basic FPGA fabric does not support very well. Therefore, Xilinx has introduced several modifications to their basic FPGA fabric that enables certain types of functions to be implemented more efficiently. We will discuss these modifications in the following sections. In Section 3, we discuss modifications that lie close to the LUTs themselves, termed "micro" modifications. In Section 4, we discuss large units that are often added to the FPGA to enable specific algorithms to run more efficiently.

3 Micro-modifications to FPGA Fabric

In this section, we discuss two modifications to the basic LUT-based FPGA fabric discussed in the previous section that helps enable certain types of digital logic to be implemented more efficiently. Because both of these modifications are made to the basic LUT units, we call these modifications "micro-modifications." The two modifications we discuss here are used to implement boolean functions with a large number of inputs and addition/subtraction operations.

3.1 Implementing multi-input boolean formulas more efficiently

As shown in Section 2.1 and Equation (3), one difficulty with using FPGAs is that the number of LUTs increases exponentially with the number of inputs to a logic function. Xilinx realized this problem and addressed it by placing two LUT / flip-flop combinations (shown in Figure 1) together in a single *slice*. Besides just placing these two LUTs together, they also added a "F5MUX" to each slice which, simply put, takes the place of the third LUT that would normally be required to implement a 5-input logic function. By replacing a LUT with a single 2x1 MUX, they are able to significantly reduce the routing overhead and LUT requirements of implementing a 5-input function.

To enable even larger functions, they have also grouped four slices into a "combinational logic block" (*CLB*) and added another mux to each slice in the CLB. Because there are four slices in a CLB, and each slice has one extra 2x1 mux, each CLB can implement either 2, 6-input functions or 1 7-input function. Also, if an 8-input function is desired, there are enough muxes that 2 CLBs can be used together to implement one 8-input function. Note that with these modifications, a generic 8-input boolean function can be implemented using 16 LUTs (2 CLBs), a 2x improvement over the 31 required as shown in Equation (3). The routing for these muxes is also designed to avoid crossbars, requiring far less timing to implement 5 or more input functions than if the basic LUT structure alone was used.

3.2 Enabling efficient additions/subtractions

The other type of operation that Xilinx decided to explicitly support in their FPGA fabric is addition and subtraction. Without modifying the FPGA fabric, there are two major problems with implementing addition/subtraction on the FPGA, namely:

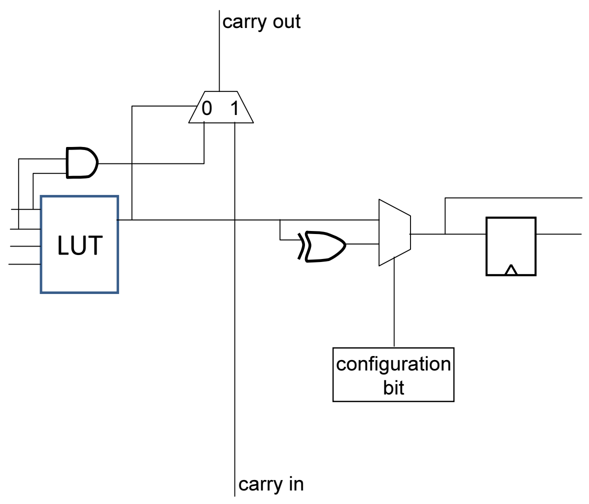

Figure 3: The basic computation unit of the FPGA modified to enable fast carry chain logic.

1. Each full adder block in an adder/subtracter must generate both the sum and a carry out bit. Therefore, two LUTs are required per bit in an adder/subtracter, a rather high cost for such a basic operation.
2. If a carry out bit is generated in the typical fashion by a LUT and is used by another LUT, this carry bit has to go through the routing fabric of the FPGA. Therefore, when implementing an adder/subtracter, the carry bits will have to go through the routing fabric multiple times, dramatically increasing the timing required to perform the addition/subtraction.

To solve this problem, Xilinx has implemented specific carry chain logic in their FPGAs to help with addition and subtraction. A basic picture of the carry chain logic is shown in Figure 3. To implement carry chain logic in an efficient manner, several portions have been added to the basic block shown in Figure 1. In Figure 3, note that the carry in bit comes in at the bottom. The final `xor` used to determine the sum has been removed from the LUT and added as custom logic, enabling more rapid execution once the carry in arrives. In addition, it allows the output of the LUT to be, essentially, a "propagate" signal similar to that used by the carry-lookahead adder. An `and` gate is the "generate" signal, which is passed through to the next adder if and only if the propagate signal is 0. If the propagate signal is one, then the carry in is passed on to the carry out. This approach overcomes the two problems with addition described above because (1) only one LUT is used per bit of addition and (2) the carry in/out is always sent to the element above and does not run through a routing block, enabling fast carrys in the FPGA.

While these modifications to the basic building blocks of the FPGA have proven to be very useful, more recent versions of Xilinx FPGAs have also added specific hardware units to help speed up specific applications on the FPGA that can be routed to, but are not implemented using LUTs. Some of the basic blocks that have been added to FPGAs are discussed in the next section.

4 Macro-modifications to FPGA Fabric

While there are several different macro-modifications that are possible and have been included in some Xilinx FPGA, we cover just three in this section. The first two are included in the Spartan III FPGAs used in the ECEn 320 lab, while the third one is not included in the FPGAs for our class, but is something which I think you should be exposed to. These three modifications are (1) large memories, (2) multipliers, and (3) microprocessors.

4.1 Large memories in the FPGA fabric

While the basic layout of an FPGA is to have multiple small (16-element) memories, most applications that require significant computing require some kind of memory access on most clock cycles. While the memory can all be placed off-chip, there are timing, power, and throughput advantages to having multiple medium-sized memories on-board the FPGA. Xilinx has made this option available through the use of *Block RAMs*.

Block RAMs are simply a configurable 1024x18 bit memory. The memory itself is both read and write dual-ported. This enables several different configurations of the memory. First, any number of bits (up to 18) can be used as the data width of the block RAMs. Note that this includes having two 1024x9 bit single-ported memories in the same block RAM. Second, because the memory is dual ported, we can have any combination of two ports available, making the block RAMS effective for FIFOs, stacks, etc.

4.2 On-chip multipliers

Early Xilinx FPGAs were essentially designed to be nothing but glue logic (i.e., a few and/or gates between other components.) However, as FPGAs became larger in size, it quickly became apparent that FPGAs could be used for serious computation as well. (As an aside, BYU research was quite involved in helping this transition in how FPGAs were viewed in the community. The FPGA lab on-campus implemented several high-performance computing algorithms on FPGAs, proving it was possible.) One of the most fundamental operations in most compute-intensive algorithms (i.e., DSP applications) is a multiply. While a multiply operation can be efficiently implemented on FPGAs using a shift-and-add architecture, the timing, area, and power are dramatically reduced in an ASIC implementation. Therefore, Xilinx places several 18x18 multipliers (producing 36 bit outputs) on-chip, enabling fast, low-latency multiplication to occur.

4.3 On-chip microprocessors

In Xilinx's "Vertex Pro" FPGAs, they have placed an entire microprocessor on-board as an ASIC design. The on-chip processor is a PowerPC microprocessor, running at speeds of up to 400 MHz. The processor communicates with the rest of the FPGA using IBM CoreConnect-based protocols, enabling any number of units to be placed on the FPGA or communicated to off-chip, utilizing the FPGA fabric as an intermediary. This concept of placing a powerful microprocessor on-chip close to hardware enables unique and powerful hardware/software designs on a single FPGA chip.

5 Conclusion

In these course notes, we have introduced the basic technology used to implement generic boolean functions on-board the FPGA. We have also talked about several different ways in which Xilinx has improved on this basic technology to enable more advanced applications to be implemented on the FPGA with minimal cost.