# FPGA Fundamentals

Publish Date: May 03, 2012

## Overview

Field-programmable gate arrays (FPGAs) are reprogrammable silicon chips. Ross Freeman, the cofounder of Xilinx, invented the first FPGA in 1985. FPGA chip adoption across all industries is driven by the fact that FPGAs combine the best parts of application-specific integrated circuits (ASICs) and processor-based systems. FPGAs provide hardware-timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design.

Reprogrammable silicon also has the same flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can function autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing.

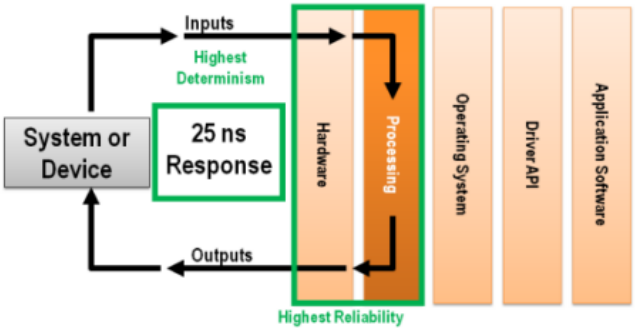


Figure 1.One of the benefits of FPGAs over processor-based systems is that the application logic is implemented in hardware circuits rather than executing on top of an OS, drivers, and application software.

If you are new to the concept of an FPGA, the goal of this paper is to introduce you--the nondigital hardware designer--to the building blocks of an FPGA and the design tools that make it possible to have a reconfigurable silicon chip.

## Table of Contents

### Defining the Parts of an FPGA

Every FPGA chip is made up of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit and I/O blocks to allow the circuit to access the outside world.

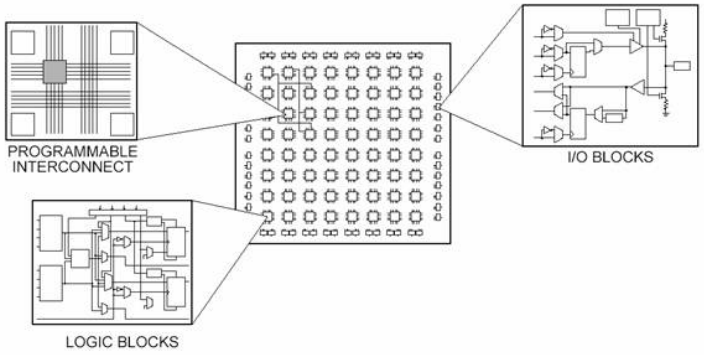


Figure 2. The Different Parts of an FPGA

FPGA resource specifications often include the number of configurable logic blocks, number of fixed function logic blocks such as multipliers, and size of memory resources like embedded block RAM. Of the many FPGA chip parts, these are typically the most important when selecting and comparing FPGAs for a particular application.

The configurable logic blocks (CLBs) are the basic logic unit of an FPGA. Sometimes referred to as slices or logic cells, CLBs are made up of two basic components: flip-flops and lookup tables (LUTs). Various FPGA families differ in the way flip-flops and LUTs are packaged together, so it is important to understand flip-flops and LUTs.

Expand the sections below to learn more about each component.

+ **Learn More About Flip-Flops**

+ **Learn More About LUTs**

+ **Learn More About Multipliers and DSP Slices**

+ **Learn More About Block RAM**

### Designing FPGAs Into a System

With this understanding of the fundamental FPGA components, you can clearly see the advantage of implementing your logic in hardware circuitry: you can realize improvements in execution speed, reliability, and flexibility. However, you face trade-offs using only an FPGA for the processing and I/O connectivity in your system. FPGAs do not have the driver ecosystem and code/IP base that microprocessor architectures and OSs do. In addition, microprocessors coupled with OSs provide the foundation for file structures and communication to peripherals used for many, often essential, tasks such as logging data to disk.

As a result, over the last decade a hybrid architecture, sometimes called a heterogeneous architecture, has emerged in which a microprocessor is paired with an FPGA that is then connected to I/O. This approach takes advantage of the benefits that both these targets offer. In recent years, companies such as Xilinx, with its Zynq family of targets, have adopted this approach and released solutions that combine both the processor and FPGA onto a single chip to create this hybrid architecture.

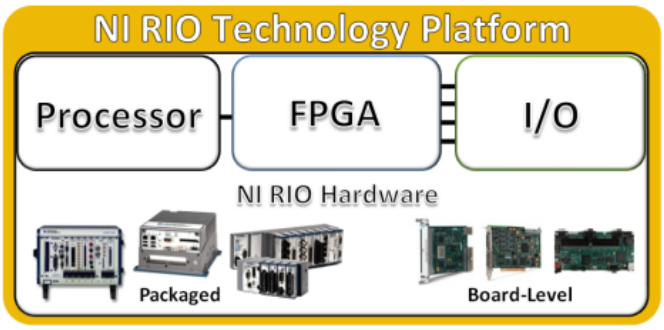


Figure 8. NI offers an entire family of RIO devices—both packaged and board-level—that you can program with LabVIEW based on this ideal hybrid architecture coupling both a microprocessor and an FPGA.

For the past nine years, National Instruments has implemented this powerful microprocessor plus FPGA architecture in its reconfigurable I/O (RIO) devices. These devices span many different form factors, from rugged to high-performance systems, all based on this same RIO architecture.

>> Learn More About NI FPGA-Based RIO Hardware Targets

### FPGA Design Tools

Now that you know the building blocks of an FPGA chip, you may ask, "How do you configure all of these millions of components to build up the logic that you need to execute?"

The answer is that you define digital computing tasks in software using development tools and then compile them down to a configuration file or bitstream that contains information on how the components should be wired together. The challenge in the past with FPGA technology was that the low-level FPGA design tools could be used only by engineers with a deep understanding of digital hardware design. However, the rise of high-level synthesis (HLS) design tools, such as NI LabVIEW system design software, changes the rules of FPGA programming and delivers new technologies that convert graphical block diagrams into digital hardware circuitry.

#### Traditional FPGA Design Tools

Through the first 20 years of FPGA development, hardware description languages (HDLs) such as VHDL and Verilog evolved into the primary languages for designing the algorithms running on the FPGA chip. These low-level languages integrate some of the benefits offered by other textual languages with the realization that on an FPGA, you are architecting a circuit. The resulting hybrid syntax requires signals to be mapped or connected from external I/O ports to internal signals, which ultimately are wired to the functions that house the algorithms. These functions execute sequentially and can reference other functions within the FPGA. However, the true parallel nature of the task execution on an FPGA is hard to visualize in a sequential line-by-line flow. HDLs reflect some of the attributes of other textual languages, but they differ substantially because they are based on a dataflow model where I/O is connected to a series of function blocks through signals.

To then verify the logic created by an FPGA programmer, it is common practice to write test benches in HDL to wrap around and exercise the FPGA design by asserting inputs and verifying outputs. The test bench and FPGA code are run in a simulation environment that models the hardware timing behavior of the FPGA chip and displays all of the input and output signals to the designer for test validation. The process of creating the HDL test bench and executing the simulation often requires more time than creating the original FPGA HDL design itself.

Once you have created an FPGA design using HDL and verified it, you need to feed it into a compilation tool that takes the text-based logic and, through several complex steps, synthesizes your HDL down into a configuration file or bitstream that contains information on how the components should be wired together. As part of this multistep manual process, you often are required to specify a mapping of signal names to the pins on the FPGA chip that you are using.

```
-- First we sychronize the asynchronous digital input to our clock
-- by inserting two flip flops.
SynchronizationFFs:
process( aReset, Clk )
begin
  if aReset then
    cDigitalInput_ms <= false;
    cDigitalInput <= false;
  elsif rising_edge(Clk) then
    cDigitalInput_ms <= aDigitalInput;
    cDigitalInput <= cDigitalInput_ms;
  end if;
end process SynchronizationFFs;

-- Then we keep track of what the digital input was on the previous
-- clock cycle by inserting another flip flop
PreviousDigitalInputFF:
process( aReset, Clk )
begin
  if aReset then
    cPrevDigitalInput <= false;
  elsif rising_edge(Clk) then
    cPrevDigitalInput <= cDigitalInput;
  end if;
end process PreviousDigitalInputFF;

-- Then we have a little combinatorial logic to detect a rising edge
cRisingEdgeDetected <= cDigitalInput and not cPrevDigitalInput;

-- And finally we have a register that increments when that rising
-- edge is detected.
CounterRegister:
process( aReset, Clk )
begin
  if aReset then
    cCountReg <= (others=>'0');
  elsif rising_edge(Clk) then
    if cRisingEdgeDetected then
      cCountReg <= cCountReg + 1;
    end if;
  end if;
end process CounterRegister;
cCount <= cCountReg;

end rtl;
```

Figure 9. Simple Counter FPGA Design in VHDL

Ultimately, the challenge in this design flow is that the expertise required to program in traditional HDLs is not widespread, and as a result, FPGA technology has not been accessible to the vast majority of engineers and scientists.

#### High-Level Synthesis Design Tools

The emergence of graphical HLS design tools, such as LabVIEW, has removed some of the major obstacles of the traditional HDL design process. The LabVIEW programming environment is distinctly suited for FPGA programming because it clearly represents parallelism and data flow, so users who are both experienced and inexperienced in traditional FPGA design processes can leverage FPGA technology. In addition, so that previous intellectual property (IP) is not lost, you can use LabVIEW to integrate existing VHDL into your LabVIEW FPGA designs.

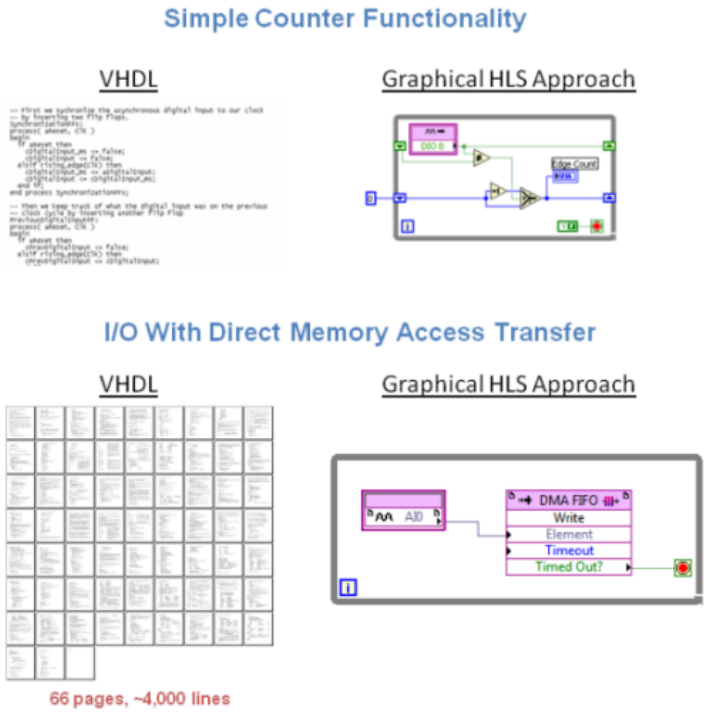


Figure 10. On the right is LabVIEW system design software, which is a high-level design tool for FPGAs in NI RIO hardware devices. It provides abstraction for the low-level complexity often found when creating and scaling VHDL designs.

Then to simulate and verify the behavior of your FPGA logic, LabVIEW offers features directly in the development environment. Without knowledge of the low-level HDL language, you can create test benches to exercise the logic of your design. In addition, the flexibility of the LabVIEW environment helps more advanced users model the timing and logic of their designs by exporting to cycle-accurate simulators such as Xilinx ISim.

LabVIEW FPGA compilation tools automate the compilation process, so you can start the process with a click of a button and receive reports and errors, if any, as compilation stages are completed. If timing errors do occur because of your FPGA design, LabVIEW highlights these critical paths graphically to expedite the debugging process.

>> Learn More by Viewing the Introduction to LabVIEW FPGA Webcast

## Conclusion

The adoption of FPGA technology continues to increase as higher-level tools such as LabVIEW, the standard microprocessor, and the FPGA RIO architecture are making FPGAs more accessible. It is still important, however, to look inside the FPGA and appreciate how much is actually happening when block diagrams are compiled down to execute in silicon. Comparing and selecting hardware targets based on flip-flops, LUTs, multipliers, and block RAM is the best way to choose the right FPGA chip for your application. Understanding resource usage is extremely helpful during development, especially when optimizing for size and speed. This paper is not meant to be a comprehensive list of all FPGA fundamental building blocks. You can learn more about FPGAs and digital hardware design from the resources below.

## Next Steps

Evaluate NI FPGA Hardware and Software
Learn About FPGA Design With NI LabVIEW System Design Software
Explore the Different NI FPGA Hardware Products