BASICS of Design

FPGAs

David Maliniak, Electronic Design Automation Editor

Tradeoffs Abound in FPGA Design

Understanding device types and design flows is key to getting the most out of FPGAs

Field-programmable gate arrays (FPGAs) arrived in 1984 as an alternative to programmable logic devices (PLDs) and ASICs. As their name implies, FPGAs offer the significant benefit of being readily programmable. Unlike their forebearers in the PLD category, FPGAs can (in most cases) be programmed again and again, giving designers multiple opportunities to tweak their circuits.

There's no large non-recurring engineering (NRE) cost associated with FPGAs. In addition, lengthy, nerve-wracking waits for mask-making operations are squashed. Often, with FPGA development, logic design begins to resemble software design due to the many iterations of a given design. Innovative design often happens with FPGAs as an implementation platform.

But there are some downsides to FPGAs as well. The economics of FPGAs force designers to balance their relatively high piece-part pricing compared to ASICs with the absence of high NREs and long development cycles. They're also available only in fixed sizes, which matters when you're determined to avoid unused silicon area.

What are FPGAs?

FPGAs fill a gap between discrete logic and the smaller PLDs on the low end of the complexity scale and costly custom ASICs on the high end. They consist of an array of logic blocks that are configured using software. Programmable I/O blocks surround these logic blocks. Both are connected by programmable interconnects (Fig. 1). The programming technology in an FPGA determines the type of basic logic cell and the interconnect scheme. In turn, the logic cells and interconnection scheme determine the design of the input and output circuits as well as the programming scheme.

Just a few years ago, the largest FPGA was measured in tens of thousands of system gates and operated at 40 MHz. Older FPGAs often cost more than $150 for the most advanced parts at the time. Today, however, FPGAs offer millions of gates of logic capacity, operate at 300 MHz, can cost less than $10, and offer integrated functions like processors and memory (Table 1).

FPGAs offer all of the features needed to implement most complex designs. Clock management is facilitated by on-chip PLL (phase-locked loop) or DLL (delay-locked loop) circuitry. Dedicated memory blocks can be configured as basic single-port RAMs, ROMs, FIFOs, or CAMs. Data processing, as embodied in the devices' logic fabric, varies widely. The ability to link the FPGA with backplanes, high-speed buses, and memories is afforded by support for various single-ended and differential I/O standards. Also found on today's FPGAs are system-building resources such as high-speed serial I/Os, arithmetic modules, embedded processors, and large amounts of memory.

Initially seen as a vehicle for rapid prototyping and emulation systems, FPGAs have spread into a host of applications. They were once too simple, and too costly, for anything but small-volume production. Now, with the advent of much larger devices and declining per-part costs,

Do's And Don'ts For The FPGA Designer

Do's

1. Do concentrate on I/O timing, not just the register-to-register internal frequency that the FPGA place-and-route tools report. Frequently, the hardest challenge in a complete FPGA design is the I/O timing. Focus on how your signals enter and leave your FPGA, because that's where the bottlenecks frequently occur.

2. Do create hierarchy around vendor-specific structures and instantiations. Give yourself the freedom to migrate from one technology to another by ensuring that each instantiation of a vendor-specific element is in a separate hierarchical block. This applies especially to RAMs and clock-management blocks.

3. Do use IP timing models during synthesis to give the true picture of your design. By importing EDIF netlists of pre-synthesized blocks, your synthesis tool can fully understand your timing requirements. Be cautious when using vendor cores that you can bring into your synthesis tool if they have no timing model.

4. Do design your hierarchical blocks with registered outputs where possible to avoid having critical paths pass through many levels of hierarchy. FPGAs exhibit step-functions in logic-limited performance. When hierarchy is preserved and the critical path passes across a hierarchical boundary, you may introduce an extra level of logic. When considered along with the associated routing, this can add significant delay to your critical path.

5. Do enable retiming in your synthesis tool. FPGAs tend to be register-rich architectures. When you correctly constrain your design in synthesis, you allow the tool to optimize your design to take advantage of positive slack timing within the design. Sometimes this can be done after initial place and route to improve retiming over wireload estimation.

Dont's

1. Don't synthesize unless you've fully and correctly constrained your design. This includes correct clock domains, I/O timing requirements, multicycle paths, and false paths. If your synthesis tool doesn't see exactly what you want, it can't make decisions to optimize your design accordingly.

2. Don't try to fix every timing problem in place and route. Place and route offers little room for fixing timing where a properly constrained synthesis tool would.

3. Don't vainly floor plan at the RTL or block level hoping to improve place-and-route results. Manual area placement can cause more problems than it might initially appear to solve. Unless you are an expert in manual placement and floorplanning, this is best left alone.

4. Don't string clock buffers together, create multiple clock trees from the same clock, or use multiple clocks when a simple enable will do. Clocking schemes in FPGAs can become very complicated now that there are PLLs, DLLs, and large numbers of clock-distribution networks. Poor clocking schemes can lead to extended place-and-route times, failure to meet timing, and even failure to place in some technologies. Simpler schemes are vastly more desirable. Avoid those gated clocks, too!

5. Don't forget to simulate your design blocks as well as your entire design. Discovering and back-tracking an error from the chip's pins during on-board testing can be extremely difficult. On-board FPGA testing can miss important design flaws that are much easier to identify during simulation; they can be rectified by modifying the FPGA's programming.

FPGAs are finding their way off the prototyping bench and into production (Table 2).

Comparing FPGA Architectures

FPGAs must be programmed by users to connect the chip's resources in the appropriate manner to implement the desired functionality. Over the years, various technologies have emerged to suit different requirements. Some FPGAs can only be programmed once. These devices employ antifuse technology. Flash-based devices can be programmed and reprogrammed again after debugging. Still others can be dynamically programmed thanks to SRAM-based technology. Each has its advantages and disadvantages (Table 3).

Most modern FPGAs are based on SRAM configuration cells, which offer the benefit of unlimited reprogrammability. When powered up, they can be configured to perform a given task, such as a board or system test, and then reprogrammed to perform their main task. On the flip side, though, SRAM-based FPGAs must be reconfigured each time their host system is powered up, and additional external circuitry is required to do so. Further, because the configuration file used to program the FPGA is stored in external memory, security issues concerning intellectual property emerge.

Antifuse-based FPGAs aren't in-system programmable,

Table 1: KEY RESOURCES AVAILABLE IN THE LARGEST DEVICES FROM MAJOR FPGA VENDORS

Features	Xilinx Virtex II Pro	Altera Stratix	Actel Axcelerator	Lattice ispXPGA
Clock management	DCM Up to 12	PLL Up to 12	PLL Up to 8	SysCLOCK PLL Up to 8
Embedded memory blocks	BlockRAM Up to 10 Mbits	TriMatrix memory Up to 10 Mbits	Embedded RAM Up to 338 kbits	SysMEM blocks Up to 414 kbits
Data processing	Configurable logic blocks and 18-bit by 18-bit multipliers Up to 125,000 logic cells and 556 multiplier blocks	Logic elements and embedded multipliers Up to 79,000 LEs and 176 embedded multipliers	Logic modules (C-Cell and R-Cell) Up to 10,000 R-Cells and 21,000 C-Cells	Based on programmable functional unit Up to 3844 PFUs
Programmable I/Os	SelectI/O	Advanced I/O support	Advanced I/O support	SysI/O
Special features	Embedded PowerPC 405 cores RocketI/O multi-gigabit transceiver	DSP blocks High-speed differential I/O and interface standards support	PerPin FIFOs for bus applications	SysHSI for high-speed serial interface

but rather are programmed offline using a device programmer. Once the chip is configured, it can't be altered.

However, in antifuse technology, device configuration is nonvolatile with no need for external memory. On top of that, it's virtually impossible to reverse-engineer their programming. They often work as replacements for ASICs in small volumes.

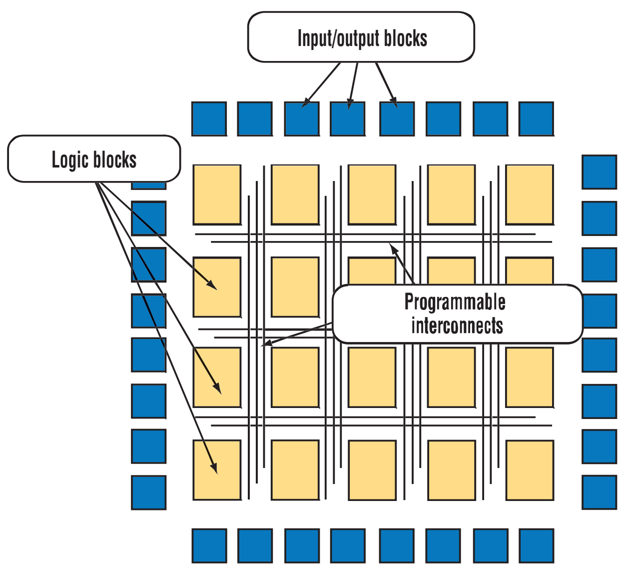
In a sense, flash-based FPGAs fulfill the promise of FPGAs in that they can be reprogrammed many times. They're nonvolatile, retaining their configuration even when powered down. Programming is done either in-system or with a programmer. In some cases, IP security can be achieved using a multibit key that locks the configuration data after programming.

But flash-based FPGAs require extra process steps above and beyond standard CMOS technology, leaving them at least a generation behind. Moreover, the many pull-up resistors result in high static power consumption.

FPGAs can also be characterized as having either fine-, medium-, or coarse-grained architectures. Fine-grained architectures boast a large number of relatively simple logic blocks. Each logic block usually contains either a two-input logic function or a 4-to-1 multiplexer and a flip-flop. Blocks can only be used to implement simple functions. But fine-grained architectures lend themselves to execution of functions that benefit from parallelism.

Figure 1

Just about all FPGAs include a regular, programmable, and flexible architecture of logic blocks surrounded by input/output blocks on the perimeter. These functional blocks are linked together by a hierarchy of highly versatile programmable interconnects.



Coarse-grained architectures consist of relatively large logic blocks often containing two or more lookup tables and two or more flip-flops. In most of these architectures, a four-input lookup table (think of it as a 16 x 1 ROM) implements the actual logic.

The FPGA design flow

After weighing all implementation options, you must consider the design flow. The process of implementing a design on an FPGA can be broken down into several stages, loosely definable as design entry or capture, synthesis, and place and route (Fig. 2). Along the way, the design is simulated at various levels of abstraction as in ASIC design. The availability of sophisticated and coherent tool suites for FPGA design makes them all the more attractive.

At one time, design entry was performed in the form of schematic capture. Most designers have moved over to hardware description languages (HDLs) for design entry. Some will prefer a mixture of the two techniques. Schematic-based design-capture tools gave designers a great deal of control over the physical placement and partitioning of logic on the device. But it's becoming less likely that designers will take that route. Meanwhile, language-based design entry is faster, but often at the expense of performance or density.

For many designers, the choice of whether to use schematic- or HDL-based design entry comes down to their conception of their design. For those who think in software or algorithmic-like terms, HDLs are the better choice. HDLs are well suited for highly complex designs, especially when the designer has a good handle on how the logic must be structured. They can also be very useful for designing smaller functions when you haven't the time or inclination to work through the actual hardware implementation.

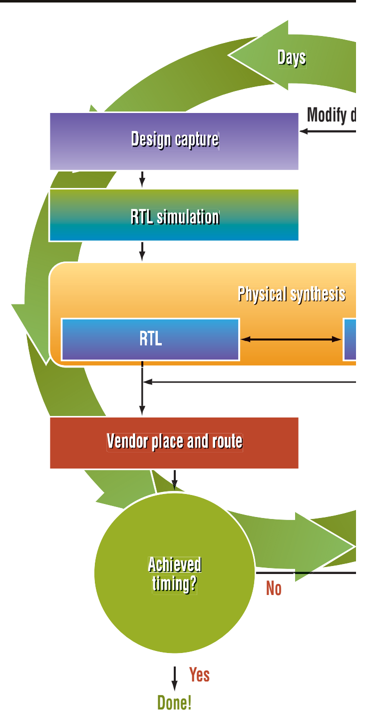
On the other hand, HDLs represent a level of abstraction that can isolate designers from the details of the hardware implementation. Schematic-based entry gives designers much more visibility into the hardware. It's a better method for those who are hardware-oriented. The downside of schematic-based entry is that it makes the design more difficult to modify or port to another FPGA.

A third option for design entry, state-machine entry, works well for designers who can see their logic design as a series of states that the system steps through. It shines when designing somewhat simple functions, often in the area of system control, that can be clearly represented in visual formats. Tool support for finite state-machine entry is limited, though.

Some designers approach the start of their design from a level of abstraction higher

Figure 2

A "big picture" look at an FPGA design flow shows t… design entry, synthesis from RTL to gate level, and … route is done using the FPGA vendors' proprietary t… devices' architectures and logic-block structures.



consider
on an
definable
ute (Fig.
evels of
histicated
all the

n of
o hardware
will prefer
ign-cap-
he physical
it's
. Mean-
at the

e schemat-
their con-
software
choice.
ially when
must be
ing small-
ion to
n.
abstraction
hardware

Table 2: FPGA USAGE

	Emulation: 3%	Prototyping: 30%	Preproduction: 30%	Production: 37%
Time-to-market	Fairly high; fast compile times	Fairly high; fast compile times	Fairly high; fast compile times	Fairly high; fast compile times
Performance	Not stringent	Not stringent	Very critical	Very critical
Volume	Very low per application	Low per application	Moderately high per application	High per application

Table 3: ADVANTAGES/DISADVANTAGES OF VARIOUS FPGA TECHNOLOGIES

Feature	SRAM	Antifuse	Flash
Reprogrammable?	Yes (in-system)	No	Yes (in-system or offline)
Reprogramming speed (including erasure)	Fast	Not applicable	3X SRAM
Volatile?	Yes	No	No (but can be if required)
External configuration file?	Yes	No	No
Good for prototyping?	Yes	No	Yes
Instant-on?	No	Yes	Yes
IP security	Poor	Very good	Very good
Size of configuration cell	Large (six transistors)	Very small	Small (two transistors)
Power consumption	High	Low	Medium
Radiation hardness?	No	Yes	No

than HDLs, which is algorithmic design using the C/C++ programming languages. A number of EDA vendors have tool flows supporting this design style. Generally, algorithmic design has been thought of as a tool for architectural exploration. But increasingly, as tool flows emerge for C-level synthesis, it's being accepted as a first step on the road to hardware implementation.

After design entry, the design is simulated at the register-transfer level (RTL). This is the first of several simulation stages, because the design must be simulated at successive levels of abstraction as it moves down the chain toward physical implementation on the FPGA itself. RTL simulation offers the highest performance in terms of speed. As a result, designers can perform many simulation runs in an effort to refine the logic. At this stage, FPGA development isn't unlike software development. Signals and variables are observed, procedures and functions traced, and breakpoints set. The good news is that it's a very fast simulation. But because the design hasn't yet been synthesized to gate level, properties such as timing and resource usage are still unknowns.

The next step following RTL simulation is to convert the RTL representation of the design into a bit-stream file that can be loaded onto the FPGA. The interim step is FPGA synthesis, which translates the VHDL or Verilog code into a device netlist format that can be understood by a bit-stream converter.

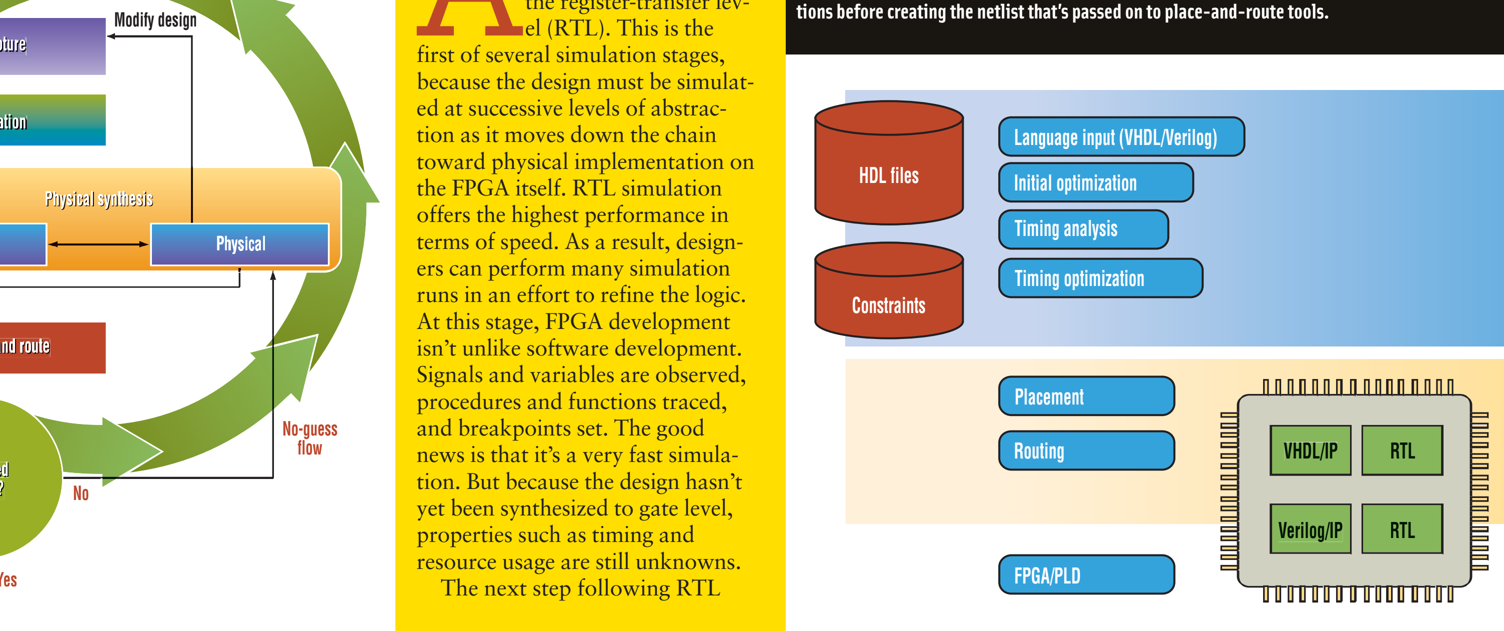
The synthesis process can be broken down into three steps. First, the HDL code is converted into device netlist format. Then the resulting file is converted into a hexadecimal bit-stream file, or .bit file. This step is necessary to change the list of required devices and interconnects into hexadecimal bits to download to the FPGA. Lastly, the .bit file is downloaded to the physical FPGA. This final step completes the FPGA synthesis procedure by programming the design onto the physical FPGA.

It's important to fully constrain designs before synthesis (Fig. 3). A constraint file is an input to the synthesis process just as the RTL code itself. Constraints can be applied globally or to specific portions of the design. The synthesis engine uses these constraints to optimize the netlist. However, it's equally important to not over-constrain the design, which will generally result in less-than-optimal results from the next step in the implementation process—physical device placement—and interconnect routing.

PGA design flow shows the major steps in the process:
m RTL to gate level, and physical design. Place and
A vendors' proprietary tools that account for the
ogic-block structures.

Figure 3

The implementation flow for FPGAs begins with synthesis of the HDL design description into a gate-level n
Accounting for user-defined design constraints on area, power, and speed, the tool performs various optim
tions before creating the netlist that's passed on to place-and-route tools.



Synthesis c
This trad
iterations b
have incorp
which auto
across regis
also anticip
F oll
ne
int
do
optimizatio
ware partit
Good parti
and high pe
Increasin
after synthe
work from
Floorplann
good idea t
After par
to place the
monitors ro
blocks. It m
delays to m
Overall, the
route.
Function
synthesis a
This step e
After imple
tion step w
placement
delays are b
netlist for t

Synthesis constraints soon become place-and-route constraints.

This traditional flow will work, but it can lead to numerous iterations before achieving timing closure. Some EDA vendors have incorporated more modern physical synthesis techniques, which automate device re-timing by moving lookup tables (LUTs) across registers to balance out timing slack. Physical synthesis also anticipates place and route to leverage delay information.

Following synthesis, device implementation begins. After netlist synthesis, the design is automatically converted into the format supported internally by the FPGA vendor's place-and-route tools. Design-rule checking and optimization is performed on the incoming netlist and the software partitions the design onto the available logic resources. Good partitioning is required to achieve high routing completion and high performance.

Increasingly, FPGA designers are turning to floorplanning after synthesis and design partitioning. FPGA floorplanners work from the netlist hierarchy as defined by the RTL coding. Floorplanning can help if area is tight. When possible, it's a good idea to place critical logic in separate blocks.

After partitioning and floorplanning, the placement tool tries to place the logic blocks to achieve efficient routing. The tool monitors routing length and track congestion while placing the blocks. It may also track the absolute path delays to meet the user's timing constraints. Overall, the process mimics PCB place and route.

Functional simulation is performed after synthesis and before physical implementation. This step ensures correct logic functionality. After implementation, there's a final verification step with full timing information. After placement and routing, the logic and routing delays are back-annotated to the gate-level netlist for this final simulation. At this point, simulation is a much longer process, because timing is also a factor (Fig. 4). Often, designers substitute static timing analysis for timing simulation. Static timing analysis calculates the timing of combinational paths between registers and compares it against the designer's timing constraints.

Once the design is successfully verified and found to meet timing, the final step is to actually program the FPGA itself. At the completion of placement and routing, a binary programming file is created. It's used to configure the device. No matter what the device's underlying technology, the FPGA interconnect fabric has cells that configure it to connect to the inputs and outputs of the logic blocks. In turn, the cells configure those logic blocks to each other. Most programmable-logic technologies, including the PROMs for SRAM-based FPGAs, require some sort of a device programmer. Devices can also be programmed through their configuration ports using a set of dedicated pins.

Modern FPGAs also incorporate a JTAG port that, happily, can be used for more than boundary-scan testing. The JTAG port can be connected to the device's internal SRAM configuration-cell shift register, which in turn can be instructed to connect to the chip's JTAG scan chain.

If you've gotten this far with your design, chances are you have a finished FPGA. There's one more step to the process, however, which is to attach the device to a printed-circuit board in a system. The appearance of 10-Gbit/s serial transmitters, or I/Os, on the chip, coupled with packages containing as many as 1500 pins, makes the interface between the FPGA and its intended system board a very sticky issue. All too often, an FPGA is soldered to a pc board and it doesn't function as expected or, worse, it doesn't function at all. That can be the result of errors caused by manual placement of all those pins, not to mention the board-level timing issues created by a complex FPGA.

More than ever, designers must strongly consider an integrated flow that takes them from conception of the FPGA through board design. Such flows maintain complete connectivity between the system-level design and the FPGA; they also do so between design iterations. Not only do today's integrated FPGA-to-board flows create the schematic connectivity needed for verification and layout of the board, but they also document which signal connections are made to which device pins and how these map to the original board-level bus structures.

Integrated flows for FPGAs make sense in general, considering that FPGA vendors will continue to introduce more complex, powerful, and economical devices over time. An integrated third-party flow makes it easier to re-target a design to different technologies from different vendors as conditions warrant.

Figure 4

FPGA simulation occurs at various stages of the design process: after RTL design, after synthesis, and once again after implementation. The latter is a final gate-level check, accounting for actual logic and interconnect delays, of logic functionality.