

Digital Design Basics

Cliff Brake
2002-11-21

Abstract

With the advances of FPGA (field programmable gate array) technology, many engineers now have the opportunity to do medium scale digital design that involves more than connecting a few large chips with glue logic. This paper attempts to address the basic design issues that must be understood to accomplish a medium scale FPGA design.

FPGA Background

FPGA devices are one of the modern technologies that are changing the electronic industry. FPGAs are riding the same integrated circuit process curves as processors and memories and keep getting larger, faster, and cheaper. FPGAs are now common in low and mid volume embedded products where they offer the following advantages:

- Fast time to market
- Better integration
- In system programmability
- FPGAs tend to have long life cycles and are usually replaced with pin compatible parts
- Cores such as PCI are available and able to integrate with relative ease

Getting started in FPGA design is easy. The tools are cheap (and sometimes free) for low-end devices and affordable for the high end. Modern HDL (hardware design language) environments are very powerful for creating and verifying a design. There is plenty of documentation available for using different vendor's FPGA design tools and exploiting features of different FPGAs. There is also a great supply of books that cover the details of HDL languages and how to use them. Even with modern tools, the fundamentals of digital design still remain intact and must be understood. If the fundamentals are ignored, there is a good chance that your design will not work consistently and will probably exhibit intermittent modes of operation. Digital design with FPGAs can get complex. Many of the approaches used in software design can also be applied to modern digital design. Keep things clean and modular. Make sure you have a methodology for design specification, implementation, and verification. Understand the basic tasks your tools must do for you. The topics presented in this paper are well understood by digital designers with several years of experience. Hopefully this paper will be of assistance to

designers getting started in FPGA digital design who do not have access to an experienced digital design team to answer questions.

Asynchronous vs. Synchronous design

One of the most important digital design fundamentals is that of asynchronous and synchronous design. Synchronous design refers to digital designs that use a common clock where signals change only after clock edges. Asynchronous design covers everything else. Synchronous design is preferred in most applications because it is nicely constrained and the tools support it. As designs become more complex, we have to rely on tools to synthesize logic. It is no longer feasible to handcraft large amounts of asynchronous logic at the gate level. Unfortunately, the real world is asynchronous; therefore, most designs contain some amount of asynchronous logic.

The basic requirement in a synchronous design is that inputs must arrive at flops (registers) before the clock does. FPGA implementation tools can accurately predict the maximum delay from the output of one flop to the input of another flop. If the delay is too long (longer than the clock period), the design can be modified until all delays are shorter than the clock period. The problem arises when asynchronous inputs are fed into a synchronous design. If an asynchronous input is used as one of the control inputs for a state machine, eventually the situation will occur when the input is feeding multiple flops. Because of differing combinational logic and routing delays, the input reaches flop A on a different clock cycle as flop B. The state machine can then enter an unintended state. If the unintended state is undefined, the state machine will most likely crash (quit responding). Thus, asynchronous inputs must be synchronized using one or more flops before they can be used in a synchronous design.

Metastability is a variant on the above scenario and occurs when a flop input switches very close to the clock edge and does not meet minimum setup time requirements. Because a flop is really a high-gain amplifier, the output can go metastable and not switch within the standard clock-to-output delay time for the flop. The output of the metastable flop is now an asynchronous signal. The logical conclusion is that it is impossible to synchronize asynchronous signals.

Fortunately, metastability is a well understood statistical phenomena that can be described with an equation that is a function of clock rate, input signal switching frequency, maximum flop settling time, and a few constants determined by the logic family characteristics. For non-mission critical designs, the solution to metastability is to design the synchronizing logic such that the probability of metastability is at a tolerable level. For mission critical systems, the design should be able to recover from a metastable condition. If the asynchronous input signal is a debounced user push-button switch, and the clock rate is reasonably slow, the probability of seeing metastable behavior is very low. If you are synchronizing datapath control signals between two asynchronous high-speed clock domains, do the metastability calculations. A standard way to improve metastability performance is to add synchronization stages or flops at the cost of a delayed input signal. This allows all but the last synchronizing stage to go metastable without adversely affecting the synchronous logic it is driving.

System Design Issues

A FPGA designer should understand several key points about the system the FPGA is going to be used in. Determine what the FPGA is going to do for the system and how it will interact with the rest of the system. The classic approach is to break a digital design down into datapath and control logic. Generally the purpose of digital logic in a system is to move data. If the FPGA is involved in a datapath, design this first. Figure out what data has to be moved and how fast. Understand and document the datapath requirements fully before designing the control architecture.

Two important issues that are usually dictated by the system is the reset and clock strategy. Bandwidth of the datapath and available system clocks will probably dictate the clock speed. Signals that connect two different asynchronous clock domains must be synchronized. Synchronization costs clock cycles and sometimes requires FIFOs to maintain data throughput. The reset strategy is a simple, but very critical aspect of any digital design. In any reset scenario, reset should be deasserted synchronously with the clock so that all flops exit reset on the same clock cycle. There are two basic scenarios. Reset scenario number one is where the system comes out of

reset and then the clock starts running. This scenario can occur when using a clock output from a microcontroller to clock a FPGA. The system is taken out of reset and then the oscillator circuitry in the microcontroller starts running. An asynchronous reset is required in this case. Designers should be aware that some clock ICs output a series of runt pulses while the oscillator starts up. If the clock does not start up cleanly, the FPGA should be held in reset until the clock is stable. Reset scenario number two is where reset is deasserted after the clock is running. In this case, it is necessary for reset to be synchronous to the clock so that all flops exit reset on the same clock. If the reset is asynchronous such as specified in PCI v2.1 specification, it is safest to synchronize reset inside the FPGA. One elegant solution that came out of a discussion on the PCI email reflector is shown in Illustration (1). In this circuit, the internal reset to the FPGA can be asynchronously set, but reset does not deactivate until a clock edge occurs. This allows the chip to be put in reset without the clock running, but forces the chip to come out of reset synchronously.

FPGA Design

Now that the system datapath, control architecture, reset, and clock strategies are understood and documented, the actual FPGA design can start. The three primary considerations for selecting a FPGA are speed, pin count, and amount of logic. If there is time for 5-7 levels of combinational logic between flops, the design should be fairly straightforward. More levels of combinational logic between flops will require more careful design and more pipelining effort. There are numerous other considerations that are more device specific such as I/O buffer features, on chip PLLs, available tools and cores, on chip RAM, etc. It is a good idea to write a functional specification for the design. This forces the designer to understand what the chip is supposed to do before implementing large amounts of logic and will lead to cleaner

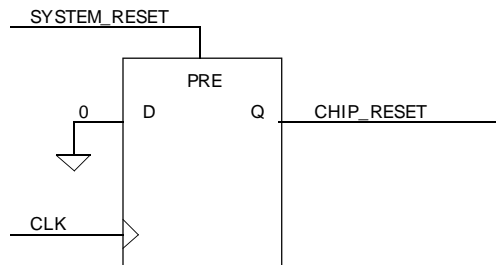


Illustration (1): System Reset Circuit

implementations. A functional specification is also very useful for the system designer who will eventually use the FPGA. The partitioning of the FPGA design falls out after the functional specification. Partitioning a design by functional blocks is a logical partitioning strategy, but it may also be necessary to partition parts of the design by routing strategy if there are performance critical parts of the design. A rough guideline that seems to work well is that the lowest level blocks in the design should be 2000 to 5000 gates in

size. The top level module should be used only to connect lower level blocks. Avoid logic in the top level module if possible.

The design entry and implementation is where the tools start to become important. VHDL and Verilog are industry standard HDLs (hardware description languages) for digital design entry and simulation. Text based code still seems to be the preferred way to express large, complex designs as evidenced by the fact the most people still write software in text based languages. The terms RTL (Register Transfer Level) and Structural HDL are often used to describe two different levels of HDL. The core of the design entry code is usually written at the RTL level but also contains structural HDL code to connect modules together. Structural HDL is essentially a netlist that describes how different components are tied together. A HDL file of a routed design created by place and route tools is a structural HDL level file. Behavioral HDL is a third level of HDL that is used to model the behavior of a device and is used only during simulation. Behavioral HDL is generally not used for design entry as it is difficult to synthesize. Schematic entry is still used for FPGA design entry but is cumbersome for large designs and is more difficult to simulate and verify. The general flow for a HDL based design is shown in Illustration (2). There are additional steps such as timing simulation of the routed design and timing analysis at the synthesis level, but the majority of the time will be spent iterating the steps show in Illustration (2). Investing some time in setting up a good simulation environment is necessary for a large design and will save you time on any design. Some of the low end FPGA design tools do not include a source level HDL simulator which must be purchased separately. It is a good idea to synthesize each block as it is coded so that inefficient and slow designs can be fixed before the entire chip is synthesized. Registering the outputs of blocks will make meeting timing requirements easier but might require more FPGA resources.

Timing and related tools can be somewhat of a mystery in FPGAs.

For synchronous logic, you must verify that the inputs of flops arrive before the clock does. This can be controlled in a large part by carefully coding the design to minimize the levels of combinational logic between flops. After synthesizing some code and looking at the timing results, you quickly learn how to code for synthesis. Pipelining is a standard technique for speeding

up a design and is done by breaking up large amounts of combinational logic into several stages with flops between the stages. Medium speed designs (30MHz +) will probably require some degree of pipelining. Timing can also be improved by entering timing constraints that the place and route tools try to meet. Timing constraints should be standard part of any FPGA design entry. The following groups of constraints are typical:

- Input pin to flop input
- Flop to flop (must be less than the clock period)
- Flop to output pin
- Pin to pin (asynchronous stuff)

Higher end synthesis tools can also use timing constraints. Static timing analysis is the primary timing verification tool that checks the worst case delays against the timing constraints entered by the designer. Timing simulations of routed designs are useful for debugging and sanity checks, but can be misleading because a timing simulation will most likely not exercise every possible combinational path.

After the design is implemented in real hardware, the debug phase starts. FPGA designs are difficult to debug on the bench because internal signals are not visible unless routed to unused pins or analyzed by special FPGA debug equipment. This is where a good verification environment pays off. The design should be mostly correct if you did the functional simulations and static timing analysis. Chances are some corners were cut and some problems will come up in unverified parts of the design. Once the symptoms are identified, the designer can go back to the functional simulation, recreate the scenario and observe what is going on inside the chip. If timing problems are suspected, a timing simulation can be run on a structural model of the chip.

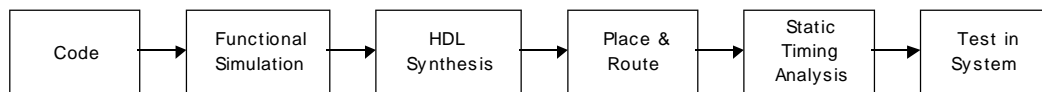


Illustration (2): Digital Design Flow

Timing simulations are useful for finding asynchronous problems.

That's it! If you understand the fundamentals and lay solid foundation for your design, modern digital design can be a rewarding experience.