Introduction to an FPGA Tool-chain

Spring Embedded Systems Conference
San Francisco ❖ 2004
Course ESC-348

Many engineers are confused about the difference between typical software development targeted for an embedded microcontroller and the steps necessary to create a system using a hardware definition language (HDL). This session will discuss the different steps that one takes when turning a design into an embedded system using programmable logic via a HDL as well as highlighting the tools and intermediate products of the development process.

by: Michael T. Trader
michael.trader@eds.com

Introduction

Many traditional software engineers are interested in making the jump to developing embedded devices via programmable logic. New electrical and computers engineers graduating college have generally been exposed to developing these devices, but firmware developers who have been out of school for five, ten, or more years may not have had the opportunity to develop these kinds of systems. This paper will attempt to give enough of a general overview that those interested in developing this skill will feel comfortable jumping into this new realm.

This paper is written from the viewpoint of a seasoned 'C' software engineer. The target audience would be those who have been creating software for embedded systems for a few years, who may or may not have been exposed to the hardware engineering side of development, but who wants to know more about how a programmable logic device gets designed, constructed, implemented, and tested. This paper will focus on FPGAs (Field Programmable Gate Arrays), a family of reconfigurable logic, and VHDL – a hardware definition language.

Traditional System Design Methodology

Figure 1 is a generalized diagram showing a development flow chart that should be very familiar to software engineers. Typically, the topic of 'design' concerns itself with specification, requirements, and other natural language mechanisms for defining system behavior. For this paper, we will also consider the concepts of design entry when we talk about the topic of design. The topic of "construct" will include all discussion regarding how a design is transformed from natural language descriptions to final target representation. All relevant verification and validation efforts will be covered in the test section. For the remainder of this paper, the concept of release will be ignored.

Figure 1 is really only helpful in understanding the families of jobs that are performed. The flow also is only 'generally' accurate in its relative order of tasks. This flow doesn't allow for any iteration nor does it represent the widely held belief that testing should happen throughout the development cycle. As we shall see, the actual tool chains are not so restrictive. Before discussing the tools used for FPGA development, the 'C' software tools and tasks will be reviewed.

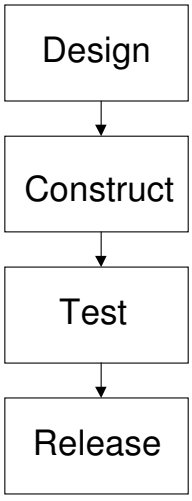

Figure 1: Synchronous Design Flow

Software Development

Figure 2 illustrates the typical steps one would execute to create a 'C' language embedded system. As mentioned previously, these steps are not necessarily executed in this exact order, but this is a good reference point for discussion and comparison to FPGA development. Since this paper is not a treatise on classical software development, we are going to move forward without significant discussion regarding the steps shown in figure 2.

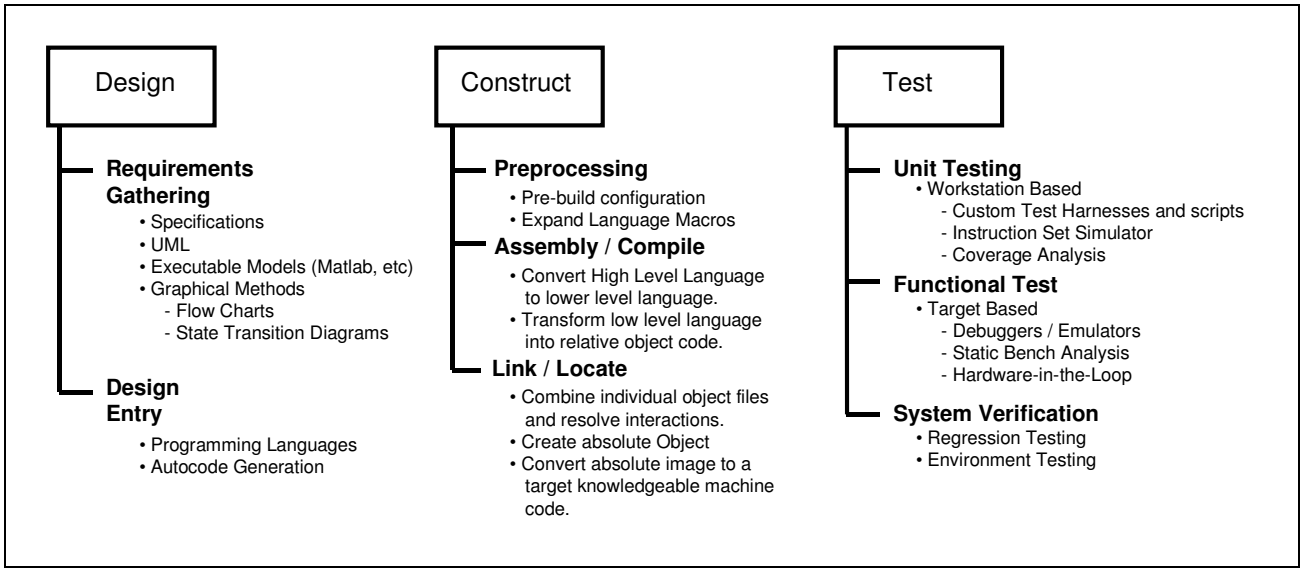

Figure 2: Software Development Steps

Figure 3 shows the comparable development steps when creating an embedded system employing a configurable logic chip like an FPGA. This example is modeled generally on the steps one would take when creating software for a Xilinx Spartan IIE class FPGA, but development for other vendors and products would be similar. The biggest differences in the tool chains revolve around idiosyncrasies in each vendor's tool suite and in the steps found in the "synthesize" and "implement" steps.

The "requirements" step of design is similar for FPGA development, but usually more specific requirements are specified regarding the performance of the physical part. Obviously there are performance requirements when developing 'C' software as well. In FPGA development, the constraints are used directly by several FPGA tools later during implementation (synthesis / translation).

Constraint files can contain desired performance regarding propagation delays between specific inputs and outputs. Constraints can be defined which dictate that certain areas of the FPGA chip be used for some specific functionality. Constraint files are also used to tell the synthesis tool which system input and outputs are tied to which pins. Obviously the system will be more efficient if the tool can determine the best design without being constrained by specific inputs and outputs having to be located on exact pins. However, many times, the designer doesn't have this flexibility.

Design entry for VHDL is conceptually the same as for 'C'; enter the code and, later on, try to build it using the synthesis tool (sometime referred to as the silicon compiler). Developers creating FPGA based systems generally have to be a little 'closer' to the hardware than traditional software programmers. As such, sometimes it is easier to define certain functionality with a circuit diagram. While this might seem completely foreign to a software developer (unless they have a background in engineering), it is rather practical when one realizes that the algorithms developed in this step are going to end up as a circuit again anyway. Developing FPGA behavior with circuit diagrams is called schematic capture.

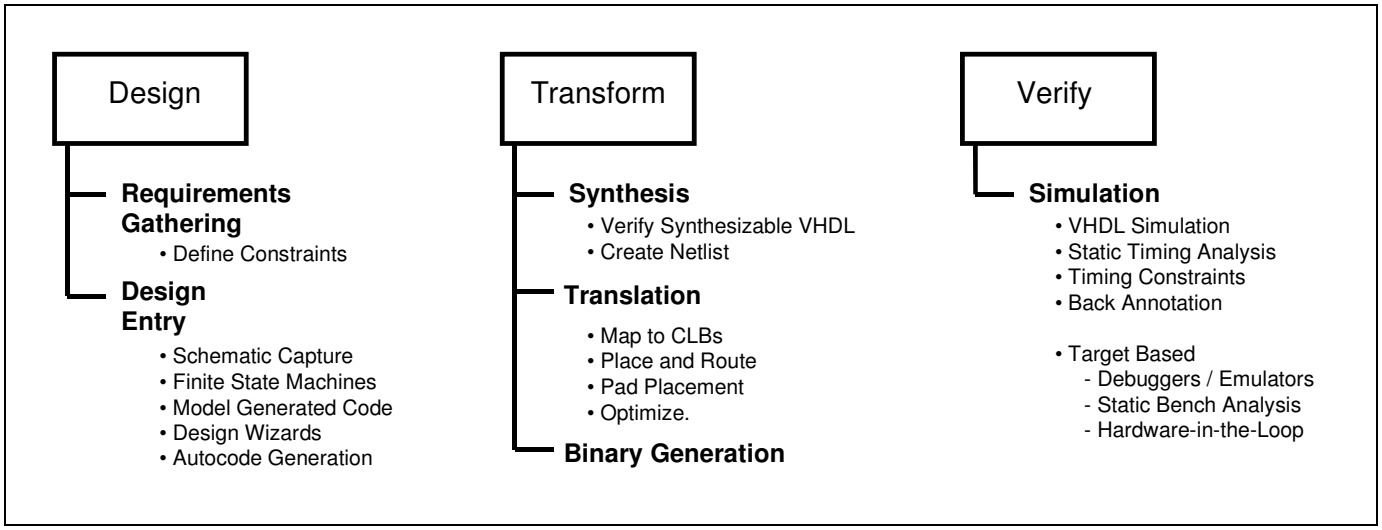

Figure 3: FPGA Development Steps

An example of VHDL can be found in Listing 1. This piece of VHDL defines debounce circuit functionality which can be used to qualify the input of a push-button switch. The circuit diagram that this VHDL describes can be found in figure 4. This functionality is a basic behavior that most embedded systems developers will already be fairly familiar with. The 'C' logic to implement the same behavior in a microcontroller is straightforward. Essentially this logic is attempting to keep a noisy input signal from creating multiple transitions. In addition, this logic will ensure that the output indicating a button press is of a fixed duration, regardless of how long the actual button is pressed.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity debounce is
       port (
              input, clock, reset: in std_logic;
              output: out std_logic
              );
end debounce;

architecture rtl of debounce is
signal D1, D2, D3: std_logic;
begin
process(clock, reset)
    begin
      if reset = '1' then
          D1 <= '0';
          D2 <= '0';
          D3 <= '0';
       elsif clock'event and clock='1' then
          D1 <= input;
          D2 <= D1;
          D3 <= D2;
       end if;
    end process;
    output <= D1 and D2 and (not D3);
end rtl;
```

Listing 1 – debounce.vhd: VHDL defining switch debounce behavior

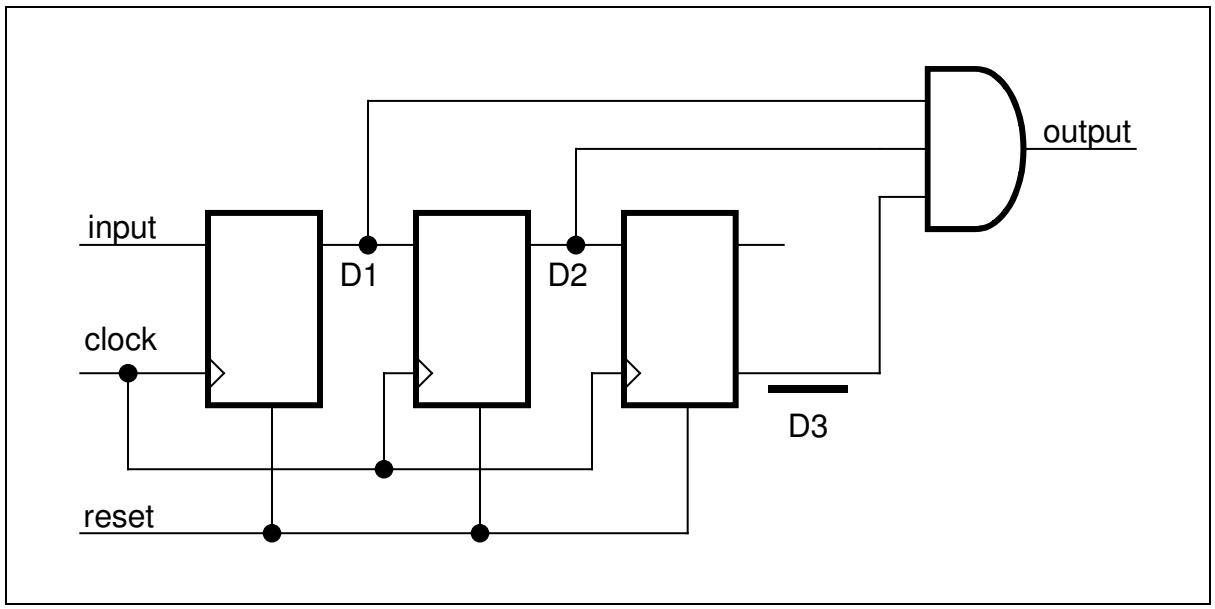

Figure 4: A circuit representation of the "debounce" VHDL in listing 1.

A system can then be built up around this function. Other functions (multiplexers, registers, latches, stacks, etc.) would be created to develop the primitives. These primitives are then combined to create more sophisticated modules that form the basis of the system. Each of these functions can be simulated before being synthesized to ensure that the function works as desired. Figure 5 is a representative waveform out of a VHDL simulation tool.

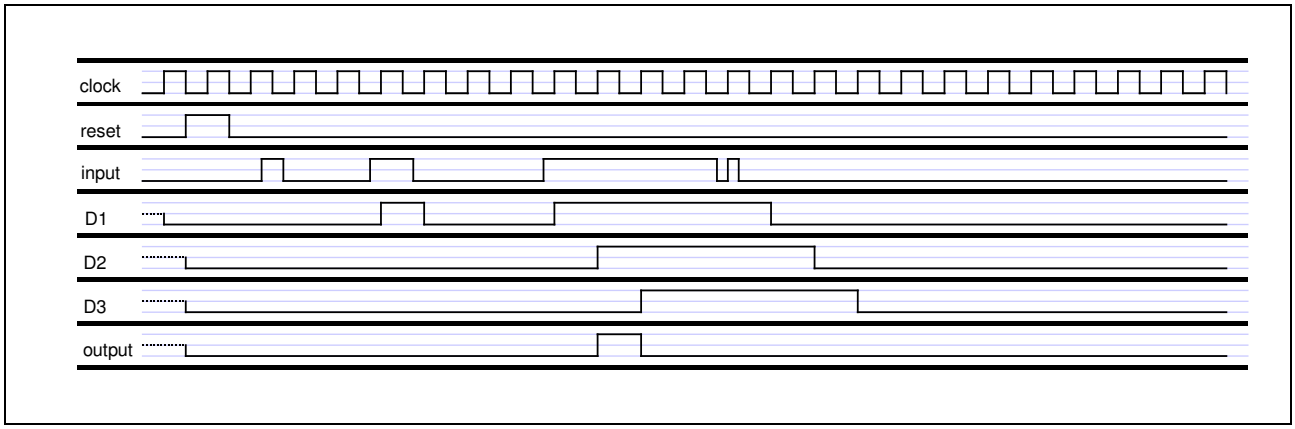

Figure 5: A simulation of the VHDL in listing 1.

In this simulation, the clock is generated by a simulation function defining the period. The other input signals; "reset" and "input", are defined by facilities within the simulation tool. The simulation tool allows the user to specify assertion times and duration for each of the different input signals. In this waveform, one can see that the reset signal has been asserted relatively early in the simulation to drive the D1, D2, D3, and output signals to a known state. Then several custom input bursts are defined at different points in the timeline to help verify the different expected responses from the debounce functionality.

After each VHDL function and sub-system has been simulated to a satisfactory level, the complete system can be synthesized. Synthesis is essentially a transformation that takes the abstract definitions (VHDL source, function libraries, and captured circuits) and combines them to create one giant 'netlist'. The netlist is a complete definition of the system from an interconnectivity perspective.

After a successful synthesis, the system must be transformed to the specific architecture of the FPGA device. The first step in the "implementation" stage is to map the netlist to the chosen architecture by breaking it up into CLB-sized pieces [3].

Configurable logic blocks (CLBs) are the fundamental build blocks at the heart of the FGPA. A CLB is like a junk drawer containing a variety of items that can be put together in different ways to create necessary circuitry. Typically, the CLB will contain lookup tables, multiplexers, latches, and glue logic. FPGAs have historically been measured by a relative 'gate' count. The gate count is meant to represent how many individual 'gates' it would take to create comparable circuitry to what any particular FPGA is capable of producing. However, FPGAs are also measured by how many CLBs they contain. The Xilinx Spartan IIE FPGA actually measures capacity with a benchmark called a 'slice'. It takes two slices to create one Spartan CLB. [4]. Figure 6 shows a graphical representation of a Spartan IIE CLB.

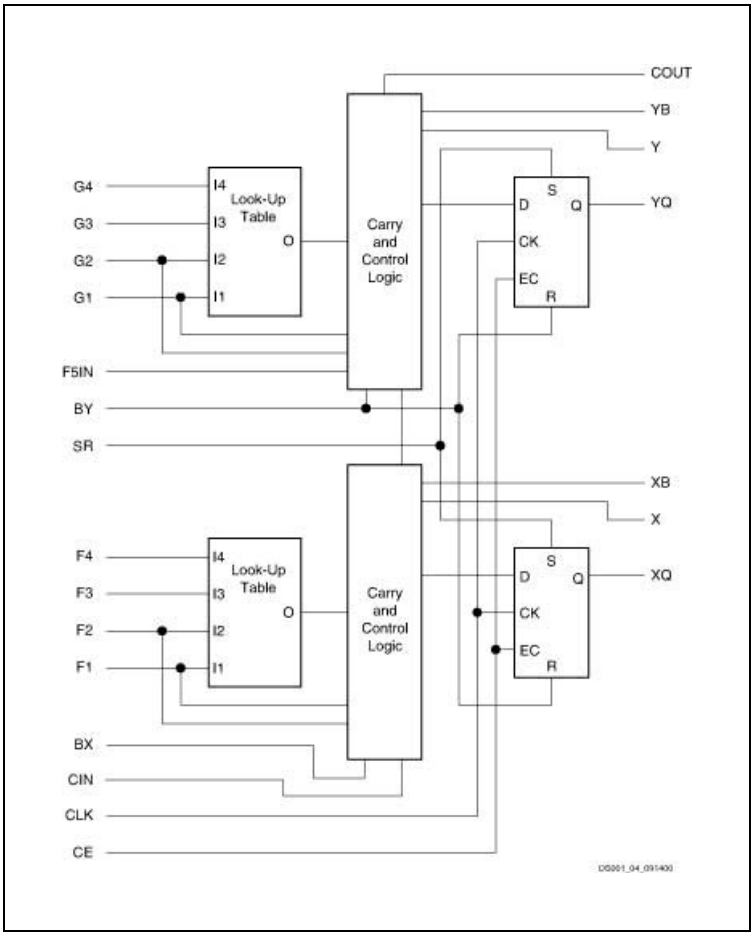

Figure 6: A Spartan IIE CLB [4]

The CLBs form the central logic structure of the FPGA. The CLBs in a FPGA are typically arranged on a square matrix, with each FPGA manufacturer having their own interconnection scheme. Figure 7 is a representation of the Spartan IIE FPGA Family. Like other FPGA brands, this specific chip has some block RAM built-in so that more powerful designs can be created. This chip also provides four Clock DLLs. DLL stands for delay-locked loop. The DLL is a circuit used to manage clock-distribution delay.

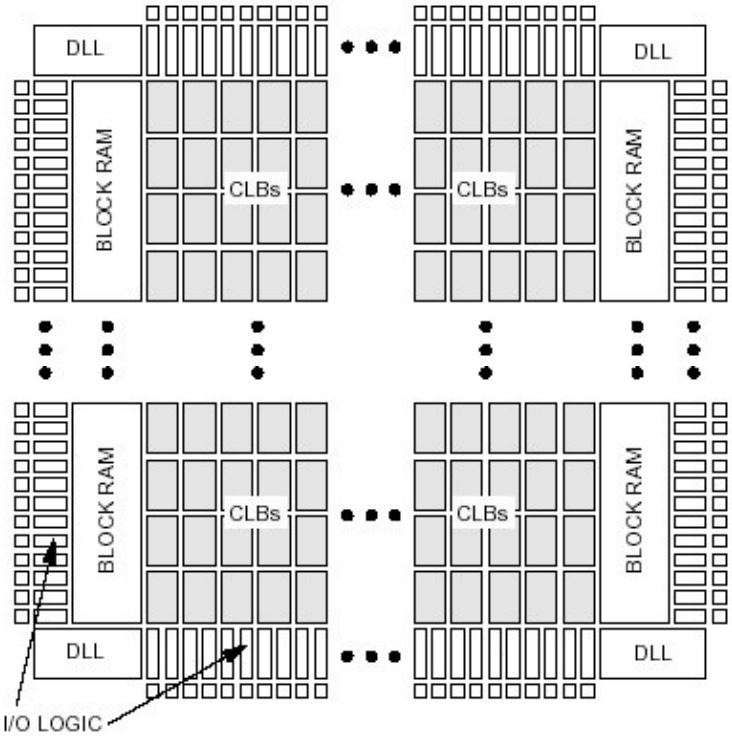

Figure 7: A partial Spartan FPGA Block Diagram [4]

Finally, note that the perimeter of the chip is surrounded by items identified as I/O logic. These blocks are called IOBs, or input/output blocks. IOBs provide the interface between the package pins and the internal logic. Figure 8 shows the contents of an individual IOB. Note that IOBs contain a fair amount of circuitry. This allows the IOB to do a significant amount of input pre-processing or output post-processing without wasting CLB resources. In addition, IOBs contain circuitry more germane to the types of tasks needed for signal conditioning (buffering, latching, biasing, etc).

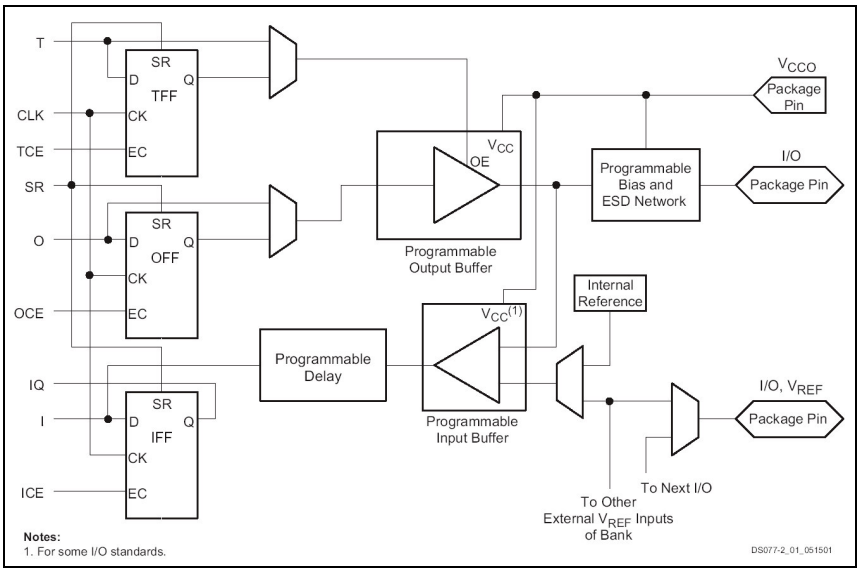

Figure 8: A Spartan IIE IOB [4]

After the netlist has been mapped to specific FPGA resources, the resources must be placed in the device and connected with other resources. This step is sometimes referred to as “place and route”. This step requires the tool to be rather intelligent so that the FPGA resources used are located as close as possible to

each other to reduce propagation delay. In addition, the place and route tool must also be cognizant of the constraints placed on the design in regards to the items mentioned earlier (separation of high speed / low speed logic, inputs tied to specific pins, area minimization, etc). Once the place and route tool has made an attempt at creating a minimal design, its output is simulated to determine if it has met its timing requirements.

In some development suites, the place and route tool can be run iteratively to help the tool optimize the design. Afterwards, the design can be visualized so the developer can see how the circuitry is actually organized in the target FPGA. Figure 9 shows the output of the Xilinx Floorplanner tool (part of the free Xilinx WebPack development suite – see the Appendix for more development resources.) In this figure it is easy to spot the three flip flops that are being used to implement the debounce VHDL. This is similar to the representative circuit shown in figure 4. Figure 10 shows the same debounce output with interconnects drawn.

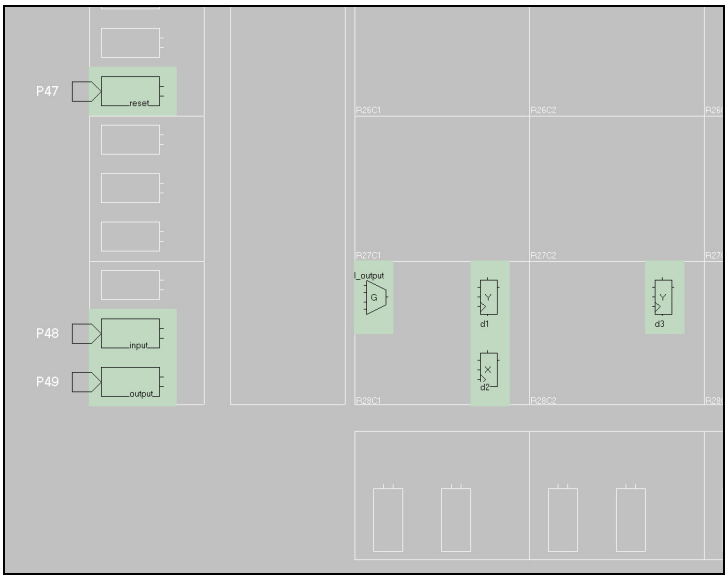

Figure 9: Placement of logic elements for the debounce VHDL

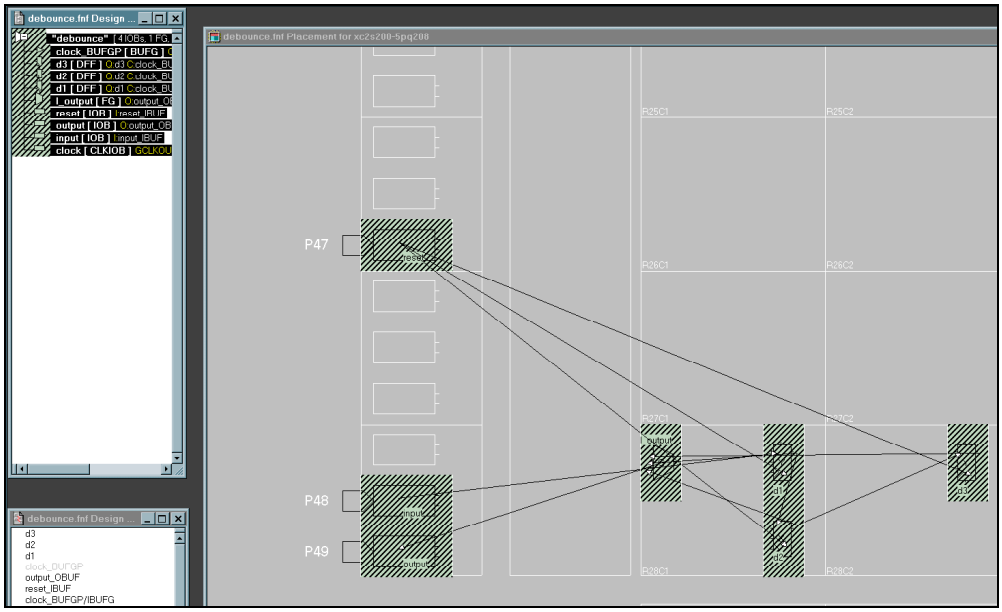

Figure 10: The debounce elements with interconnections shown.

The debounce example is nice for learning to understand the FPGA tool chain, but it doesn't really help to show off the floor planning functionality. Figure 11 shows the place and route output of another VHDL project. This project in figure 11 is a frequency counter based around a reconfigurable FORTH processor. The details of this project aren't important, it's being exploited to show the floor plan of a FPGA that is roughly 30% utilized.

As mentioned, the place and route tool performs timing analysis to ensure that constraints are met. This tool also looks for congestion in the interconnection of the device. Figure 12 shows what the floor-planning tool has identified as congestion for the frequency counter project show also in figure 11. It is interesting to note that the congestion is somewhat counter-intuitive in that it does not overlap where large majorities of CLBs are used in the FPGA. The cause of the congestion is the physical constraints, which identify that the pins in the upper right corner of the FPGA (as represented in these graphics) must be connected to a majority of the inputs and outputs for this project.

If this placement had caused problems (it didn't) the placement and/or routing would need to be tuned to provide a more desirable outcome.

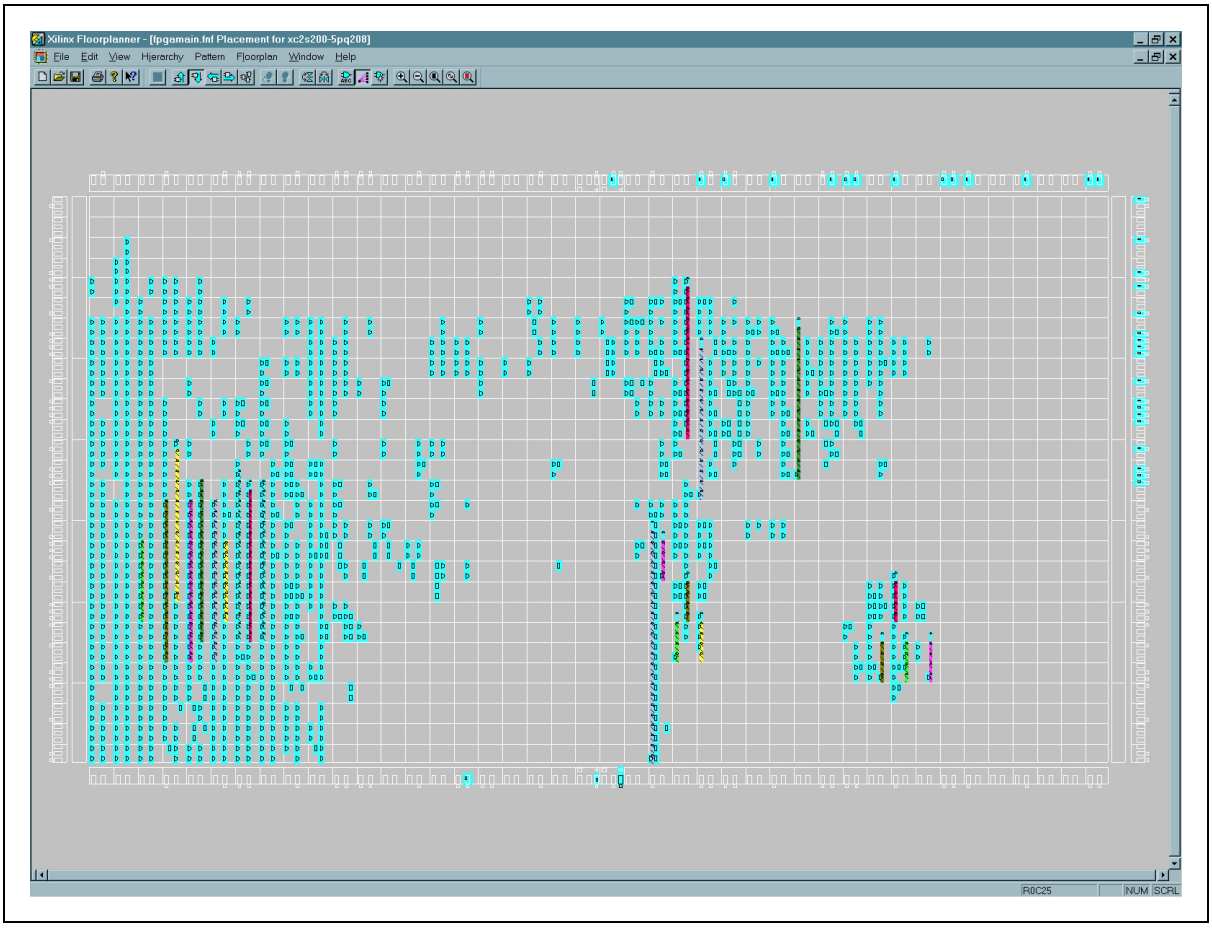

Figure 11: The logic placement of a frequency counter project

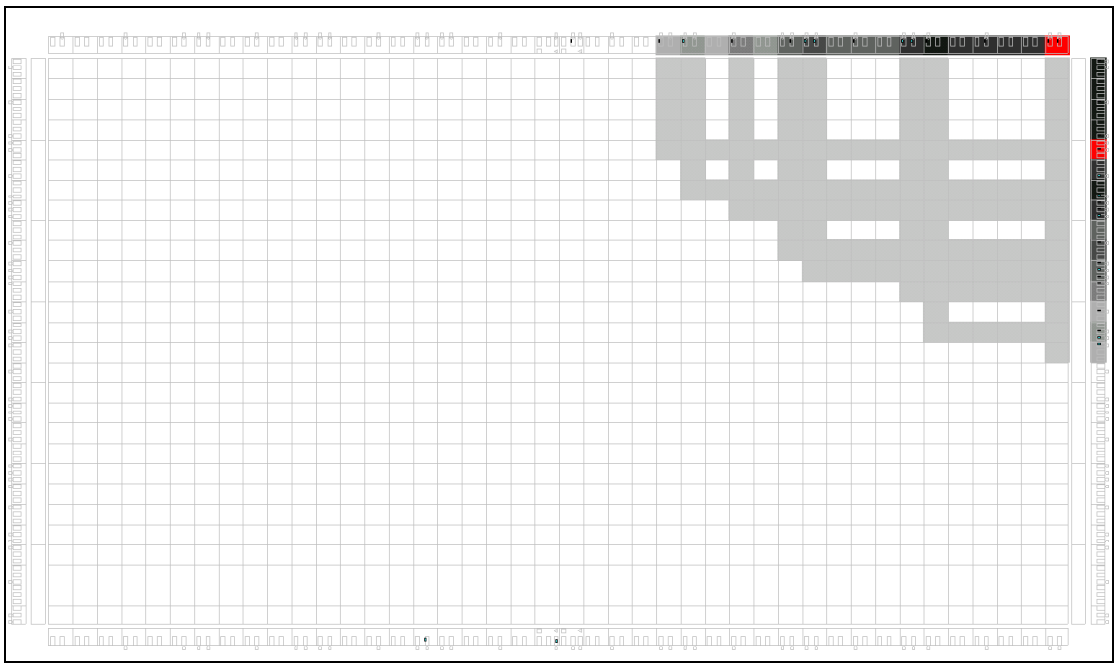

Figure 12: Congestion as identified by the Floorplanner tool.

Another useful function of the floor-planning tool is to create a pin output diagram that shows a view of the chip's footprint with the used pins highlighted in some way. Figure 13 is the pin view of a Spartan IIE FPGA as implemented with the frequency counter VHDL mentioned earlier.

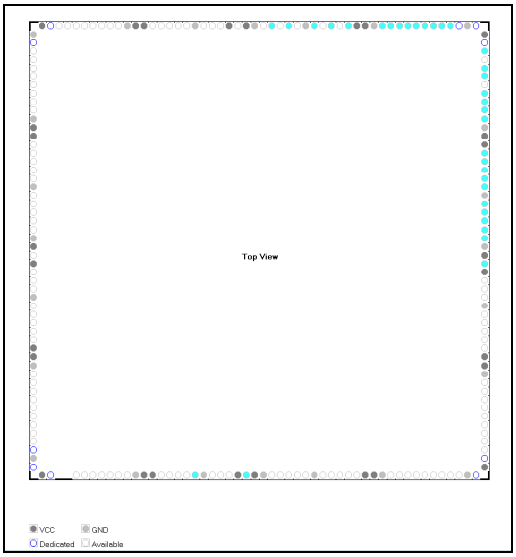

Figure 13: A Pin diagram for the FPGA with the frequency counter VHDL implemented

After the place and route effort has successfully completed, the detailed timing analysis can be used to provide a more accurate simulation of the project and its sub-systems. Recall that pieces of the VDHL system were simulated before even being synthesized. This simulation was very useful for understanding if the functionality would perform as designed, but it didn't take into consideration any propagation or timing delays that the physical layout would contribute. By using a technique call back-propagation, the timing analysis resulting from the actual placement of logic devices can be used to fine-tune the simulation tool. This is a highly recommended step as there could have been some VHDL which was functional in only the most favorable of time conditions, a condition which may be impractical or difficult to obtain without giving up some performance in another area.

Finally, after all of the synthesis, routing, and simulation task have been successfully concluded, the FPGA project must be output to a format that the FPGA can understand. The step is very similar to the loader task in software development. A bit stream generation tool takes the optimized logic placement output and creates a binary file which will be communicated (serially typically) to the FPGA for testing. After a project has passed all of the systems testing and verification that any typical embedded program would, it can be programmed to a PROM or some other persistent memory that can be used to initialize the FPGA every power cycle.

Wrapping it up…

This has been a relatively high level overview of a an FPGA development cycle. The intent was to frame the discussion in a way that the development steps would make sense to a embedded systems engineer who may have more experience in the traditional software development method of creating solutions. This overview was fairly Xilinx-centric, but the tools, tasks, and FPGA concepts are applicable to other vendors and manufacturers. This paper didn't (intentionally) get into the specific nuances or syntax of VHDL. As such, the paper should be just as useful to those who are using an HDL other that VHDL (i.e. Verilog, System-C, Handel-C, etc.)

This paper was written specifically for those who are interested in expanding their skill set – for engineers who are trying to stay abreast of technology while keeping their day jobs. FPGA evaluation boards are readily available from a variety of vendors. There is also a wealth of information on the Internet, including many free and inexpensive tools. It is the hope of the author that this paper can serve as an impetus to exploit these other great learning sources.

Appendix A: Bibliography

[1] Armstrong, James R, and Grey, F. Gail, VHDL Design Representation and Synthesis, Second Edition, Prentice Hall PTR, 2000.

[2] "New Spartan-IIE FPGA Family ", Ashok Chotai, Xcell Journal Online, 11/19/2001, http://www.xilinx.com/publications/products/sp2e/sp2e_cons.htm

[3] "FPGA Terminology", Sprow's Webpages, http://homepage.ntlworld.com/rpsprowson/fpgas/lingo.htm

[4] "Spartan-IIE 1.8V FPGA Family: Functional Description", Xilinx Corp, 2001, Document DS077-2(v1.0)

[5] Haskell, R.E., "Xilinx CPLDs and FPGAs: Module F2-1", CSE-670, Oakland University, Rochester, MI.

Appendix B: Other Resources

"Mentor Graphics FPGA Methodology and Tools Overview", Kevin Morris, Mentor Graphics, 2003

Xilinx Free Development tool: ISE WebPack:
http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Design+Tools

comp.lang.vhdl, FAQ, Section 3: VHDL on the Web,
http://www.vhdl.org/comp.lang.vhdl/FAQ1.html

Open IP Cores and Tools: http://opencores.org