Programmable logic: A practical introduction for beginners

Michael Dunn - November 19, 2013

Only one sixth of the respondents to a poll we ran last year on Scope Junction reported HDL (hardware definition language, i.e., programmable logic) experience. Thus was this article inspired. If your exposure to this ubiquitous technology has been nil to minimal, or you just want to brush up on the basics and history, read on.

A bit of history
The first practical programmable logic chips, known as PLAs (programmable logic arrays), became available in the mid 1970s. Things really took off though when MMI introduced a simplified series of PLAs they called PALs. Using a basic HDL like PALASM, one could create simple functions, such as address decoders or control logic. Performance was often greater than discrete logic, because many functions that would otherwise need two or more levels of discrete logic could now be accomplished with just one "pass" through the PAL.

Be my PAL
The PAL family continued to evolve into the late 1980s, with faster, lower-power devices, including ones that could be reprogrammed. The most complex device type developed was probably the PAL22V10, which could implement higher-level functions like state machines, and had a more generalized logic array structure than the earlier PALs.

The smaller PALs only allowed for simple AND-OR-based logic, whereas a part like the 22V10 contained registers, tristatable outputs, and so on.

The CPLD
CPLDs (complex programmable logic devices) were the next evolutionary step. These tended to be register-oriented, and sometimes even contained "buried" registers – ones not directly associated with an output pin. Output "macrocells" became relatively sophisticated, and might contain a register, a bunch of configuration and control logic, an XOR gate, and so on.

Because of the increased size of these devices, the logic array tended to get broken into segments. Without this, the AND/OR array would have become unwieldy and slow. Internal signals could travel between these segments, though with some limitations, and also some restrictions on functional change once the pinout was fixed.

An Intel 80486 microprocessor board I designed ca. 1996 used several 64-register CPLDs. One chip would be able to hold several state machines and a significant amount of miscellaneous logic.

Actually, I can stop talking in the past tense, as CPLDs like this are still alive and kicking. They find use whenever an FPGA would be overkill. In fact, a medium-sized CPLD can contain my base-2 log
Some parts are actually FPGAs in CPLD clothing. I think this practice is perpetrated by companies to differentiate their RAM-configured FPGAs from their non-volatile "CPLDs."

The FPGA
Xilinx released the FPGA (field-programmable gate array) into the wild ca. 1985, and 25 years later, the FPGA has become virtually synonymous with "programmable logic." The guts of an FPGA is of the "sea of LUTs (lookup tables)" variety. Each of these LUT cells typically contains a flip flop, and yes, a LUT, along with lots of muxes and configuration logic. The LUTs are typically 4- or 5-input, which allows a pretty wide variety of logical behaviors to be described.

These LUTs tend to be arranged in some sort of hierarchy - an arrangement that allows for practical handling of signal routing. Fortunately, we rarely need concern ourselves with such details. That's the design software's problem.

FPGAs these days often include other features, such as PLLs and clock management blocks, high-speed serial transceivers (the range today being about 3-30Gb/s), RAM (a large chip might have megabytes), DSP blocks (hundreds or even thousands of multipliers are not unheard of!), and PCIe interfaces.

There are also "mixed-domain" FPGAs. Several manufacturers have parts with one or two microprocessors on-chip, analog peripherals, or both.

Not to fear if your FPGA has no processor. You can instantiate as many "soft-core" processors as will fit in the programmable logic. Even a mid-range FPGA can hold dozens of 32-bit processors!

Making the logic dance to your tune

Once again, a bit of history
HDLs (hardware description languages) were developed, not to create circuits, but to document and describe them! The modern languages Verilog and VHDL got their start this way in the mid-1980s. But, FPGAs were also introduced around this time, and it didn't take long for these HDLs to assume a design role.

What's an HDL?
HDLs provide a way to design digital logic without using schematics. "Why do that?" you ask? As you will discover if you delve into the topic even a bit, HDL code can be more compact, more portable, and, if built well, more understandable than a schematic. Picture a relatively simple design—say, a couple dozen counters, muxes, decoders, comparators, and gates. Would you prefer pages of schematics filled with rectangles and gates, or a page of elegantly written (one hopes) HDL?

What if you wanted to replicate your circuit 16 times? You could add a few lines of HDL. Or many, many schematic pages. Sure, a hierarchical schematic would reduce the work. But what if each instance of your circuit was a bit different? Again, a line of HDL would likely suffice. What if you wanted to double the width of several buses? You tell me.

Don't get me wrong. I know a clearly drawn schematic can bring a bit of joy to a techie's heart. But there comes a point where one has to admit its limitations and look for a better way.
Examples, please
OK, you're convinced. What might such code look like? See if you can grok this:

- if sel = 1
  - out = a + b
else
  - out = b + c
endif

  if clk
    - a += 1
  endif

d = 10 < a < 80

e = a xor c

(Note that this is pseudocode – not a real HDL – but close enough.)

I'd be willing to wager that if, you have experience with software, but not with HDL, you'll misinterpret the above! What's the first rule of Fight Cl... uh, HDL? It's not sequential. What's the second rule? It's not sequential. Each of those lines or blocks defines a piece of logic. They could be written in any order, and the meaning would not change.

The first block defines a mux and two adders (though some EDA software would recognize that muxing a and c and adding that to b would save logic. If you don't want that to happen, your code may need to be more explicit). The second block is a counter, continually incrementing at the clk rate. Next, we have a pair of ANDed comparators. Finally, we have an XOR of counter a and input c.

Languages

Which language?
You may now be wondering which HDL to study. As with software, this question is more likely to provoke a religious war than result in any useful answers. The best answer I've heard to the question: "What's your favourite language?" is "Whichever one I'm not using at the moment."

Both VHDL and Verilog are worth knowing. Verilog is being superseded by System Verilog, which has better support for testing code, and cleans up a few rough spots in the language. Otherwise, they're essentially the same.

If you want to work at a higher level, most FPGA software supports schematic entry, though like I explained above, I really don't recommend this for anything but the simplest little designs and
experiments. Let's call schematic entry a "different" level, not a "higher" one.

A true higher-level approach is to convert C code into FPGA hardware, but obviously, it must be written according to detailed rules. A related idea is to convert the most time-critical parts of a C program into HDL. (As an example, Altera has a couple of devkit demos which navigate the Mandelbrot Set. One version runs purely on their softcore processor. The other version has some critical code replaced by a custom processor instruction which invokes dedicated hardware that was created from the replaced C code. This version runs hundreds of times faster.)

Some FPGA software lets you patch together a system using a drag-and-drop type of front-end, but that method is more applicable to processor systems and datapaths, not general logic. Still, the ability to create a custom 32-bit microcontroller in five minutes can be handy!

Bluespec claims its HDL operates at a higher level than the traditional languages, making designers more productive. Creations in LabView and MATLAB can also be translated into FPGA code.

You will find lots of educational websites with a bit of searching. Here are a few to get started with:

- FPGA & VHDL course by The Hamster.
- OpenCores, an open-source repository of HDL code.
- Verilog Tutorial
- Introduction to Verilog
- SystemVerilog Tutorials

Don't have an FPGA devboard yet? (though some under $100 can be found) Download at least one software package to play with anyway! Even the free versions are quite capable, and you'll be able to try out your code using the built-in simulator. If you do have an actual FPGA board, you'll be able to "scope" internal signals virtually via the JTAG interface.

- Altera's Quartus software is my recommendation for a first download.
- Xilinx design software
- Actel/Microsemi's Libero suite is worth a look, especially if you're intrigued by their low-power or non-volatile parts, or their SmartFusion chips, which combine FPGA, analog componentry, and an ARM Cortex-M3 (which is also similar to Cypress' PSoC).

I hope you found this short intro to programmable logic of value. Comments, questions, and reports
of your experiences are welcome!

Also see:

- [FPGA Gurus blog](#)
- [Why Matlab Can Come in Handy](#)
- [LabView FPGA becomes tool for system configuration](#)
- [Lattice and iCE40 USB development stick—practically giving it away](#)
- [How FPGAs and multicore CPUs are changing embedded design](#)
- [An introduction to offloading CPUs to FPGAs: Hardware programming for software developers](#)
- [CoolRunner-II Retro-pong](#)
- [Implement a stepper-motor driver in a CPLD](#)

Follow Michael Dunn at 🌐 Twitter 🎥 Facebook