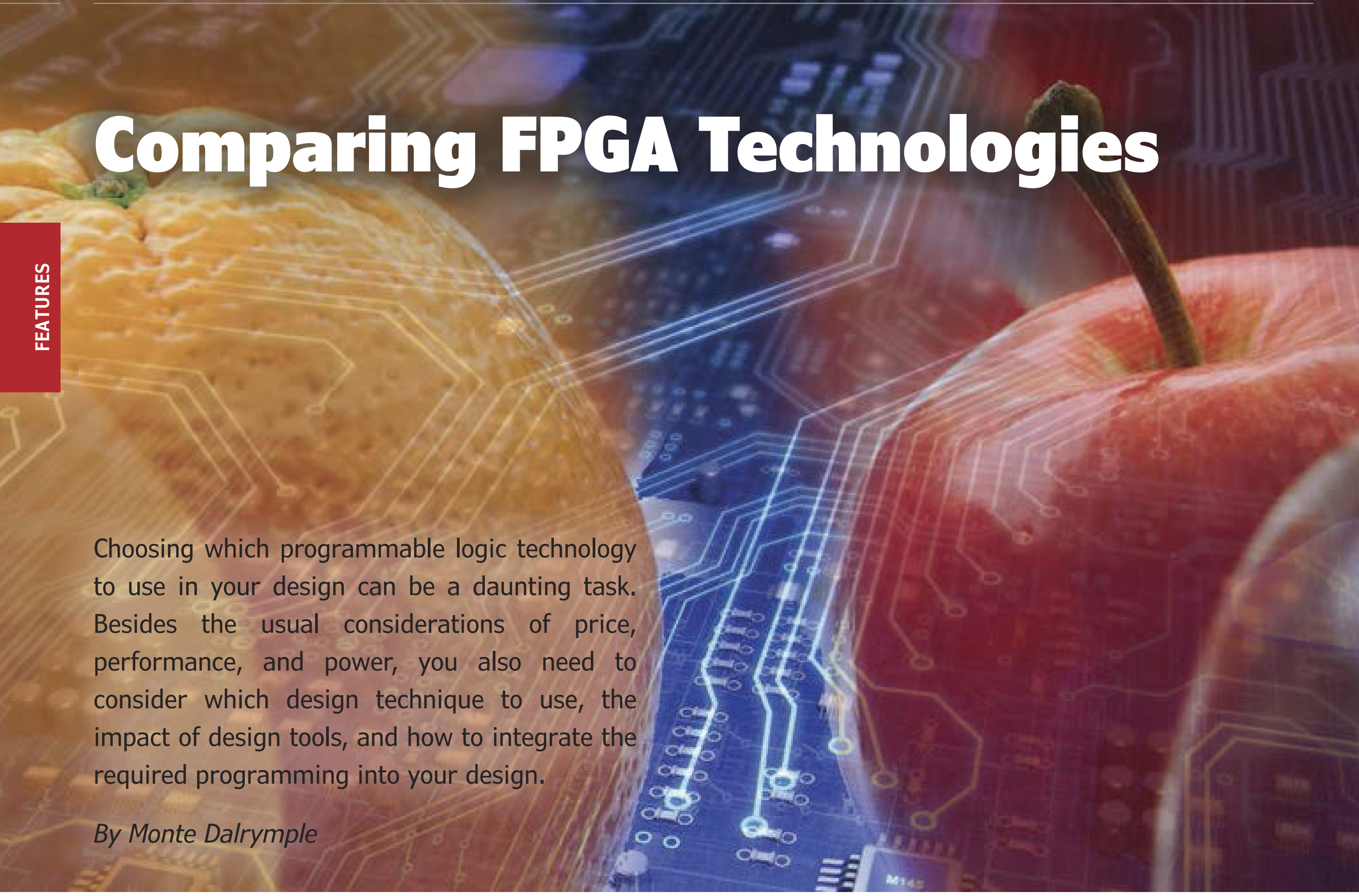

Comparing FPGA Technologies

Choosing which programmable logic technology to use in your design can be a daunting task. Besides the usual considerations of price, performance, and power, you also need to consider which design technique to use, the impact of design tools, and how to integrate the required programming into your design.

By Monte Dalrymple

Programmable logic, in the form of the field-programmable gate array (FPGA), provides an incredible amount of design flexibility. But being able to take advantage of all of this design flexibility requires a steep learning curve. Much of this learning curve is independent of the specific FPGA technology you choose, but there are still a significant number of vendor-specific and even device-specific features that you need to be aware of.

It would take a book, or several books, to cover all of this material, so in this article I'm only going to provide an overview of some of the things that I have encountered when choosing an FPGA. Then I'll take several real-world designs through the implementation process with different FPGA families to give you an idea of how these families compare.

FPGAs range in size from the equivalent of a few thousand gates to flagship devices containing billions of transistors and complete multicore 32-bit processors. I'm going to assume that the majority of readers are interested in something fairly small and limit my discussion to those kinds of devices.

FIRST THINGS FIRST

It isn't really feasible to use anything other than a hardware description language (HDL) to design for an FPGA, so your first decision is between Verilog and VHDL. All FPGA tools are happy with either choice, but I use Verilog exclusively.

Your second decision is whether or not you want your design to be technology-agnostic or whether you are okay with being locked to a particular vendor. The design suites supplied by every FPGA vendor make it easy to use the special features in their devices, and taking advantage of these special features often makes a lot of sense. Just be aware that the further you go down this path, the harder it will be to change course in the future. I'll discuss an example of one way to manage a common case later in this article.

There are four primary suppliers of FPGAs: Altera, Lattice, Microsemi, and Xilinx. Altera and Xilinx are the big guys, while Lattice and Microsemi are smaller, but with some significant technical advantages in certain applications. Here is a quick overview of each of them.

Altera supplies four FPGA families. Three of the families require an external source for their configuration information, and one family has on-chip flash memory used to load the configuration. The Cyclone IV family is targeted at cost-sensitive applications, so that's the one I'll be concentrating on here.

Lattice offers three main FPGA families. One family requires and external configuration source, while the other two use on-chip flash

```
  /*********************************************************************/
  /* fpga-independent 128x8 RAM, write-before-read                     */
  /*********************************************************************/
  assign addr_mtch  = ~|(ram_raddr ^ ram_waddr);

`ifdef SIM_VERSION

  always @ (posedge clk or negedge resetb) begin
    if     (!resetb) ram_data <= 8'h00;
    else if (ram_rd) ram_data <= (ram_wr && addr_mtch) ? ram_wdata :
                                 ram_mem[ram_raddr];
    end

  always @ (posedge clk) begin
    if (ram_wr) ram_mem[ram_waddr] <= ram_wdata;
    end

`else  // !SIM_VERSION

  always @ (posedge clk or negedge resetb) begin
    if (!resetb) begin
      ram_byp  <= 1'b0;
      ram_ddly <= 8'h00;
      end
    else begin
      ram_byp  <= ram_wr && addr_mtch;
      ram_ddly <= ram_wdata;
      end
    end

  assign ram_data <= (ram_byp) ? ram_ddly : ram_rdata;

`ifdef CYC4_VERSION

  A_128x8 RAM0 ( .clock(clk), .data(ram_wdata), .rdaddress(ram_raddr),
                 .rden(ram_rd), .wraddress(ram_waddr), .wren(ram_wr),
                 .q(ram_rdata) );
`endif  // CYC4_VERSION
`ifdef ICE40_VERSION

  L_128x8 RAM0 ( .ram_rdata(ram_rdata), .clk(clk), .ram_rd(ram_rd),
                 .ram_raddr(ram_raddr), .ram_waddr(ram_waddr),
                 .ram_wdata(ram_wdata), .ram_wr(ram_wr) );

`endif  // ICE40_VERSION
`ifdef A3P_VERSION

  M_128x8 RAM0 ( .WD(ram_wdata), .RD(ram_rdata), .WEN(ram_wr),
                 .REN(ram_rd), .WADDR(ram_waddr), .RADDR(ram_raddr),
                 .RWCLK(clk) );
`endif  // A3P_VERSION
`ifdef XC3S_VERSION

  X_128x8 RAM0 ( .clka(clk), .ena(ram_wr), .wea(ram_wr),
                 .addra(ram_waddr), .dina(ram_wdata), .clkb(clk),
                 .enb(ram_rd), .addrb(ram_raddr), .doutb(ram_rdata) );

`endif  // XC3S_VERSION
`endif  // !SIM_VERSION
```

LISTING 1
Implementing a vendor-independent block RAM isn't too difficult. Newer compilers provide for multi-way `` `ifdef `` directives, but I always stick with the simpler two-way directives for compatibility reasons.

TABLE 1
Four different vendors and four different design suites can make moving to a different FPGA family painful.

	Altera	Lattice	Microsemi	Xilinx
Design Suite	Quartus II	iCEube2	Libero SOC	ISE
Version	15	2014.12	11.5	14.7
Size	5.9 GB	6.2 GB	3.3 GB	16.7 GB
Logic Sim	none	ActiveHDL	ModelSim	ISIM
Logic Synthesis	Quartus II IS	Synplify Pro or Lattice LSE	Synplify Pro	Xilinx XST

TABLE 2
Block RAMs in FPGAs are incredibly useful, but again, every vendor has a different implementation.

	Altera	Lattice	Microsemi	Xilinx
Bits per RAM	9216	4096	4608	18432
Native size	256 × 36	256 × 16	256 × 18	512 × 36
Alternate sizes	256 × 32	512 × 8	512 × 9	512 × 32
	512 × 18	1k × 4	1k × 4	1k × 18
	512 × 16	2k × 2	2k × 2	1k × 16
	1k × 9		4k × 1	2k × 9
	1k × 8			2k × 8
	2k × 4			4k × 4
	4k × 2			8k × 2
	8k × 1			16k × 1
Available modes	1-port	1-port	1-port	1-port
	2-port	2-port	2-port	2-port
	Dual-port	ROM	Dual-port	Dual-port
	ROM		ROM	ROM
	FIFO		FIFO	FIFO
Special features	Byte enables	Bit mask		Byte enables

memory to hold their configuration. The flash-based iCE40 family is unique among FPGAs in that it has very low quiescent current, and this is the family I'll be using from Lattice.

Microsemi supplies four FPGA families. All four families use distributed flash to store the configuration information, which means none of them require any time to load the configuration at start-up. The ProASIC3 family is the low-power, low-cost family that I will use here.

Xilinx was the first FPGA supplier, and also offers four main FPGA families. All four families require an external source for their configuration information, but one variation within the Spartan-3 family includes a configuration flash memory in the same package and that variant is what I will be using here.

Although all four FPGA suppliers have settled on three or four main families, they all have multiple variants within each family. This can make choosing a device even more complicated, and your choice will depend on the specific requirements of your application. For what follows here, I started with a fairly arbitrary set of constraints: the devices had to be regularly stocked at Digi-Key with a single-unit price under $25 and have at least 150 I/Os. If you're shocked at that price, consider that a flagship device from Altera or Xilinx might go for over $2,000. The design flexibility of an FPGA does not come without a price.

DESIGN SUITES

An overview of the tools I'll be using is shown in Table 1. All of the design suites shown here are free, which often means that they don't support the full range of FPGA families or even all of the sizes available within families. This can create a chicken-and-egg situation if you're not sure how large an FPGA you're going to need, so keep that in mind when you're first starting out

One potential pitfall when downloading

ABOUT THE AUTHOR

Monte Dalrymple (monted@systemyde.com) has been designing integrated circuits for over 35 years. He earned a BSEE and MSEE from the University of California at Berkeley and holds 17 patents. He is the author of the Circuit Cellar-published book Microprocessor Design Using Verilog HDL. Not limited to things digital, he holds both amateur and commercial radio licenses.

	Altera	Lattice	Microsemi	Xilinx
Part number	EP4CE6F17C8N	ICE40LP4K-CM225	A3P250-FGG256	XC3S200AN-4FTG256C
Package	256-BGA	225-BGA	256-BGA	256-BGA
Package size	17 × 17	7 × 7	17 × 17	17 × 17
PCB pitch (mm)	0.8	0.4	0.8	0.8
Unit price	14.94	8.38	24.80	23.10
Logic Elements	6272	3520	6144	3584
Block RAMs	30	20	8	16
Available I/O	179	167	157	195
Total RAM bits	270K	80K	36K	288K
Core voltage	1.2 V	1.2 V	1.5 V	1.2 V

TABLE 3
These are the devices I'll be using for the comparisons. I wanted to avoid BGA packages, but it just wasn't possible given my I/O requirements.

a design suite is that both Lattice and Xilinx have two different design suites available. In the case of Lattice, it's because the iCE40 family was acquired rather than developed internally. For Xilinx, the newer FPGA families use a new design suite that was built from scratch rather than adding to the existing set of tools. Since I know that I'll be targeting the Spartan-3AN family, I'll use the older set of tools.

All of these design suites have version numbers in the teens. This is primarily due to the length of time that FPGAs have been around, but it also highlights something else that you need be careful about. I never upgrade to a major new version of tools in the middle of a project unless the upgrade fixes something that directly affects me. These tools are so complex that a major revision inevitably introduces new issues, which can sometimes break a working project. For the same reason, when you finish a project always archive the version of the design suite you used along with the design, because there is no guarantee that a new version of the tools will be able to recreate the same programming for the FPGA.

The size of these design suites varies over a range of five to one, and this is primarily a function of the number of devices that the tools support. The Lattice and Microsemi design suites support a relatively small number of devices, while Altera lets you choose which devices to include in the download, which directly affects the size, and Xilinx automatically includes support for every FPGA allowed. The Altera approach seems to presuppose that you know which family you intend to use before you download the tools.

You will always need to simulate your FPGA design, and three of these design suites include a logic simulation tool which also supports vendor-specific features. Altera does not bundle a simulator with the free version of their design suite, but they are happy to sell you one. If you have written your HDL code properly, these three bundled simulators should always give the same results, so I can't say that one is better than another. But if you use HDL constructs that are outside the normal synthesis-friendly subset, which happens occasionally in a test bench, your results might vary depending on the simulator. Of course, you are always free to use a stand-alone simulation tool with any of these design suites, which is what I do for my FPGA designs.

Every design suite has to include a logic synthesis tool, with four different tools between the four vendors. Three of the tools are proprietary, with only Lattice and Microsemi offering a third-party tool. You can use a third-party synthesis tool with either Altera or Xilinx, but such tools are not free.

All four design suites provide the ability to implement a design with a single click, assuming that the various default options are okay. The default options are usually a good place to start when first attempting to implement a design, but I often find myself needing to change some of these defaults before I am finished with a project. This seems to be particularly true when dealing with logic synthesis for Lattice and Microsemi.

Whether you choose the one-click option or run each of the tools individually, it is imperative that you take the time to review all of the log files. There are almost always warnings or notes buried in these files that can be used to improve the design. A "turn and burn" approach can be used if your project is small enough, but I don't recommend this unless you're only making an incremental change to an existing design.

TECHNOLOGY INDEPENDENCE

As I mentioned previously, making your design independent of an FPGA technology can be a challenge. To illustrate this challenge let's look at the options for the internal memories among the different FPGA families. These embedded RAMs are incredibly useful in FPGA designs. They can be used as regular RAM, dual-port RAM, FIFOs, or ROM, often with different aspect ratios on different ports. But as Table 2 shows, every FPGA vendor has its own idea about these embedded RAMs should be implemented.

All FPGA embedded RAMs are fully synchronous, with separate read and write ports. Separate clocks can be used for these two ports, although the RAMs are not always true dual-port, so if you need to do simultaneous reads and writes of the same location, read the datasheet very carefully.

All of the suppliers except Lattice supply a memory wizard to automatically generate a Verilog module with the desired size and attributes. Depending on the underlying capabilities of the memory, this module may include logic in the FPGA fabric and multiple embedded RAMs. Unfortunately, everyone uses a different naming convention for the interface, so when I need a RAM in a design, I use Verilog `` `ifdef `` compiler directives to select between technologies, including a generic option for simulation, as shown in Listing 1. The listing shows a separate module for Lattice, but it needs to be generated manually.

This listing is an excerpt from the register file for a CPU that requires that the new data be returned in the case of a simultaneous read and write of the same location. The different embedded RAMs don't operate identically in this case, so it's easier to handle this special case in external logic common to all.

I always use the memory wizard where possible, because each supplier has a different method for selecting among the various options. Altera uses a synthesizable Verilog module, with all of the options controlled by parameters set up by the memory tool. Xilinx uses a behavioral module with options controlled by inputs. Microsemi uses a synthesizable module with options controlled by inputs. You can see why a wizard makes things easier.

Using an embedded RAM as a ROM can be easy, or complex, depending on the supplier. The Altera memory wizard accepts an Intel Hex format initialization file, and includes this data in the programming bit stream. But if you need to change the memory contents, you need to recompile the entire design. Lattice documents how to arrange the data to pass to the synthesis tool but it's an odd format that will require a program (I use Perl) to create from Intel Hex. Again, you need to recompile the design if you change the code. Microsemi embedded RAMs can only be loaded with ROM data via the JTAG port or from the FPGA fabric, which can only be done with the device installed in the final system. In most cases this is not very practical. Xilinx allows an initialization file to be specified, but in a proprietary format, requiring yet another formatting program. But Xilinx does offer a utility that can modify an existing bit stream if the code changes. Yet another example of how everyone does things differently.

All FPGA suppliers document how they have organized the programmable fabric in their devices, along with the different types of interconnect available, and it's interesting to compare the different choices they have made. But unless you are doing a very high-performance design or want to squeeze every last bit of functionality into the FPGA, this isn't something that will matter much in your choice of which FPGA to use. As long as there are enough look-up tables (LUTs) and flip-flops to implement the design, the details of how the LUTs and flip-flops are organized inside the FPGA fabric isn't particularly important to know. One exception is Microsemi, because unlike everyone else their basic logic element can only be used as either a LUT or a flip-flop, but not both simultaneously.

	Altera	Lattice	Microsemi	Xilinx
Available Logic Elements	6272	3520	6144	7168
Required Logic Elements	2314	3262	4866	3492
Utilization	37%	93%	79%	48%
Flip-flops	405	504	424	412
Frequency (MHz)	53.2	31	33.3	41.3

TABLE 4
The Y80 microprocessor fits in each chosen FPGA (except for Xilinx), but with wildly different results for both size and frequency.

	Altera	Lattice	Microsemi	Xilinx
Available Logic Elements	6272	3520	9216	3584
Required Logic Elements	1560	2507	6963	1955
Utilization	25%	71%	76%	53%
Flip-flops	772	869	1181	803
Frequency (MHz)	75.3	42.7	37.9	61.9

TABLE 5
The SHA-256 core is smaller, except in the case of Microsemi. The Microsemi synthesis tool doesn't recognize a pair of FIFOs in the Verilog, which leads to the large logic element requirement.

COMPARING THE IMPLEMENTATIONS

Table 3 shows the devices I will be using for my comparisons. I chose these devices, in terms of logic elements, to be large enough

	Altera	Lattice	Microsemi	Xilinx
Available Logic Elements	6272	3520	6144	3584
Required Logic Elements	2301	2893	4242	2345
Utilization	37%	82%	69%	65%
Flip-flops	653	687	660	672
Frequency (MHz)	65.7	43.3	55.9	60.3

TABLE 6
The Y51 microcontroller is a two-clock-per-instruction implementation of the 8051 instruction set, with a full complement of peripherals. It fits nicely in each FPGA and uses on-chip memory for the register file.

to fit most of the designs that I will be using here. I should note that I would have used a smaller Altera device had one been available, but this is the smallest member of the Cyclone IV family. I originally wanted to try to use an identical package for all four FPGA technologies, but this was not feasible because Lattice only uses high-density BGA packages.

My I/O requirement is difficult to meet without using a BGA package, but for everyone except Lattice there are other package options with less I/O, so keep that in mind when starting your own FPGA design. Don't forget that Altera requires an external serial Flash memory to hold the configuration, and Lattice may require one, if you are not comfortable with the one-time-programmable nature of the on-chip configuration memory. The Microsemi and Xilinx configuration memory can be reprogrammed, so they don't need an external memory that adds to the total system cost.

To make these comparisons as close as possible I tried to use similar synthesis options across all four FPGA technologies. I disabled the finite state machine (FSM) optimization so that all of the state machines were identical, but I did allow resource sharing, because that usually lets the synthesis tool take better advantage of the underlying FPGA capabilities. With everything else I used the default options, and I did not constrain either the clock frequency or the I/O placement.

The first design I use is the microprocessor that I covered in my book, Microprocessor Design Using Verilog HDL (Circuit Cellar, 2012). The Y80 microprocessor design implements the Z80 instruction set, with a two-clock machine cycle, and a bus interface that is compatible with FPGA embedded memories. The results for this design are shown in Table 4.

Note that this design doesn't quite fit in my target Xilinx device, so I had to use the next larger device in the family. The XCS400AN is pin-compatible with my target device, but doubles the number of logic elements. Unfortunately this device is not normally stocked at Digi-Key and costs $26.46 with a minimum order quantity of fourteen. These are the kinds of things you run into designing with FPGAs.

While this design may not be the best fit for an FPGA, because of the ratio of logic to flip-flops, the results show that all of the devices do a good job here. It's interesting that the maximum frequency varies over almost a two-to-one range between the different devices, which probably has more to do with the chip fabrication technology than anything else. The different number of flip-flops in each implementation is due to the synthesis tools replicating flip-flops to reduce the loading on any individual flip-flop.

In the Y80 design I use one-hot encoding for the main state machine, and because of how I use this state machine in the design, there are synthesis directives embedded in the Verilog code. Lattice, Microsemi and Xilinx all place warnings about these directives in the synthesis log as they should, but the Altera synthesis tool appears not to understand these directives and generates over two hundred spurious warnings as a result. Spurious warnings like this make finding useful information in log files more difficult, but I still recommend going through the log files line by line.

The second design implements the 256-bit Secure Hash Algorithm (SHA-256). For this design the ratio of logic to flip-flops takes better advantage of the structure of an FPGA, and I use embedded memory as the FIFO required for the input data. The results for this design are shown in Table 5. (Visit the *Circuit Cellar* FTP site to download the Verilog source code for the Y80 microprocessor or SHA-256 core.)

This design doesn't quite fit in my target Microsemi device, so I had to use the next larger device in the family. The A3P400 is pin-compatible with my target device, but increases the number of logic elements by fifty percent. Unfortunately this device costs $39.04 for a single unit.

I originally planned to use the different memory wizards to automatically implement the input buffer for this design, but decided that it was better to design the control logic

circuitcellar.com/ccmaterials

SOURCES

EP4CE6F17C8N
Altera Corp. | www.altera.com

ICE40LP4K-CM225
Lattice Semiconductor | www.latticesemi.com

A3P250-FGG256
Microsemi Corp. | www.microsemi.com

XC3S200AN-4FTG256C
Xilinx, Inc. | www. Xilinx.com

myself and use a plain embedded RAM. One problem is that Lattice doesn't supply built-in FIFO operation for the embedded RAM, so I need to design a FIFO controller for at least one case anyway. The second problem is that the Xilinx FIFO function does not directly support an arbitrary level for the `almost_empty` signal that I use to hold off the start of the hash computation. But the most compelling reason is that a common FIFO controller guarantees that the timing will be identical across all FPGA technologies.

You're probably wondering about the large number of logic elements used for this design in the Microsemi device. There are two reasons why the number is so much smaller for the other suppliers. First, the hash algorithm uses a pair of thirty-two bit wide FIFOs. One FIFO is five words deep and the other is eight words deep. Both Altera and Xilinx allow each logic element to be used as a simple shift register, and the synthesis tool recognizes these FIFOs (they are coded as normal flip-flops in the Verilog) and uses this special feature of the underlying logic elements. The synthesis tool for Lattice also recognizes these FIFOs and automatically uses embedded RAMs to implement them. The second reason is that Microsemi does not have dedicated carry logic in the FPGA fabric like the other suppliers do. Part of the hash algorithm is a five-input, 32-bit-wide adder, and without dedicated carry logic this requires extra logic elements.

It would be nice if the Microsemi version of the synthesis tool took the Lattice approach and did this automatically. Alternatively, I could rewrite the Verilog to explicitly use embedded memory to implement these two FIFOs. Either method would remove about four hundred flip-flops and potentially make the design fit in the original Microsemi target device. This is another example of how it is often necessary to adjust a design for a specific FPGA target.

The third design is my implementation of an 8051-compatible processor with the standard complement of peripherals. It uses the same two-clock instruction cycle and memory interface as the Y80, and embedded memories for the register file. The 8051 instruction set is not nearly as complex as that of the Z80, so this design has a slightly better balance of logic to flip-flops. The results for this design are shown in Table 6.

It's interesting to compare the number of logic elements required for this design with the same numbers for the Y80 design. The differences range from less than 1% for Altera, to almost 33% for Xilinx. There is no obvious reason for this difference, but it may be that the Xilinx logic elements can be better packed (with both LUT and flip-flop functionality) in this design.

It's also interesting to compare the differences between the frequencies for these two cases. Here, the differences range from about 20% to over 40%, and the largest change occurs with Microsemi. For both designs the slowest paths are in the arithmetic logic unit (ALU), which is considerably simpler in the 8051 case. Since Microsemi does not have dedicated carry logic, it makes sense that this case would show a larger difference in performance.

The final design is my implementation of an AVR-compatible processor. It also uses the same two-clock instruction cycle and memory interface as the Y80, and embedded memories for the register file. The results for this design are shown in Table 7.

As a RISC machine, you might expect this design to do better in terms of both area and frequency, but this isn't really the case. There are a couple reasons for this. First, AVR instructions are sixteen bits wide, which leads to wider busses and more logic, even though the instruction set is quite regular. Second is that the register file also requires sixteen-bit accesses and the ALU performs a number of sixteen-bit operations.

As far as frequency, the slowest paths are in the multiplier, even though I break the multiply into two steps with a pipeline register between the two stages. Both Altera and Xilinx have dedicated two's-complement multipliers in the FPGA fabric, and it might be possible to change the Verilog to take advantage of this dedicated logic. This might give about a 15% improvement in speed, but I'm not going to go through that process at this point.

WRAPPING UP

Even the low-cost FPGA families used here have more than enough resources to implement what would have only been available in a custom chip a few years ago. Yes, it's still more expensive to use an FPGA, but costs will continue to go down, and the flexibility is impossible to beat. Hopefully, the information presented here will help you to avoid some of the pitfalls and choose the right FPGA for your next design.

	Altera	Lattice	Microsemi	Xilinx
Available Logic Elements	6272	3520	6144	3584
Required Logic Elements	2217	3104	4987	2419
Utilization	35%	88%	81%	67%
Flip-flops	706	762	715	727
Frequency (MHz)	58.6	33.1	28.8	49.1

TABLE 7
The YVR microcontroller is a two-clock-per-instruction implementation of the AVR instruction set. It also fits nicely in each FPGA and uses on-chip memory for the register file.