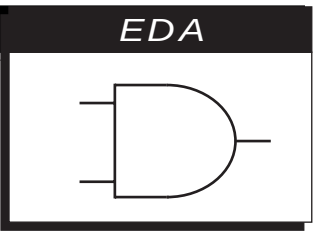

Programmable logic overcomes processor bottlenecks

Steve Knapp

General-purpose microprocessors and microcontrollers have long been mainstays in embedded systems, with programmable-logic devices (PLDs) functioning in a support role. The flexibility and integration of these processors simplify overall system design. However, what happens when the application requires additional computing power? Can programmable logic do more than just decode access to memory and peripherals?

Usually, a designer faced with a performance problem simply upgrades to a more powerful processor with a wider word size and higher clock frequency. This upgrade, though, involves higher component costs—faster memories and perhaps a costly board to accommodate the increased clock frequencies.

What choices do designers have? A more powerful processor is the easy solution, but it's not always the best or right one. For some applications, a modest processor working in conjunction with programmable logic proves a powerful combination. Programmable logic allows a designer to create a standard processor for parts of an application and to offload from it burdensome processes, highly parallel algorithms or time-critical functions.

Steven K Knapp is the founder and president of OptiMagic Inc (Aptos, CA, www.optimagic.com), a firm that develops intellectual property and design software for programmable logic. Prior to founding this firm he held various applications, engineering and management positions at Xilinx and Intel's former programmable-logic division.

Odd-size math

Though processors handle practically any design problem, they might not perform some math or bit-twiddling functions efficiently or quickly. A typical embedded processor possesses limitations such as fixed-datapath width. Simple 8-bit microcontrollers have an 8-bit accumulator or processing unit. Eight bits are fine for many functions, but what happens when you need to process 10-, 12- or even 16-bit data? One solution is to write code to handle multibyte arithmetic. This code is straightforward but quickly consumes processor bandwidth in math-intensive applications. It's not a solution for time-critical functions. You could turn to a more powerful processor with a wider word width. This alternative overcomes the performance limitations but might considerably boost system cost. Yet another approach

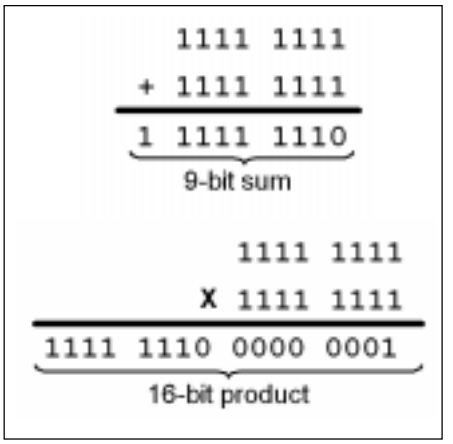

Fig 1—Adding or multiplying two large 8-bit values creates a wider result. Handling numbers at or over a processor's bit width becomes slow and cumbersome.

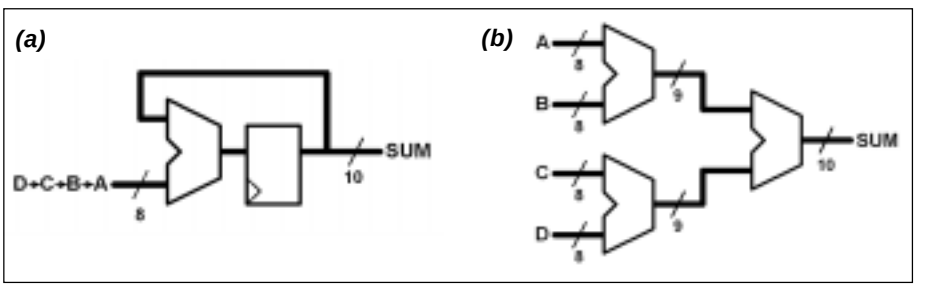

Fig 2—Two possible methods of adding four 8-bit integers, A, B, C and D, demonstrate the size-performance tradeoff typical in FPGAs. All four values share a common space-efficient but slow accumulator in (a). This method is similar to a processor's implementation. A fully parallel implementation (b) is much faster but uses more logic.

(a)

```
; SOLVE A RANDOM LOGIC FUNCTION
: OF 6 VARIABLES BY DIRECTLY POLLING
; EACH BIT (APPROACH USING MCS-51
; UNIQUE BIT-TEST INSTRUCTION
; CAPABILITY) SYMBOLS USED IN LOGIC
; DIAGRAM ASSIGNED TO
; CORRESPONDING 8051 BIT
; ADDRESSES
U        BIT     P1.1
V        BIT     P2.2
W        BIT     TFO
X        BIT     IE1
Y        BIT     20H 0
Z        BIT     21H 1
G        BIT     P3.3
;
TEST_V:  JB      V, TEST_U
         JNB     W, TEST_X
TEST_U:  JB      U, SET_G
TEST_X:  JNB     X, TEST_Z
         JNB     Y, SET_G
TEST_Z:  JNB     Z, SEG_G
CLR_G:   CLR     G
         JMP     NXTTST
SET_G    SETB    G
NXTTST:
```

(b)

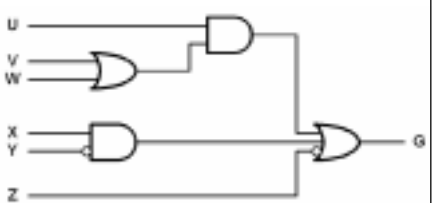

Fig 3—Bit-testing (a) using a microcontroller is thousands of times slower than a programmable logic solution (b).

offloads the wider arithmetic functions to logic implemented in a PLD.

The bit width of an arithmetic operation grows with increased processing. Add two 8-bit values and you could end up with a 9-bit result. Multiplying two 8-bit values might produce a 16-bit result. Add or multiply these values a few times and you could wind up with a very large word (Fig 1). Programmable logic allows you to custom tailor the architecture. Need a 13-bit adder? You'll probably have trouble implementing it with an 8-bit microcontroller but not in a PLD.

An additional benefit of programmable logic is that you choose how parallel to make the overall operation. Like a processor, the programmable device could initiate multiple add operations sequentially and share a common accumulator (Fig 2a). Or it could implement the multiple add operations in parallel and dramatically reduce overall processing time but at the expense of additional logic (Fig 2b).

Another limitation of most processors is that they don't handle bit manipulation well, though some processors have built-in instructions to simplify bit-oriented functions. Programmable logic easily and efficiently handles bit-wise functions and does so with amazingly improved performance. A complex control function might simplify into a small collection of combinatorial logic. Fig 3a demonstrates the difference in complexity between 8051 assembly code and a function that easily fits into just about any PLD (Fig 3b and Ref 1).

Parallel structure

A major limitation of traditional processors for some functions is their fundamental architecture. Most of them employ a von Neuman-style architecture where instructions execute sequentially. While the processor executes one instruction, others wait their turn. Consequently, traditional processors are notoriously poor at implementing parallel algorithms such as those found in most DSP apps.

For example, filtering and image processing are inherently parallel and often contain feedback loops. When handled by a traditional processor, these algorithms are split in time over tens to hundreds of clock periods, consuming a significant portion of the

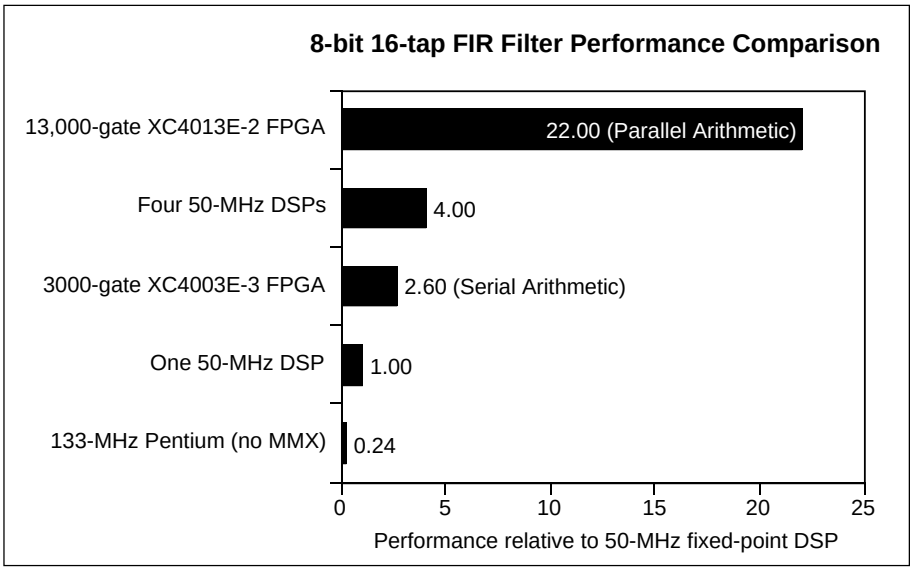

Fig 4—The relative performance for various implementations of an 8-bit 16-tap FIR filter compared to a 50-MHz fixed-point TMS320-series DSP shows price-performance tradeoffs between processors and FPGAs. Note the FPGA performance in the black boxes, where a serial approach is slower but uses less logic. A parallel tack outperforms a typical DSP by a factor of 22 in this example (Ref 2).

EXPERT COLUMN
EDA

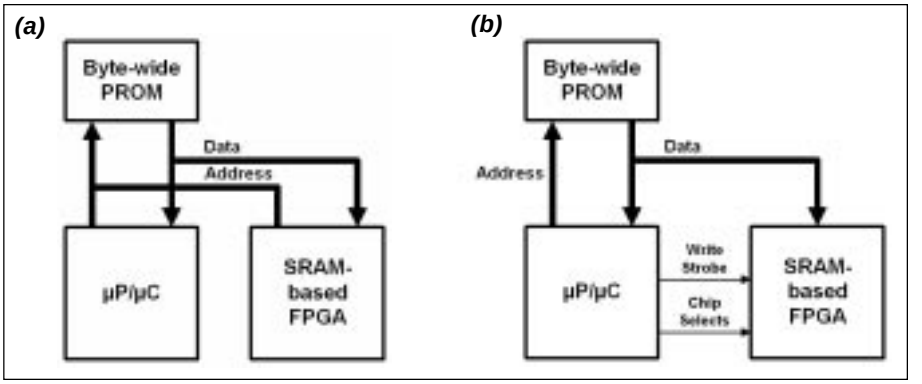

Fig 5—SRAM-based FPGAs have a simple interface with a processor. FPGAs can boot themselves and share a PROM with a processor (a). They can also act like a peripheral, receiving their programming from a processor (b).

processor's possible bandwidth. A designer can balance logic utilization and performance by implementing some functions sequentially and others in parallel.

A PLD can outperform a seemingly fast processor in highly parallel operations. Fig 4 shows the performance of an 8-bit 16-tap FIR filter implemented on various processors. The logic device in this example is a Xilinx 4000E series field-programmable gate array (FPGA). Depending on how the algorithm is implemented, the FPGA outperforms a Pentium-class processor as well as a 50-MHz TMS320-series fixed-point DSP processor. More details on working with signal-processing functions in FPGAs is a topic of a future article.

Essentially, programmable logic helps turbo charge practically any high-performance processing application. An additional advantage is that these devices can integrate interface and system glue logic. For most applications, I tend to favor coarse-grained SRAM-based FPGAs (Ref 3). Devices such as Xilinx's (San Jose, CA) XC4000-Series and Spartan chips, Altera's (San Jose, CA) Flex 10K devices and Lucent Technologies' (Allentown, PA) Orca FPGA family are all high-density high-performance devices with dedicated carry logic for implementing fast arithmetic and counters. They're also based on SRAM technology and are therefore reprogrammable, even in system. Further, these devices have on-chip SRAM that can serve as small FIFOs or other small data buffers. These FPGAs easily interface to a standard microprocessor or microcontroller (Fig 5) either through a parallel peripheral interface or a self-booting interface that shares the processor's boot PROM.

With all of its benefits, engineers should be aware of some potential pitfalls when using programmable logic for embedded processing. Creating a programmable-logic design is more difficult than writing code for a processor. A designer typically creates programmable logic using schematics or describes it with a hardware description language such as VHDL, Verilog or ABEL. While not difficult, writing in these languages presents an extra skill to learn. Obtaining the best performance and density from a PLD usually requires good working knowledge of the underlying architecture.

Further, programmable logic requires another, sometimes expensive, toolset. However, most vendors now supply a lower-cost version of their

This space left blank intentionally.

software supporting their small-to-moderate density devices (see *www.optimagic.com/lowcost.html* for information on free or low-cost software available on the web).

Of course, it's not just the selection of tool sets that determines which path to take. The task at hand dictates the solution. Generally, a processor better implements complex control and general algorithms. Building complex control in a PLD is more difficult and time consuming. Furthermore, a small change in a control program has little effect on the processor implementation.

In contrast, a small change in a control state machine built using programmable logic might have a significant effect on the size and resulting performance of the logic implementation. Typically, programmable logic is better than processor-based programs at simple control functions using high-speed state machines.

Function for function, a processor's logic is generally less expensive than a PLD. Thousands of separate functions or processes share this logic. However, programmable logic generally outperforms a processor on dedicated functions. For performance-critical applications, a PLD can be less expensive than a dedicated processor. In an extreme example (see *http://www.hpcc.gov/pubs/blue94/section.5.15.html*), a board full of FPGAs outperformed conventional supercomputers by a factor of 100 or more—and it was many orders of magnitude less expensive. ***PE&IN***

References

1. Wharton, J, Intel Corp (*http://developer.intel.com/design/mcs51/applnots/01502a01.htm*), App Note AP-69, pgs 27-28.
2. Knapp, S, Xilinx Inc (*http://www.xilinx.com/appnotes/dspintro.pdf*), "Using Programmable Logic to Accelerate DSP Functions," pg 2.
3. Knapp, S, "Understanding programmable logic means digesting its alphabet soup," *PE&IN*, July 1997, pgs 75-79.

Editorial Feedback
This article's value to me was:
High—263 Average—264 Low—265

This space left blank intentionally.