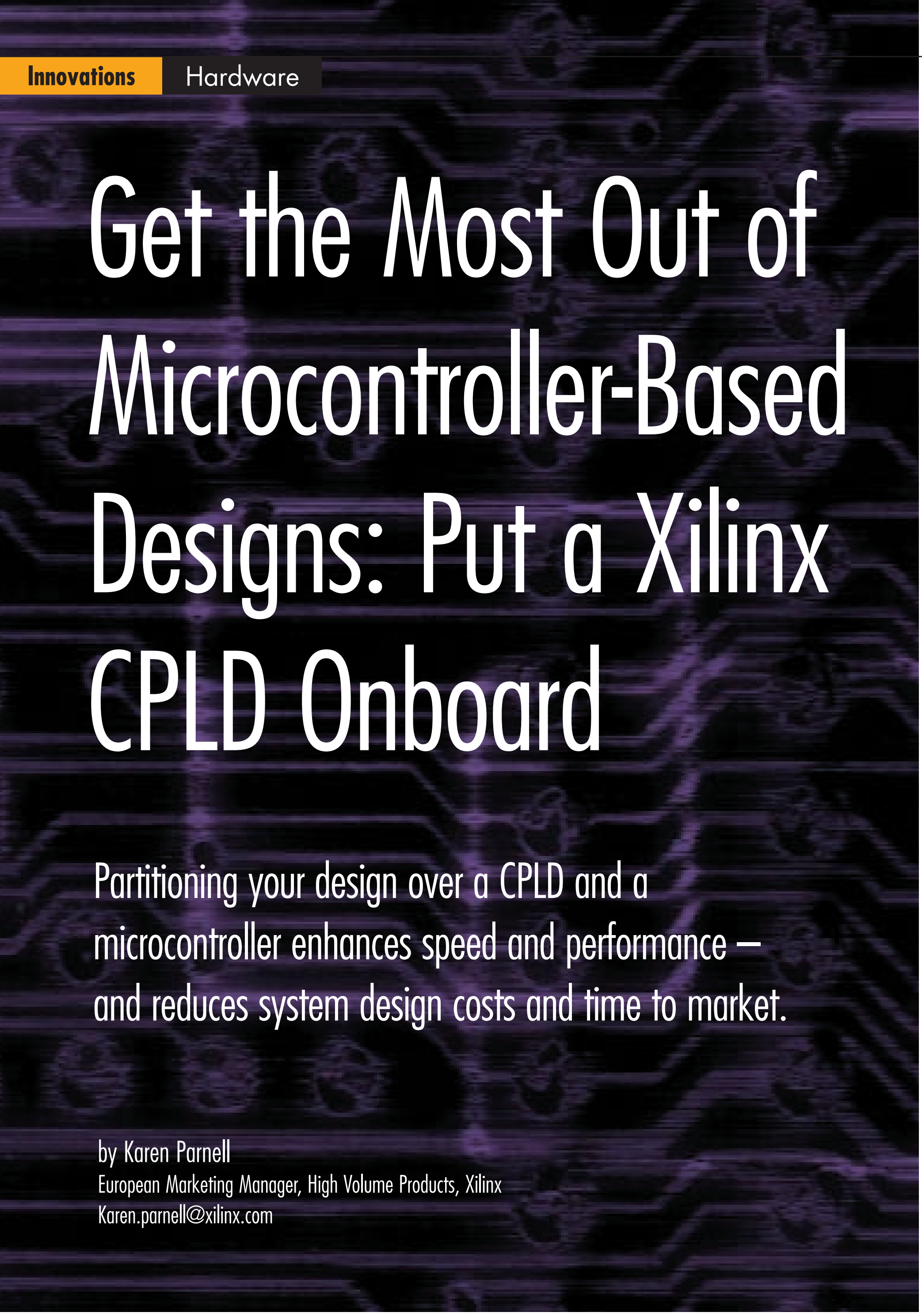

Get the Most Out of Microcontroller-Based Designs: Put a Xilinx CPLD Onboard

Partitioning your design over a CPLD and a microcontroller enhances speed and performance – and reduces system design costs and time to market.

by Karen Parnell
European Marketing Manager, High Volume Products, Xilinx
Karen.parnell@xilinx.com

Microcontrollers don't make the world go round, but they most certainly help us get around in the world. You can find microcontrollers in automobiles, microwave ovens, automatic teller machines, VCRs, point of sale terminals, robotic devices, wireless telephones, home security systems, and satellites, just to name a very few applications.

In the never-ending quest for faster, better, cheaper products, advanced designers are now pairing complex programmable logic devices (CPLDs) with microcontrollers to take advantage of the strengths of each. Microcontrollers are naturally good at sequential processes and computationally intensive tasks, as well as a host of non-time-critical tasks. CPLDs such as Xilinx CoolRunner™ devices are ideal for parallel processing, high-speed operations, and applications where lots of inputs and outputs are required.

Although there are faster and more powerful microcontrollers in the field, 8-bit microcontrollers own much of the market because of their low cost and low power characteristics. The typical operational speed of an 8-bit microcontroller is around 20 MHz, but some microcontroller cores divide clock frequency internally and use multiple clock cycles per instruction (operations often include fetch-and-execute instruction cycles). Thus, with a clock division of two and with each instruction taking up to three cycles, the actual speed of a 20 MHz microcontroller is divided by six. This works out to an operational speed of only 3.33 MHz.

CoolRunner CPLDs are much faster than microcontrollers and can easily reach system speeds in excess of 100 MHz. Today, we are even seeing CoolRunner devices with input to output delays as short as 3.5 ns, which equates to impressive system speeds as fast as 285 MHz. CoolRunner CPLDs make ideal partners for microcontrollers, because they not only perform high-speed tasks, they perform those tasks with ultra low power consumption.

Also, Xilinx offers free software and low cost hardware design tools to support CPLD integration with microcontrollers. The Xilinx CPLD design process is quite similar to that used on microcontrollers, so designers can quickly learn how to partition their designs across a CPLD and a microcontroller to maximum advantage.

So far, a design partition over a microcontroller and a CPLD sounds good in theory, but will it work in the field? We will devote the rest of this article to design examples that show how you can enhance a typical microcontroller design by utilizing the computational strengths of the microcontroller and the speed of a CoolRunner CPLD.

Conventional Stepper Motor Control

A frequent use of microcontrollers is to run stepper motors. Figure 1 depicts a typical four-phase stepper motor driving circuit. The four windings have a common connection to the motor supply voltage (Vss), which typically ranges from 5 volts to 30 volts. A high power NPN transistor drives each of the four phases. (Incidentally, MOSFETs – metal oxide semiconductor field effect transistors – can also be used to drive stepper motors.)

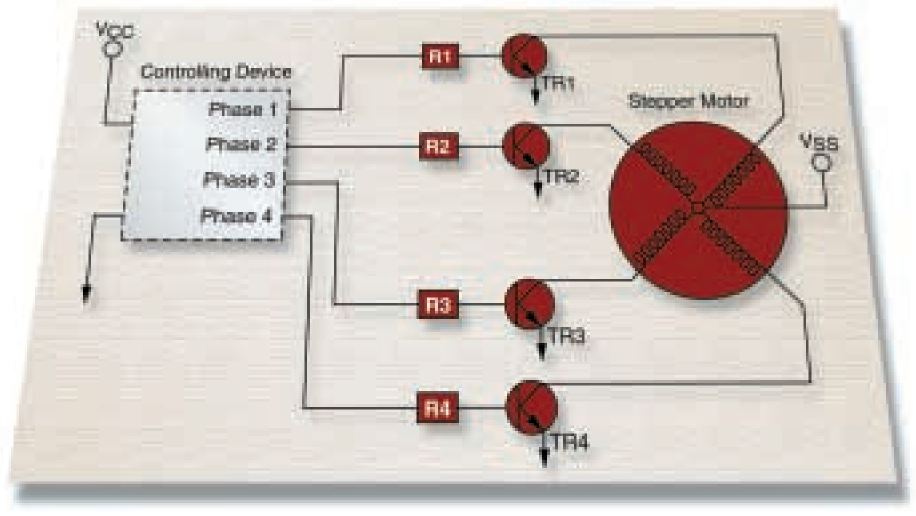

Figure 1 - Stepper motor controller

Each motor phase current may range from 100 mA to as much as 10 A. The transistor selection depends on the drive current, power dissipation, and gain. The series resistors should be selected to limit the current to 8 mA per output to suit either the microcontroller or CPLD outputs. The basic control sequence of a four-phase motor is achieved by activating one phase at a time.

At the low cost end, the motor rotor rotates through 7.5 degrees per step, or 48 steps per revolution. The more accurate, higher cost versions have a basic resolution of 1.8 degrees per step. Furthermore, it is possible to half-step these motors to achieve a resolution of 0.9 degrees per step. Stepper motors tend to have a much lower torque than other motors, which is advantageous in precise positional control.

The examples that follow show how either a microcontroller or a CPLD can be used to control stepper motor tasks to varying degrees of accuracy. We can see from Figure 2 that the design flow for both is quite similar.

Both flows start with text entry. Assembly language targets microcontrollers. ABEL (Advanced Boolean Expression Language) hardware description language targets CPLDs. After the text "description" is entered, the design is either compiled (microcontroller) or synthesized (CPLD). Next, the design is verified by some form of simulation or test. Once verified, the design is downloaded to the target device – either a microcontroller or CPLD. We can then program the devices in-system using an inexpensive ISP (in-system programming) cable.

One of the advantages of a CPLD over a microcontroller occurs during board level testing. Using a JTAG boundary scan, the CPLD can be fully tested on the board. The CPLD can also be used as a "gateway" to test the rest of the board functionality. After the board level test is completed, the CPLD can then be programmed with the final code in-system via the JTAG port.

(A JTAG boundary scan – formally known as IEEE/ANSI standard 1149.1_1190 – is a set of design rules, which facilitate testing, device programming, and debugging at the chip, board, and system levels.)

Microcontrollers can include monitor debug code internal to the device for limited code testing and debugging. With the advent of flash-based microcontrollers, these can now also be programmed in-system.

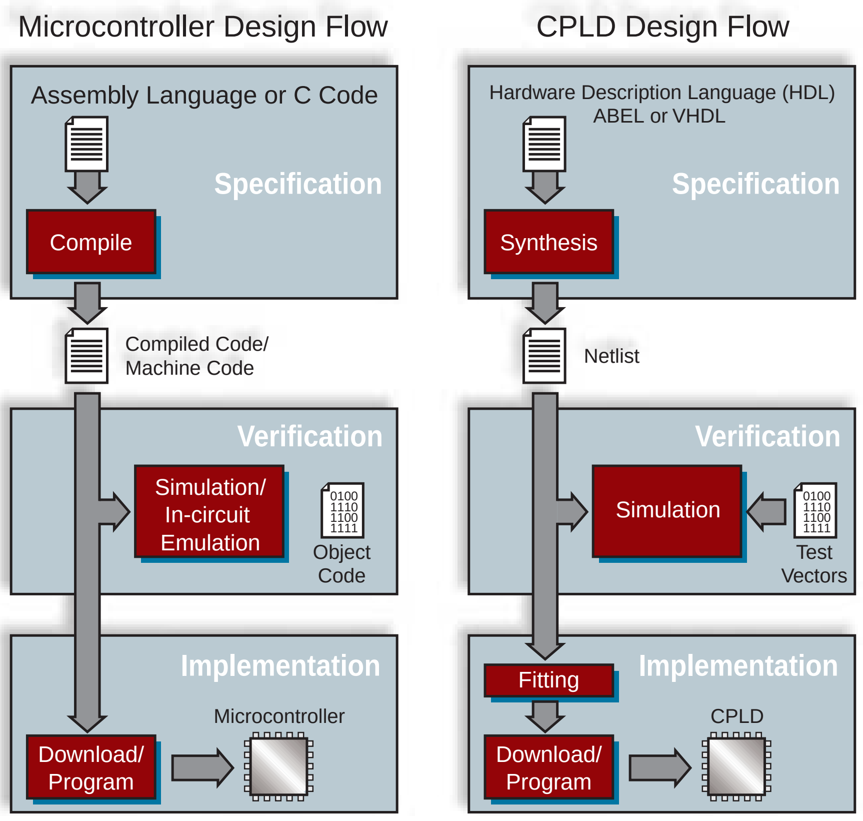

Figure 2 - Design flow comparisons

Using a Microcontroller to Control a Stepper Motor

Figure 3 shows assembly language targeting a Philips 80C552 microcontroller. The stepper motor the microcontroller will control has four sets of coils. When logic level patterns are applied to each set of coils, the motor steps through its angles. The speed of the stepper motor shaft depends on how fast the logic level patterns are applied to the four sets of coils. The manufacturer's motor specification data sheet provides the stepping motor code. A very common stepping code is given by the following hexadecimal numbers:

A 9 5 6

Each hex digit is equal to four binary bits:

1010 1001 0101 0110

```
$MOD552              ; include file for 80C552
        ORG 0    ; reset address
        SJMP START  ; jump over reserved area
        ORG 30H:program start address
START:  MOV P1,#0AH ;move hex 0A into lower
                ;4 bits of port 1
        ACAL DELAY  ;call subroutine step hold
                ;delay
        MOV P1, #09H        ;move hex 09 into lower
                ;4 bits of port 1
        ACALL DELAY
        MOV P1,#05H
        ACALL DELAY
        MOV P1, #06H
        ACALL DELAY
        SJMP START          ;repeat stepping pattern
                        ;
                        ;Double loop delay
DELAY:          MOV R1, #0FFH ;put hex FF into register 1
OUTER:          MOV R0, #0FFH ;put hex FF into register 0
INNER:  DJNZ R0, INNER;decrement r0 until it is 0
        DJNZ R1,OUTER;dec r1, go to outer until
                        ; r1 = 0
        RET ;return from subroutine
```

Figure 3 - Assembly language program to rotate the stepper motor shaft

These binary bits represent voltage levels applied to each of the coil driver circuits. The steps are:

1010	5V	0V	5V	0V
1001	5V	0V	0V	5V
0101	0V	5V	0V	5V
0110	0V	5V	5V	0V

If you send this pattern repeatedly, then the motor shaft rotates. The assembly language program in Figure 3 continually rotates the stepper motor shaft. By altering the value of R0 in the delay loop, this will give fine control over speed; altering the value of R1 will give coarse variations in speed.

Stepper Motor Control Using a CPLD

Figure 4 shows a design written in ABEL hardware description language. Within the Xilinx CPLD, four inputs are required to fully control the stepper motor. The clock (clk) input synchronizes the logic and determines the speed of rotation. The motor advances one step per clock period. The angle of rotation of the shaft will depend on the specific motor used. The direction (dir) control input changes the sequence at the outputs (ph1 to ph4) to reverse the motor direction. The enable input (en) determines whether the motor is rotating or holding. The active low reset input (rst) initializes the circuit to ensure the correct starting sequence is provided to the outputs.

The phase equations (ph1 to ph4) are written with a colon and equal sign (:=) to indicate a registered implementation of the combinatorial equation. Each phase equation is either enabled (en), indicating that the motor is rotating, or disabled (!en), indicating that the current active phase remains on and the motor is locked. The value of the direction input (dir) determines which product term is used to sequence clockwise or counter-clockwise. The asynchronous equations (for example, ph1.AR=!rst) initialize the circuit.

The ABEL hardware description motor control module can be embedded within a macro function and saved as a reuseable standard logic block, which can be shared by many designers within the same organization – this is the beauty of design re-use. This "hardware" macro function is independent of any other function or event not related to its operation. Therefore, it cannot be affected by extraneous system interrupts or other unconnected system state changes. Such independence is critical in safety systems. Extraneous system interrupts in a purely software-based system could cause indeterminate states that are hard to test or simulate.

```
MODULE step1
Title 'step1_abl'

Declarations
clk  PIN;    //input to determine speed of rotation
en   PIN;  //determines whether motor rotating or holding
dir  PIN;   //motor direction control
rst  PIN;   //resets & initialises circuit
ph1  PIN istype 'reg';       //output to motor phase 1
ph2  PIN istype 'reg';       // output to motor phase 2
ph3  PIN istype 'reg';       // output to motor phase 3
ph4  PIN istype 'reg' ;      // output to motor phase 4

Equations
//Stepper motor controller description
ph1 := !dir * en * (!ph1 * !ph2 * !ph3 * !ph4)
        + dir * en * (!ph1 * ph2 * !ph3 * !ph4)
        + !en * ph1;

ph2 := !dir * en * (!ph1 * !ph2 * !ph3 * !ph4)
        + dir * en * (!ph1 * ph2 * !ph3 * !ph4)
        + !en * ph2;

ph3 := !dir * en * (!ph1 * !ph2 * !ph3 * !ph4)
        + dir * en * (!ph1 * ph2 * !ph3 * !ph4)
        + !en * ph3;

ph4 := !dir * en * (!ph1 * !ph2 * !ph3 * !ph4)
        + dir * en * (!ph1 * ph2 * !ph3 * !ph4)
        + !en * ph4;

ph1.PR = !rst;
ph2.AR = !rst;
ph3.AR = !rst;
ph4.AR = !rst;

end step1
```

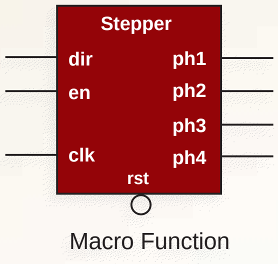

Figure 4 - Using a CPLD to control a stepper motor

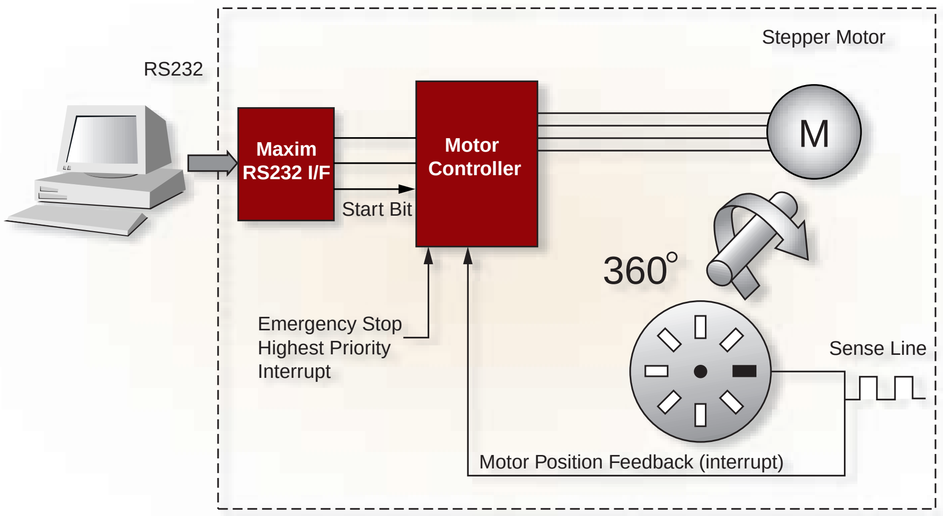

Figure 5 - Design partitioning

PC-Based Motor Control

Our next example (Figures 5 and 6) is more complex, because now the motor is connected to a PC-based system via an RS232 serial connection. This implementation has a closed loop system controlling rotation, speed, and direction. There is also the addition of a safety-critical emergency stop, which has the highest level of system interrupt. This means that if the emergency stop is activated, it will override any other process or interrupt and will immediately stop the motor from rotating.

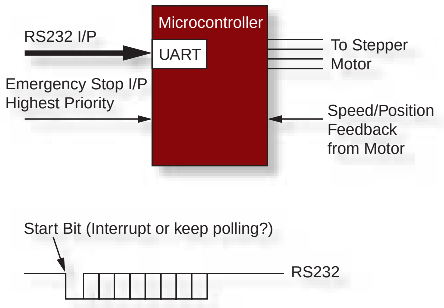

Figure 6 - Microcontroller Implementation

This design example uses only a microcontroller. The main functions it performs are:

- Interrupt control
- Status feedback to the PC
- Accurate motor control.

This configuration would probably be implemented in a single microcontroller device with specific motor control peripherals, such as a capture-compare unit. This configuration would also need a built-in UART (Universal Asynchronous Receiver Transmitter). These extra functions usually add extra cost to the overall microcontroller device.

Due to the nature of the microcontroller, the interrupt handling must be thoroughly mapped out, because interrupts could affect the speed of the motor. In a safety-critical system, emergency stops implemented in software require exhaustive testing and verification before they can be used in the final system to ensure that they operate properly under all software related conditions, including software bugs and potential software states. The output from the motor rotation sensor is very fast, so control of the speed of the motor could cause problems if system interrupts occurred.

Design Partitioning

As we noted before, microcontrollers are very good at computational tasks, and CPLDs are excellent in high speed systems and have an abundance of I/Os. Figure 7 shows how we can use a microcontroller and a CPLD in a partitioned design to achieve the greatest control over a stepper motor.

The microcontroller:

- Interprets ASCII commands from the PC.
- Reports status of the motor to the PC.
- Converts required speed into control vectors (small mathematical algorithm).
- Decides direction of rotation of the motor.
- Computes stop point and sets a value into the pulse count comparison register.
- Monitors progress (control loop) and adapts speed.
- Recovers from emergency stops.

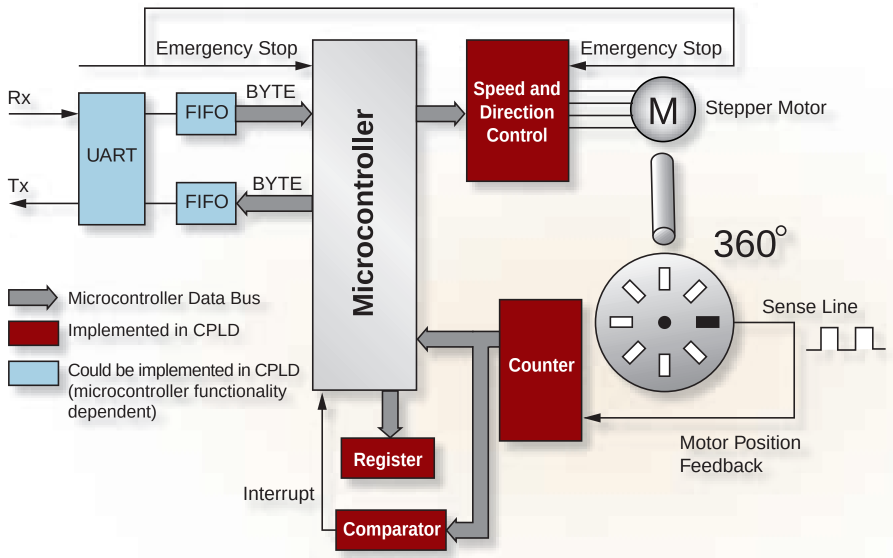

Figure 7 - Partitioned Design: Microcontroller and CPLD

Although the microcontroller performs recovery from emergency stops, the actual emergency stop is implemented by the CPLD, because this is the safety-critical part of the design. Because the CPLD is considered independent hardware, safety-critical proving and sign off are more straightforward than software safety systems. Additionally, all of the high-speed interface functions are also implemented in the CPLD, because it is very fast and has abundant inputs and outputs.

Meanwhile, the UART and FIFO (First in, First Out) sections of the design can be implemented in the microcontroller in the form of a costed microcontroller peripheral or may be implemented in a larger more granular programmable logic device like a field programmable gate array (FPGA) – for example, a Xilinx Spartan™ device. Using a programmable logic device in a design has the added benefit of the ability to absorb any other discrete logic elements on the printed circuit board or in the total design into the CPLD. Under this new configuration, we can consider the CPLD as offering hardware-based subroutines or as a mini co-processor.

The microcontroller still performs ASCII string manipulation and mathematical functions, but it now has more time to perform these operations – without interruption. The motor control is now independently stable and safe.

Microcontroller/CPLD design partitioning can reduce overall system costs. This solution uses low cost devices to implement the functions they do best – computational functions in the microcontroller and high speed, high I/O tasks in the CPLD. In safety-critical systems, why not put the safety critical functions (e.g., emergency stop), in hardware (CPLDs) to cut down safety system approval time scales?

System testing can also be made easier by implementing the difficult-to-simulate interrupt handling into programmable logic. Low cost microcontrollers are now in the region of US$1.00, but if your design requires extra peripherals (e.g., capture-compare unit for accurate motor control, analog-to-digital converters, or UARTs), this can quadruple the cost of your microcontroller. A low cost microcontroller coupled with a low cost CPLD from Xilinx can deliver the same performance – at approximately half the cost.

In low power applications, microcontrollers are universally accepted as low power devices and have been the automatic choice of designers. The CoolRunner family of ultra low power CPLDs are an ideal fit in this arena and may be used to complement your low power microcontroller to integrate designs in battery powered, portable designs (<100 µA current consumption at standby).

Conclusion

Microcontrollers are ideally suited to computational tasks, whereas CPLDs are suited to very fast, I/O intensive operations. Partitioning your design across the two devices can increase overall system speeds, reduce costs, and potentially absorb all of the other discrete logic functions in a design – thus presenting a truly reconfigurable system.

The design process for a microcontroller is very similar to that of a programmable logic device. This permits a shorter learning and designing cycle. Full functioning software design tools for Xilinx CPLDs are free of charge and may be downloaded from the Xilinx website. Thus, your first project using CPLDs can not only be quick and painless, but very cost-effective.

The following URLs provide detailed information on the topics and hardware discussed in this article:

Xilinx Website: www.xilinx.com

Xilinx CoolRunner CPLDs:
www.xilinx.com/products/xpla3.htm

Xilinx Free CPLD design software:
www.xilinx.com/products/software/webpowered.htm