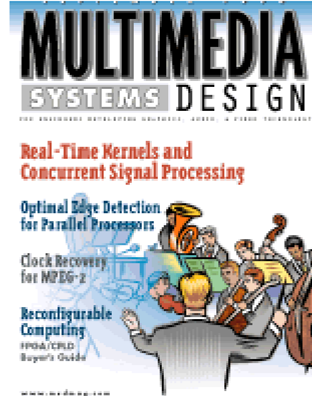
A Reconfigurable Computing Primer

by Michael Barr



Copyright © 1998 by Miller Freeman, Inc. This article may not be used for commercial purposes without the written consent of the authors and publishers.

Designers of multimedia systems face three significant challenges in today's ultra-competitive marketplace: Our products must do more, cost less, and be brought to the market quicker than ever. Though each of these goals is individually attainable, the hat trick is generally unachievable with traditional design and implementation techniques. Fortunately, some new techniques are emerging from the study of reconfigurable computing that make it possible to design systems that satisfy all three requirements simultaneously.

Although originally proposed in the late 1960s by a researcher at UCLA, reconfigurable computing is a relatively new field of study. The decades-long delay had mostly to do with a lack of acceptable reconfigurable hardware. Reprogrammable logic chips like field programmable gate arrays (FPGAs) have been around for many years, but these chips have only recently reached gate densities making them suitable for high-end applications. (The densest of the current FPGAs have approximately 100,000 reprogrammable logic gates.) With an anticipated doubling of gate densities every 18 months, the situation will only become more favorable from this point forward.

One of our clients, TSI TelSys, has been developing and using reconfigurable computing technologies for almost three years. Their primary product is groundstation equipment for satellite communications. This application involves high-rate communications, signal processing, and a variety of network protocols and data formats. What follows is an introduction to the terminology and techniques we have developed as our experience with reconfigurable computing has grown. I hope that this explanation will help other system designers benefit from the work that's already been done.

What is Reconfigurable Computing?

When we talk about reconfigurable computing we're usually talking about FPGA-based system designs. Unfortunately, that doesn't qualify the term precisely enough. System designers use FPGAs in many different ways. The most common use of an FPGA is for prototyping the design of an ASIC. In this scenario, the FPGA is present only on the prototype hardware and is replaced by the corresponding ASIC in the final production system. This use of FPGAs has nothing to do with reconfigurable computing.

However, many system designers are choosing to leave the FPGAs as part of the production hardware. Lower FPGA prices and higher gate counts have helped drive this change. Such systems retain the execution speed of dedicated hardware but also have a great deal of functional flexibility. The logic within the FPGA can be changed if or when it is necessary, which has many advantages. For example, hardware bug fixes and upgrades can be administered as easily as their software counterparts. In order to support a new version of a network protocol, you can redesign the internal logic of the FPGA and send the enhancement to the affected customers by email. Once they've downloaded the new logic design to the system and restarted it, they'll be able to use the new version of the protocol. This is configurable computing; reconfigurable computing goes one step further.

Reconfigurable computing involves manipulation of the logic within the FPGA at run-time. In other words, the design of the hardware may change in response to the demands placed upon the system while it is running. Here, the FPGA acts as an execution engine for a variety of different hardware functions — some executing in parallel, others in serial — much as a CPU acts as an execution engine for a variety of software threads. We might even go so far as to call the FPGA a reconfigurable processing unit (RPU).

Reconfigurable computing allows system designers to execute more hardware than they have gates to fit, which works especially well when there are parts of the hardware that are occasionally idle. One theoretical application is a smart cellular phone that supports multiple communication and data protocols, though just one a time. When the phone passes from a geographic region that is served by one protocol into a region that is served by another, the hardware is automatically reconfigured. This is reconfigurable computing at its best, and using this approach it is possible to design systems that do more, cost less, and have shorter design and implementation cycles.

What are the Advantages?

Reconfigurable computing has several advantages. First, it is possible to achieve greater functionality with a simpler hardware design. Because not all of the logic must be present in the FPGA at all times, the cost of supporting additional features is reduced to the cost of the memory required to store the logic design. Consider again the multiprotocol cellular phone. It would be possible to support as many protocols as could be fit into the available on-board ROM. It is even conceivable that new protocols could be uploaded from a base station to the handheld phone on an as-needed basis, thus requiring no additional memory.

The second advantage is lower system cost, which does not manifest itself

exactly as you might expect. On a low-volume product, there will be some production cost savings, which result from the elimination of the expense of ASIC design and fabrication. However, for higher-volume products, the production cost of fixed hardware may actually be lower. We have to think in terms of lifetime system costs to see the savings. Here, technical obsolescence drives up the cost of systems based on fixed-hardware designs. Systems based on reconfigurable computing are upgradable in the field. Such changes extend the useful life of the system, thus reducing lifetime costs.

The final advantage of reconfigurable computing is reduced time-to-market. The fact that you're no longer using an ASIC is a big help in this respect. There are no chip design and prototyping cycles, which eliminates a large amount of development effort. In addition, the logic design remains flexible right up until (and even after) the product ships. This allows an incremental design flow, a luxury not typically available to hardware designers. You can even ship a product that meets the minimum requirements and add features after deployment. In the case of a networked product like a set-top box or cellular telephone, it may even be possible to make such enhancements without customer involvement!

Reconfigurable Hardware

Traditional FPGAs are configurable, but not run-time reconfigurable. Many of the older FPGAs expect to read their configuration out of a serial EEPROM, one bit at a time. And they can only be made to do so by asserting a chip reset signal. This means that the FPGA must be reprogrammed in its entirety and that its previous internal state cannot be captured beforehand. Though these features are compatible with configurable computing applications, they are not sufficient for reconfigurable computing.

In order to benefit from run-time reconfiguration, it is necessary that the FPGAs involved have some or all of the following features. The more of these features they have, the more flexible can be the system design.

On-the-Fly Reprogrammability

Whenever possible, we'd like to avoid resetting the FPGA, mostly because it takes a lot of time. Ideally, we could just stop the clock going to some or all of the chip, change the logic within that region, and restart the clock. That way, there isn't as much wasted time, or configuration overhead. The more configuration overhead there is the more likely that the system performance will be unacceptably below that of a fixed-hardware version. Of course, a small performance hit (like stopping the clock) is itself a reasonable trade-off for the added benefits of hardware flexibility.

Partial Reprogrammability

Even better would be the ability to leave most of the internal logic in place and change just one part. The Atmel 40K and Xilinx 62xx series FPGAs have such a feature. Any gate or set of gates may be changed without affecting the state of the others. Figure 1 shows how this might be used in practice. It will always be much faster to change a small piece of the logic than the entire FPGA contents.

Externally-Visible Internal State

If you can see the internal state of the FPGA at any time, then it is also possible to capture that state and save it for later use. For example, the Xilinx 62xx series FPGAs feature a 32-bit data bus called the FastMAP processor interface. This allows the internal state of the FPGA to be read and written just like memory and makes it possible to “swap” hardware designs in much the same way that pages of virtual memory are swapped into and out of physical memory.

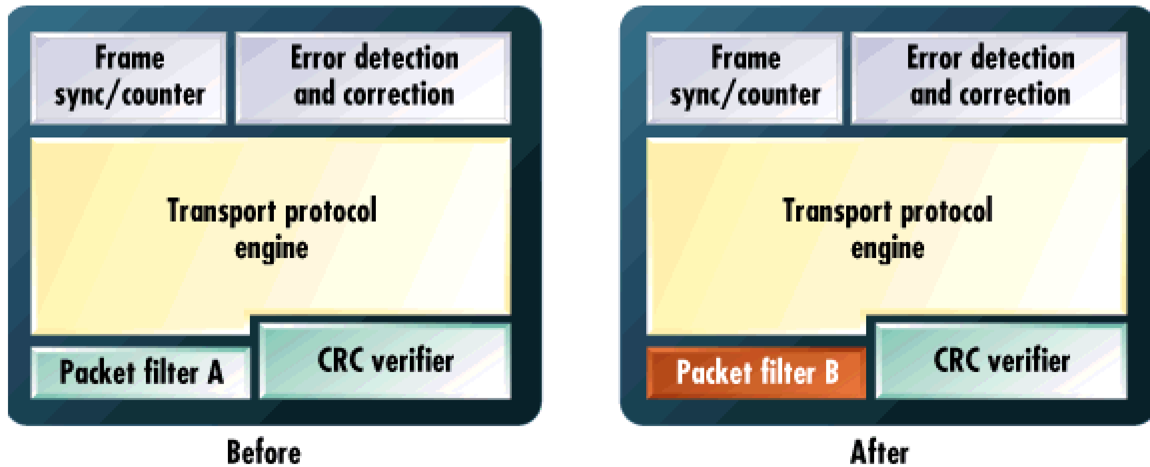


Figure 1. Partial Reprogrammability Allows Partial Changes

Hardware Objects

Before going on, we need to define a new term. A hardware object is a functional or logical hardware component that contains its own configuration and state information. In other words, it is a piece of logic that can be executed in an RPU. Hardware objects are position-independent, or relocatable, to allow us to execute the hardware object from any convenient and available position within the chip. To actually take that leap requires a few assumptions.

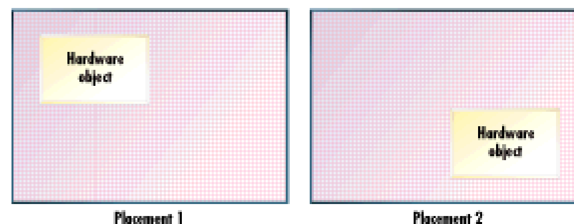


Figure 2. Relocatable Hardware Objects are Position-Independent

First, if we’re going to be working with relocatable logic blocks, it is desirable to add constraints on their size and shape. These constraints limit the number of possible positions within the FPGA and make run-time decision-making more efficient and effective. The actual constraints should be based on the features of a particular FPGA or FPGA family. However, the best constraints require that all hardware objects be rectangular in shape and have edge lengths that are multiples of some unit length (called the hardware page size), which may be any convenient number of gates. For example, page sizes of 4 and 16 gates work very well for the Xilinx 62xx series FPGAs because these parts have additional routing resources at each of those intersections, which makes routing between hardware objects or a hardware object and its I/O pins much easier.

Second, it is desirable to define a standard look and feel for hardware object interfaces. The idea here is to make interobject routing easier by defining standard interfaces between them. This is especially important if routing between objects will be performed on-the-fly, and it also paves the way for greater hardware object re-use. By standardizing the interfaces of all hardware objects, it is possible to maintain libraries of frequently used objects and to quickly build larger designs from these smaller components. In some cases, it may even be possible to purchase third-party hardware objects rather than designing your own.

You may be wondering how you can build a “generic” hardware object that will work in any system. To do that, we need to make one final assumption. Assume that any hardware objects that expect to interface to the world outside the RPU (to a block of memory, the processor, a peripheral, or even another RPU) must do so through an abstraction. This abstraction is called the hardware object framework, which is a ring of logic that is always present within the RPU and physically located along the outer edges. The framework provides a set of standard interfaces to memory and peripheral devices located outside of the RPU. This ring of logic shrinks the available space for executing hardware objects (see Figure 3), but that is a small price to pay for greater hardware object re-usability and, hence, faster design cycles.

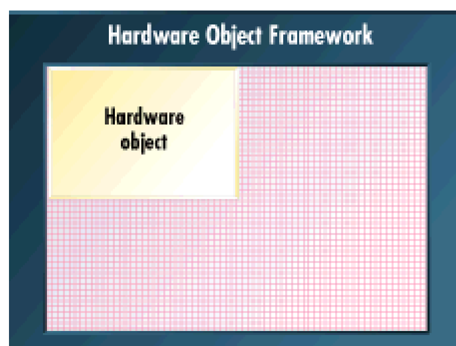


Figure 3. A Hardware Object Framework Surrounds the Hardware Objects

Run-Time Environments

Due to the dynamic nature of reconfigurable computing, it is sometimes helpful to have software manage the processes of:

- Deciding which hardware objects to execute and when
- Swapping hardware objects into and out of the reconfigurable logic
- Performing routing between hardware objects or between hardware objects and the hardware object framework.

Of course, having software manage the reconfigurable hardware usually means having an embedded processor or microcontroller on-board. (We expect several vendors to introduce single-chip solutions that combine a CPU core and a block of reconfigurable logic by year’s end.) The embedded software that runs there is called the run-time environment and is analogous to the operating system that manages the execution of multiple software threads. Like threads, hardware objects may have priorities, deadlines, and contexts, etc. It is the job of the run-time environment to organize this information and make decisions based upon it.

The reason we need a run-time environment at all is that there are decisions to

be made while the system is running. And as human designers, we are not available to make these decisions. So we impart these responsibilities to a piece of software. This allows us to write our application software at a very high level of abstraction. For example, if the application involves manipulation of images in the JPEG format, it would be ideal to have only two blocks of logic: one for JPEG compression, and the other for decompression. Then we could simply hand our input data and the appropriate logic block to the run-time environment and wait for the results. This is equivalent to saying: “Please execute the attached hardware object and let me know when it is done. If there are any results, please let me know as soon as they become available.”

To do this, the run-time environment must first locate space within the RPU that is large enough to execute the given hardware object. It must then perform the necessary routing between the hardware object’s inputs and outputs and the blocks of memory reserved for each data stream. Next, it must stop the appropriate clock, reprogram the internal logic, and restart the RPU. Once the object starts to execute, the run-time environment must continuously monitor the hardware object’s status flags to determine when it is done executing. Once it is done, the caller can be notified and given the results. The run-time environment is then free to reclaim the reconfigurable logic gates that were taken up by that hardware object and to wait for additional requests to arrive from the application software.

By assigning all of these tasks to a special piece of software, we hope to make it possible to develop generic run-time environments. Much as there is a market for commercial operating systems for CPUs, we expect a market for commercial run-time environments for RPUs will emerge if reconfigurable computing becomes popular. That would save system designers even more time by allowing them to purchase a run-time environment rather than design their own. At that point, system design becomes a matter of purchasing or developing the required hardware object libraries, configuring a third-party run-time environment, and writing the application software--a truly efficient system development paradigm.

Internally, our run-time environment can be thought of as a series of three layers (see Figure 4). The device abstraction layer is the lowest level and is responsible for hiding the details of a particular FPGA or FPGA family. This is analogous to the parts of an operating system that must be written in assembly language because they are processor-specific. The device abstraction layer can answer the following questions about the hardware: How many FPGAs are present? What types are they? What is the hardware page size? What are their dimensions (height and width as multiples of the hardware page size)? What routing resources are available at the edge of each hardware page? The device abstraction layer also provides a simple read/write interface for the layer above.

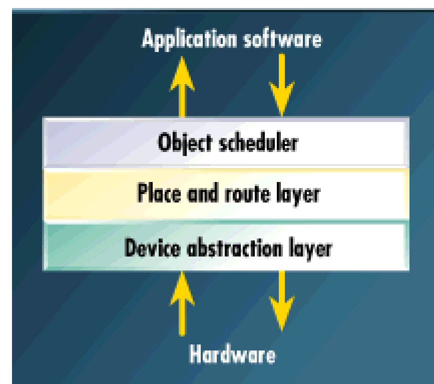


Figure 4. A Run-Time Environment Contains Three Layers

The middle layer is responsible for placement and routing of hardware objects. It maintains a logical representation of the free space within the RPU and decides where each object will be physically located within the device. It is also responsible for adding routing between hardware objects or between one hardware object and the hardware object framework. This is the most complicated layer of the three.

The uppermost layer is called the object scheduler. It provides an application programming interface (API) that makes using the RPUs easy for the application programmer and is responsible for deciding which hardware objects are currently running. This decision may be based on any convenient scheduling algorithm. For example, first-come first-serve, round-robin, and priority-based schemes are reasonable choices. But in order to implement the latter pair, it would be necessary to first implement hardware object swapping. Hardware object swapping involves saving the current state of a running piece of logic and later restoring it, and is only possible in systems that employ FPGAs with externally visible internal states.

Looking Forward

The principal benefits of reconfigurable computing are the ability to execute larger hardware designs with fewer gates and to realize the flexibility of a software-based solution while retaining the execution speed of a more traditional, hardware-based approach. This makes doing more with less a reality.

In our own business we have seen tremendous cost savings, simply because our systems do not become obsolete as quickly as our competitors'. This has even led us to use the marketing slogan "Obsolescence is Obsolete" because reconfigurable computing enables the addition of new features in the field, allows rapid implementation of new standards and protocols on an as-needed basis, and protects their investment in computing hardware.

Whether you do it for your customers or for yourselves, you should at least consider using reconfigurable computing in your next design. You may find, as we have, that the benefits far exceed the initial learning curve. And as reconfigurable computing becomes more popular, these benefits will only increase. The idea of buying third-party hardware objects and run-time environments and simply combining them in new and interesting ways to create a product is certainly forward-looking, but it may not be that far over the horizon.

This article was published in the September 1998 issue of Multimedia Systems Design. If you wish to cite the article in your own work, you may find the following MLA-style information helpful:

Barr, Michael. "A Reconfigurable Computing Primer" Multimedia Systems Design, September 1998, pp. 44-47.