System-on-a-chip design offers great promise by shrinking an entire computer to a single chip. But with the promise come challenges that need to be overcome before SoC reaches its full potential.

SoC

Software Hardware **NIGHTMARE** or *Bliss*

TELLE WHITNEY, PH.D.
GEORGE NEVILLE-NEIL

System-on-a-chip (SoC) design methodology allows a designer to create complex silicon systems from smaller working blocks, or systems. By providing a method for easily supporting proprietary functionality in a larger context that includes many existing design pieces, SoC design opens the craft of silicon design to a much broader audience.

Chip design complexity trends continue to follow Moore's law, well beyond where many pundits thought it would end. Systems using a feature size of 90 nanometers are in design, and chips of 0.13 microns are in production. Complex designs now include 20 million logic gates, or 200 million transistors on a 1 cm² die. By way of comparison, only a few years ago 2 million logic gates or 10 million transistors were common.

Embedded systems developers have embraced SoC design because it allows high-volume offerings such as

SoC

Software
Hardware
NIGHTMARE
or *Bliss*

cellphones, PDAs, and GPSs to replace expensive boards with a single chip. Although SoC designs are common in the embedded world, to date they have focused primarily on simple designs. That simplicity is no longer sufficient for the sophistication desired by many of the high-end products in development. In the near future both high-end PDA devices and sophisticated cameras will extend their functionality by offering chips that include multiple processors and high-speed I/O systems.

The very real possibilities of a sophisticated system on a single chip are useful only if you can manage the resulting system complexity and produce a working system in a reasonable amount of time. A 20-million logic gate design started from scratch could easily take 200 engineers three to five years to architect, design, verify, and build. The common wisdom is that a product design cycle needs to be approximately one year to be competitive in the marketplace.

The value of SoC design is the promise to build on other people's work. To be successful, a design team needs to acquire a large portion of the design and software, and then add a relatively small proprietary portion that uniquely defines a product. Figure 1 includes a common architecture for a communication chip. The figure illustrates a design with multiple processors, a control interface to a host computer, a memory interface, a high-speed data interface, and other logic.

A good candidate for a SoC would be a product

that includes a proprietary processor, but uses standard I/O interfaces. Another candidate might use standard processors, but add a proprietary logic block, which either adds unique functionality or speeds up the overall processing effort. These blocks of logic, including the processor, the I/O interfaces, and the memory interface are often called IP cores, because each includes intellectual property (IP). This article explores many design and system challenges faced by a designer today, including decisions about chip architecture and how to program these complex systems.

DESIGN METHODOLOGY

Design flow and methodology is broadly defined as the tools and methods used by a design team to build their design. A simplified flow is shown in Figure 2. Although design flows vary, it is critical for a single design that the flow is crisply defined and well supported. An understanding of these methods is important to understanding overall SoC design.

The method of specification for a design today is almost always a Register Transfer Language (RTL), typically either Verilog or VHDL. These languages are similar to programming languages, but more restrictive. Synthesis programs read RTL and translate them directly to circuits that are implemented on the silicon. Synthesis programs have evolved to the point that designers trust the results. Simulation programs read RTL and let designers exercise functionality and prove or disprove the design correctness. This simulation flow is key to the design verification methodology.

Design tools exist to enhance every aspect of the SoC design process, including formal verification between a high-level specification and the final circuit level, timing verification, tools that evaluate the physical correctness of the design, and sophisticated software tool suites that allow test specification development to proceed quickly. New tool offerings allow designers to express expected states and values as assertions that are inserted as monitors into a simulation specification.

IP BLOCKS AND CORES

Many of the IP cores available today are provided either by commercial vendors or by an IP group within a large organization. Most IP cores are specified in a Register

Transfer Language and can come as either hard or soft. Soft IP indicates that the core is not mapped onto silicon; thus, not all of its physical or timing characteristics can be guaranteed.

A common definition of an IP core is a design function with well-defined interfaces. Many IP cores start as a design block for a specific chip that handles a well-defined piece of the functionality, and then evolves to a standard piece of functionality, of use in multiple chips.

Some examples of IP cores include I/O systems such as Infiniband, PCI-X, Ethernet, and commodity interfaces such as memory controllers. The difference between a block on a single design and an IP core is that the signal protocol and the clock definition can be significantly different between chips. An IP core needs to be flexible enough to handle variance.

Vendors who provide useful IP typically are successful at providing a design abstraction for their IP. In particular, standard interfaces for data, control, and clocking are key to their success. Attempts at standards include the AMBA bus interface or Open Core Protocol (OCP), but many cores still provide propriety interfaces.

GLOBAL ARCHITECTURE

Although the attractive part of SoC design is the ability to build on existing IP blocks, ultimately the chip is a system and must be based on a quality system design. The challenge is that, by definition, the IP blocks are completed with no system knowledge; thus, putting the system together can create many complications. The system designer must decide on an overall system design, create an overall communication strategy, and specify and implement global signal systems.

The global communication method for a chip becomes either a system bus or a proprietary set of paths between the IP blocks. Figure 3 illustrates four choices for system-level communication.

- The first is a crossbar switch that provides dedicated wires for communication between every IP core. Although effective, this choice is expensive in terms of real estate and can require development of specialized timing and signal protocols between different IPs.
- The second example illustrates a proprietary communication system that includes a single connection point for all communication. This example requires fewer

SoC

Software
Hardware
NIGHTMARE
or *Bliss*

wires, but has a single point that becomes the communication bottleneck.

- The third choice includes a proprietary communication system that chooses the communication links and builds a set of dedicated wires and pipelining states into the communication system.
- The fourth option is a dedicated system bus. Often, a system bus is the most effective choice, but it can be time consuming to design from scratch. A number of system buses are available as IP cores that take care of the overall system communication; examples include the AMBA bus from ARM, Sonics Inc.'s MicroNetwork, and the IBM CoreConnect bus. The advantage of these system buses is that they do provide a general shared communication resource. They assume, however, a signal and clock protocol that may not be available for the IP blocks for this chip.

It is common to spend substantial design time creating the interfaces between the existing IP cores and the system communication vehicle. One final observation is that a designer may want or intend to implement one interconnect architecture, but be forced to support an alternative by a key piece of IP. In fact, it is common

for systems to incorporate multiple I/O architectures—for example—to include a system bus, but have a few direct connections.

In addition to deciding about global communication, the designer must make global clocking and protocol decisions. One of the most challenging global decisions is the system clock. The simplest system choice is to have a single system-wide clock, but each of the IP blocks is designed to support a specific clock speed. Guaranteeing a specific clock speed for a soft IP is a challenge.

Different portions of a design often support or require different clock speeds. An example, shown in Figure 4, includes a processor core running at 250 MHz. The high-speed I/O requires a clock of 100 MHz, and this requirement is part of the I/O standard definition. The debug interface is slow, and requires a 1-MHz clock. The system designer may decide to have the system bus running at the CPU speed or may have other requirements that dictate a distinct speed of its own. In the worst case, a system designer must create unique clock domains for every core, thus creating significant additional synchronization design effort. A more frequently chosen solution is to use a

Examples of Different Clock Domains

common clock for many of the IPs and provide a unique clock domain for a few specific cores, thus minimizing the system-level design effort.

In addition to the clock specification, an IP core includes a data and control interface, or protocol interface. If design reuse were a primary objective of the IP core development, the representation of this data would be common among a large number of IPs—for example, the protocol defined by one of the system bus companies. A number of standards organizations have made headway in this area, especially the Virtual Socket Interface Alliance (VSIA). But the reality is that many, if not most, of the IP cores available today have a proprietary interface dictated by the original design. Some IP vendors include bridges for their popular IPs to one of the more common interfaces, such as the AMBA bus or OCP.

A systems designer must understand the definition and timing of all of the protocol signals, however. Many systems have failed because a read-enable signal is inverted in a part of the design, or where the control signal is assumed to be at the rising edge of the clock instead of the falling edge. For SoC designers the dream of a plug-and-play set of IP cores is just that. A significant design effort is always involved in plugging these IP blocks together.

SOFTWARE ISSUES

Although sophisticated silicon hardware is a challenge for all companies today, the real product challenge is at the system level, where current products include both hard-

ware and software. Many chip companies have been dismayed at the number of software engineers required for a particular system. It is often more than double the number of chip designers required. This section addresses some of the complexities faced by a systems designer developing an application program for the prototype system illustrated in Figure 1.

Let's assume for the moment that the processor on the sample chip is a standard one, such as those offered by MIPS or ARM, or is provided by an IP vendor

such as Tensilica, thus ensuring that the product developer does not need to create the application development environment. Even if a new chip can use an existing software development environment, however, a product architect needs to develop a programming model for the application developer. For example, one choice might be to run a standard operating system on each independent processor and provide complete application flexibility to the application developer. The application developer is then faced with the task of coordinating all activity among these processors. A more common approach is to provide the application developer with a model of a single processor with multiple threads, each mapped to a separate processor. This model restricts access to each processor, but also restricts the number of potential synchronization problems that may occur at the application level.

This programming model provides a common development environment only for the processors. In fact, the chip is a much larger system and includes complex communication systems that must either be modeled in the development environment or restricted to a single method that is defined for the application developer. In any case, the standard development environment is almost never standard because it must be extended to accommodate all of the features available on this new system. Thus, any product cycle must include both the application development phase and a non-trivial investment in development tools for use by the application developer.

In an embedded device that is using a SoC, one of the

SoC

Software
Hardware
NIGHTMARE
or *Bliss*

prevalent problems is with device drivers. The IP cores that were selected to build up the system affect which device drivers will be used, and the memory architecture that ties the cores together dictates the communication mechanisms throughout the system.

In embedded parlance the set of device drivers for a board is called the Board Support Package (BSP). A number of embedded OS companies already provide BSPs for popular boards, and in the future these will certainly appear for the more common SoCs. Although this might seem comforting at first to those who have a SoC project dropped on their desks, it is unlikely to be of much help. Part of the power of the SoC paradigm is the ability to glue together disparate devices to create a whole new product. This means that while there may be a prebuilt set of device drivers for a reference design, it is unlikely that the product developers can use the reference design intact in their system; in fact, significant changes are almost always required. Reference designs are just that, references; they are rarely used in final products.

As it becomes easier to build SoCs, each one will be different from the others, in small ways at first. Eventually, these differences will grow and will cause most of the headaches for designers, implementers, and integrators.

The key to solving at least part of this problem is the proper design of interfaces. The current techniques used to specify and implement software interfaces are far from the ideal of plug and play. Although most operating systems provide a device-driver-interface standard, each one is slightly different, making driver reuse next to impossible. Even systems with a common lineage, such as the FreeBSD, NetBSD, and OpenBSD projects, cannot share drivers at the binary level.

A further complication is that the memory model used by device drivers and OS kernels often use shared memory buffers to increase speed. The product software will define a specific area in memory as shared between multiple software components, a use model commonly

called DMA, or direct memory access. The top-level program will place a block of data into a buffer, and then hand a pointer to this buffer to another software component (for example, the network stack). Having different components own the same memory in this way does not work well in a system that can be suspended or reconfigured at any time. Tracking and reclaiming memory in such systems is difficult and is almost always handled via ad-hoc mechanisms.

The last problem confronting the person doing “board bring-up,” as this process is called, is that debugging support is often minimal. It is not uncommon to debug a system with an oscilloscope in the very early stages, and print statements replacing standard debuggers. Today, more advanced tools, such as source-level debuggers that talk directly to the CPU socket (JTAG) and software oscil-

The brand new hardware and brand new software are **interrelated**, and often a **problem** in one can manifest itself as an apparent problem in the other. This leads to **finger pointing** and disagreement between the chip designer and software developer.

losopes that show the execution of tasks over time, can simplify the board bring-up process. Unfortunately, these tools often depend on the undebugged device drivers actually working. Thus, the brand new hardware and brand new software are interrelated, and often a problem in one can manifest itself as an apparent problem in the other. This leads to finger pointing and disagreement between the chip designer and software developer.

These problems confront not only the person building a product around a SoC, but also the company that wishes to sell its IP cores. Unlike the desktop world, in the embedded world a plethora of options is available for operating systems to support, and half of the embedded market still develop their own. Which device drivers, if any, should IP core manufacturers provide to their customers? A system's success is largely based on the availability of software drivers for the latest devices. This is the real reason for Linux's preeminence among the open-source operating systems: You can find a driver for just about anything you buy.

Other models exist for handling this device complexity, one of which has been used by Apple in its OS X

operating system. Apple has the luxury of controlling the internal hardware of its boxes and having to communicate with external devices only over well-specified hardware interfaces (USB, FireWire, Ethernet, and 802.11). Its method was to completely redo the device layer for one of the open-source Unix versions (FreeBSD). Clearly these solutions will not work for the SoC world in general, though certain vendors with narrow market niches may take advantage of it.

Another way to handle the complexity of communication within software is to build software based on a message-passing system, but message-passing models have not made significant inroads in the software community. In message passing, every communication between two software modules is completed using a message protocol, such as a network protocol. This creates very strong interfaces but can present performance problems. This model of programming is also not very common, so hiring engineers who write message-passing code is not easy. Only one commercial embedded OS uses message passing, and none exists in the open-source world.

Software for SoC-based designs has its own set of problems, as explored in this section. Until these problems are solved, each SoC-based design will require a large investment in device driver software to make a product successful.

CHALLENGES TO OVERCOME

Although the challenges presented here are real, the truth is that small and large companies are creating SoC designs

on a regular basis. IPs supporting common I/O systems, such as an Ethernet MAC, USB, and PCI, have become commodities. CPUs from MIPS and ARM are used on a regular basis in many designs.

Complex designs are a reality and the standards are trying to catch up with them. This is an exciting time with an increasing number of possibilities. The problems with IP standards and software interfaces are real, and they are stunting the IP and embedded markets. SoC design is too compelling for building ever-more complex devices, so it is likely that very smart people will figure out what needs to happen, and make it so. Q

TELLE WHITNEY, PH.D., is the president and CEO of the Institute for Women and Technology. Before joining the institute, she was vice president of engineering and part of the founding management team at Malleable Technologies, a start-up in the programmable communication area, and she served as director of software at Actel Corporation. Whitney cofounded the Grace Hopper Celebration of Women in Computing (GHC) conference, the largest technical conference for women in computing. She is secretary/treasurer of ACM. **GEORGE NEVILLE-NEIL** works on networking and embedded operating systems in various environments. His interests are network protocols, operating systems, and component-based computing.

© 2003 ACM 1542-7730/03/0400 \$5.00

BIO

acm
queue
tomorrow's computing today

Coming in May
Wireless