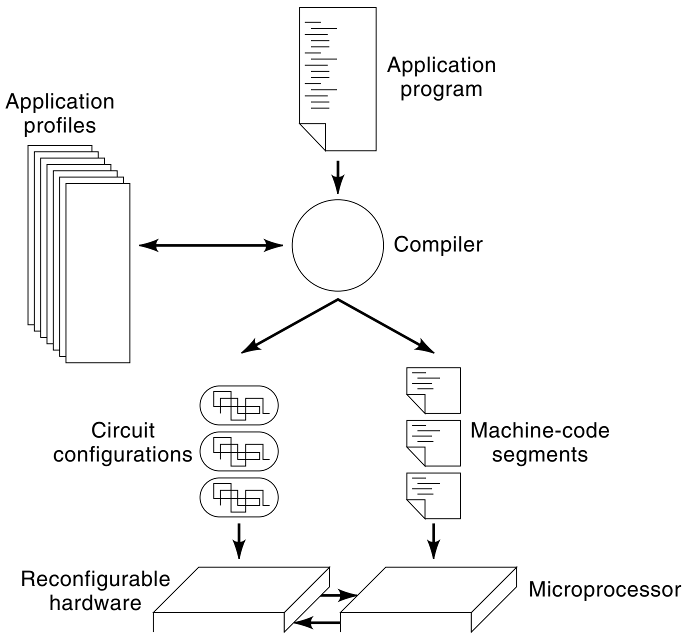

Figure 1. A configurable computing system with an advanced compiler that automatically generates machine code for the instruction-set processor and circuit configurations for the reconfigurable hardware.

CONFIGURABLE COMPUTING

Configurable computing refers to the process of employing reprogrammable hardware and interconnect to enhance the capabilities of traditional computing systems. The concept and power of configurable computing have been recognized and explored by many researchers for over a decade. Nevertheless, there is compelling evidence that recent advances in this area may impact the very foundation of modern microprocessors, or processors in general, and the way we program them. By the early 21st century, it is conceivable that future microprocessors, with several tens or hundreds of millions of transistors, will incorporate configurable logic arrays as a primary component. The integration of reconfigurable logic will pave the way for a new wave of dynamically transformable microprocessors, with dynamically alterable instruction sets and hardware resources that enable the specialization of instructions and hardware configurations to optimize the mapping of specific applications. The reconfiguration process is aimed at eliminating many of the processing bottlenecks imposed by the fixed hardware structure of current processor architectures.

Programming such powerful processing engines requires new types of compilers that can be integrated with conventional compiler technology. These new compilers must be capable of profiling applications and selecting a set of hardware configurations and library routines that best accelerate the applications subject to certain restrictions on the utilization of system resources. Ultimately, such compilers may evolve the capability of automatically generating instruction sets, hardware configurations, and correct code sequences for transformable processors starting with high-level language programs or specifications of the application. This type of compiler-architecture interaction is illustrated in Fig. 1.

This article presents an overview and survey of configurable computing trends and technologies. We will start by reviewing the technology that harnessed and motivated the rapid evolution of configurable computing, namely run-time field-programmable gate arrays (FPGA). We then review earlier work on FPGA-based transformable coprocessors and finally proceed to contemporary notions of FPGA-coupled microprocessors and configurable computing in general.

From a historical perspective, the notion of reconfigurable hardware systems is attributed to Estrin (1). However, much of the subsequent research on configurable computing appeared in the context of parallel-processing architectures (2–9). This work started around the mid-1980s and continued to be a rich source of techniques for ultrafast algorithms for arithmetic computations, image processing, sorting, searching, and a host of other applications. We devote the second part of this article to an overview of these efforts, especially because they contribute to understanding the power and limitations of reconfigurable processor arrays.

FPGA-BASED CONFIGURABLE COMPUTING

The FPGA was introduced in 1986 for designers requiring a solution that bridged the gap between programmable array logic (PAL) and application-specific integrated circuits (ASIC). In the late 1980s and early 1990s independent researchers throughout the world started demonstrating that computationally intensive software algorithms can be transposed directly into FPGAs for extreme performance gain. This continuing research and a growing commercial sector use of FPGAs have spawned numerous developments in the area of high-performance computing.

The term *configurable* (or *transformable*) *computing* refers to the process of dynamically reconfiguring field-programmable custom-computing machines to adapt quickly to varying algorithm and operating conditions under the control of a host processor. Transformable computers are those machines that use the reconfigurable aspects of FPGAs to implement an algorithm. The current state of development regarding the use of FPGA devices and the systems developed is a testament to the potential of this technology. Many computers have been designed using FPGAs to accelerate the prototyping process, and several computer systems use FPGAs in place of custom ASICs as a standard design practice.

The dynamic reconfigurability of static random access memory-(SRAM) based FPGAs provides a very flexible platform for implementing new types of coprocessor systems whose architecture can be *transformed,* or reconfigured, during run time, to realize different types of functions. The coprocessor is normally attached to the bus of a host system running high-level software. Alternatively, the reconfigurable coprocessor can be integrated with the microprocessor on the same chip, thus eliminating the bus and I/O interface bottlenecks. In a multitasking environment, a program executing on the host processor can allocate tasks dynamically to the transformable coprocessor. Reconfigurable computer systems are normally characterized by a high level of hardware concurrency and flexible routing channels that interconnect hardware modules. Therefore, time-consuming computational "loops" are offloaded from the host processor and allocated to the FPGA-based coprocessor, which employs optimized programmable hardware to execute these tasks. Because large programs involve multiple tasks that are executed in a given order, the FPGA-based coprocessor needs to be reconfigured to implement the hardware blocks required for executing the present task only. The frequency of coprocessor reconfiguration depends on a number of factors including the size of hardware blocks, reconfiguration loading time, speed of the host-coprocessor interface, and other application-specific factors.

This new use for reconfigurable logic device technology received a major endorsement in 1996 with the announcement of a Defense Advanced Research Program Agency (DARPA) funded program called Adaptive Computing. Whether it be adaptive, chameleon, or reconfigurable, growing interest in utilizing FPGAs in computing systems furthers the probability that transformable computers are the next frontier in computer architecture evolution.

Many contemporary transformable computers are designed with fine-grained parallel (systolic) computation in mind. Other systems have different design goals. The P4 Virtual Computer, developed in 1987, was designed as a vector style numeric processor (10). The DVC, from Virtual Computer Corporation (US), was designed to perform mostly symbolic processing. Other forms utilize FPGAs as high-speed communications agents as in ArMen, designed by researchers at Universite de Bretagne Occidentale (France). ArMen is a hybrid system consisting of linear asynchronous transputers. In this system, FPGAs are used to configure high-speed systolic communications agents between processors achieving improvement in data processing in excess of two orders of magnitude over conventional software methods (11).

FPGAs for Reconfigurable Computing

In many respects, FPGAs are attempting to provide an alter native to ASICs in providing highly customized fast hardware for specific applications that cannot be handled adequately by a traditional microprocessor. Figure 2 contrasts three types of systems based on FPGAs, ASICs, and microprocessors. The microprocessor is a general-purpose computing machine. It implements different functions by means of changing a sequence of machine instructions. On the other hand, an ASIC is a specialized, self-contained system that reads data operands from an external memory and performs a sequence of dedicated functions on the data before producing a result. The ASIC hardware is fixed and highly specialized to execute a single or a limited number of functions. Additionally, the ASIC I/O is strictly tailored to specific data formats. The ASIC controller is contained within the chip so that the ASIC can be controlled by very few external control signals that specify the type of function to be performed and some information about the interface and data formats. By contrast, FPGAs have reprogrammable hardware. An FPGA is normally interfaced to two types of memories: a data memory and a configuration memory. The configuration memory contains several hardware configuration files that are loaded into the FPGA according to the task flow specified by the application. The configuration files can specify either control circuits or computation (data-path) slices. In other words, the FPGA imports both its *control function* specification and its *data-path setup* from the configuration memory, thus offering a very flexible reconfigurable hardware platform for implementing specialized systems. However, the flexibility of FPGAs comes at the cost of smaller gate capacity and slower hardware speeds as compared with equivalent ASIC chips.

Figures 3 through 5 compare the mapping and execution of the same sequence of tasks on a hypothetical ASIC and a hypothetical FPGA. Figure 3 shows the relative hardware resources required for implementing each task, as well as the relative execution times. Several points of difference can be observed by studying Figs. 4 and 5. Here, an application requires the execution of three types of tasks, labeled A, B, and C, according to the task flow graph given in Fig. 4. The notation B1, B2, B3, is used to indicate different instances of activating the same task (B) on different sets of data. Similar notation is used for tasks A and C. It is assumed that the hardware resources of the ASIC are capable of executing two instances of each of the A, B, and C tasks simultaneously. The FPGA is assumed to be much more resource-limited and is capable of executing a single instance of task B (i.e., implementing task B consumes most of the FPGA hardware), two instances of task C, two instances of task A, or one instance of task A with task C. Finally it is assumed that any of the tasks requires the same execution time on either the FPGA or the ASIC. Figure 4 shows the ASIC and FPGA schedules for executing the different tasks. Figure 5 shows the various stages of task execution on the ASIC versus that of the FPGA. Note that the ASIC operation involves activating one or more dedicated hardware blocks at a time. In the FPGA, only the circuit configurations of current active tasks are loaded into the FPGA. The schedules of Fig. 4 show that, despite its limited hardware resources, the overall FPGA execution time is not much longer than that of the ASIC. Observe that the FPGA schedule (Fig. 4) does not show the FPGA configuration time during task swapping. However, the ratio of configuration time to execution time is relatively small for most nontrivial tasks. Also, modern FPGAs have configuration times on the order of a few microseconds.

Current FPGAs consist of an array of uncommitted (but field-programmable) logic blocks and programmable interconnect resources. Although many different types of FPGAs are currently available, only static random access memory (SRAM)–based reprogrammable FPGAs provide a true imple-

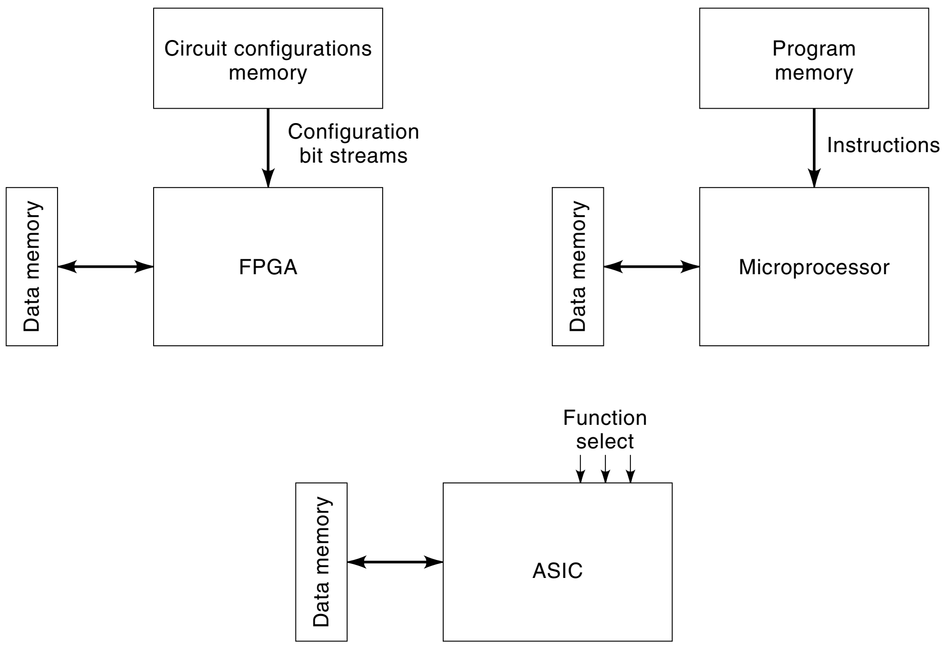

Figure 2. Three types of computing systems: a microprocessor system, a FPGA-based system, and an ASIC-based system.

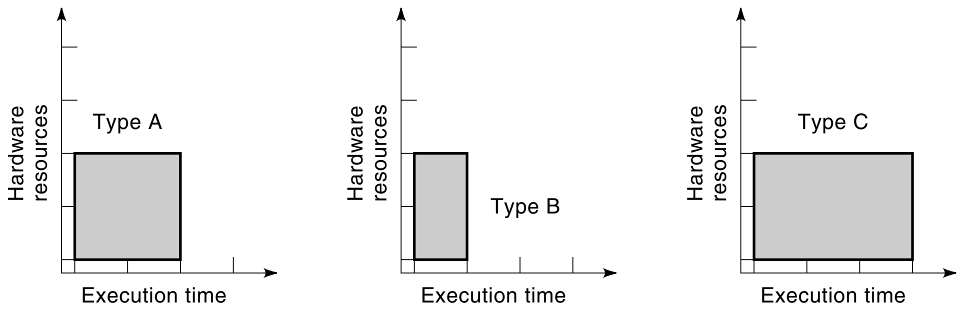

Figure 3. Relative hardware versus execution time for three types of tasks.

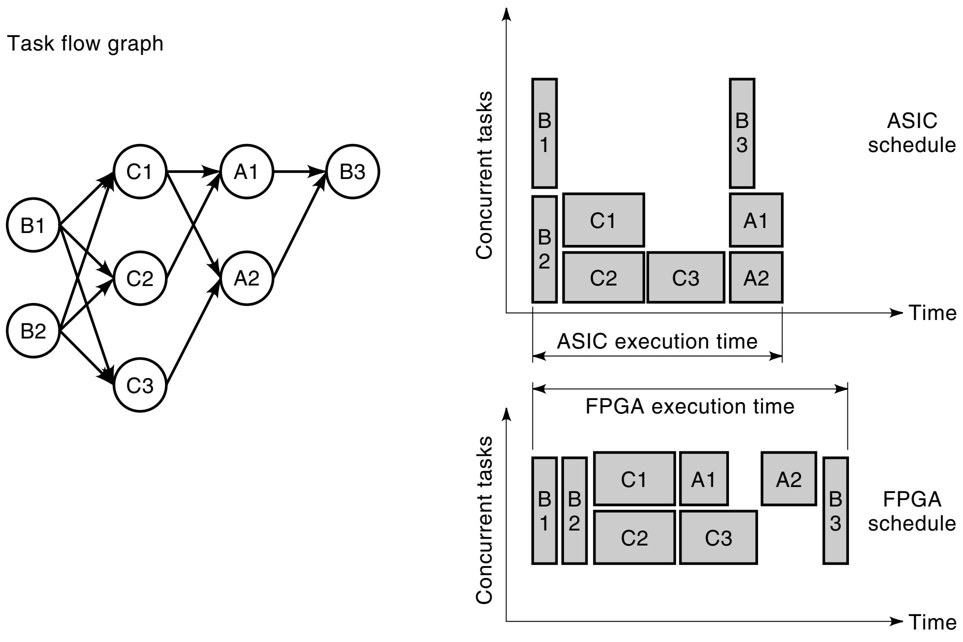

Figure 4. Task flowchart and execution schedules for hypothetical ASIC and FPGA systems.

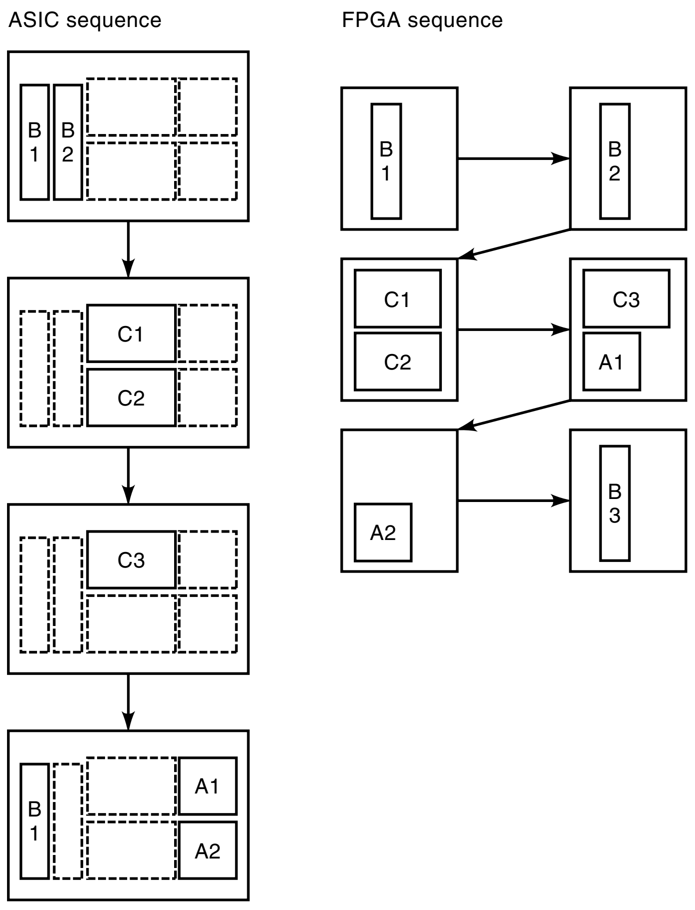

Figure 5. Task activation on an ASIC, and task reconfiguration on an FPGA.

mentation technology for reconfigurable computers. Prime examples of this technology are the Xilinx family of SRAM-based FPGAs (12,13), AT&T's ORCA series (14), and the more recently proposed multicontext FPGAs. The following sections consider some of the popular FPGA families as well as some of the promising emerging reconfigurable computing technologies. For pedagogical reasons, we start by describing the Xilinx XC4000 FPGA, which is a very good representation of the type of FPGAs used for implementing a number of transformable processors.

The Xilinx XC4000 FPGAs

The XC4000 structure is shown in Fig. 6. The major components of this FPGA are configurable logic blocks (CLBs), input-output blocks (IOBs), and switch matrix blocks (SMBs). All these structures are connected by wire segments of varying lengths, as shown in Fig. 7. Xilinx uses complementary metal oxide semiconductor (CMOS) SRAM technology to store the programming information for the FPGA. SRAM cells distributed around the FPGA are used to program specific functions in the CLBs and define the interconnectivity among the CLBs through the switch matrices. After powering up the FPGA circuits, "bit files" carrying configuration information are loaded into the SRAM cells. For this purpose the SRAM cells sprinkled around the FPGA chip are linked into a long shift register, and loading configuration bit files is done by shifting in strings of zeros and ones through I/O pins. The Xilinx technology is characterized by fast reprogrammability. The functionality of the FPGA can be altered dynamically by shifting in new configuration files.

The FPGA logic and interconnect can be programmed by loading the proper bit values in the SRAM control bits. SRAM bit control is achieved by using two different techniques. The first technique is used to set up the appropriate bits for building programmable lookup tables, which are used to realize logic functions on input data. The second technique uses SRAM bits to control multiplexing or demultiplexing logic and pass transistor circuits like those shown in Fig. 8. Figure 9 shows the pass transistor circuit for a reconfigurable interconnect switch, and Fig. 10 shows a 4×4 SMB realization using 16 copies of the switch of Fig. 9.

The P4 Virtual Computer system (10), SPLASH (6,11), and PAM (4,15), are a few configurable computing systems implemented with Xilinx FPGAs. The Virtual Computer P4 system uses over 50 of the XC4010 chips placed on a single board with additional ICUBE field-programmable interconnect devices to provide wide communication paths among the FPGAs. The overall system contains over 520,000 gates, making it one of the largest reconfigurable systems to be built up until the early 1990s. The DVC transformable coprocessor from Virtual Computer Corporation is another system based on a single XC4013 FPGA with additional memory. The DVC board is designed for interfacing with the SBUS of a Sun SPARC workstation.

The Xilinx XC6200 FPGA

The Xilinx XC6200 FPGA marks a significant departure from previous FPGA generations in several aspects, including, technology, logic-block granularity, and areas of application (13). The XC6200 family is based on a fine-grained (sea-of-gates) register-rich cell structure, with a low-delay hierarchical routing scheme that flexibly implements local and global interconnection among logic cells. The larger number of registers in the XC6200 FPGA is well suited for computationally intensive data-path applications.

The XC6200 family provides a truly powerful and very flexible platform for implementing transformable coprocessors in particular and a host of other reconfigurable architectures in general. The XC6200 architecture features a built-in processor interface to facilitate the implementation of reconfigurable coprocessors in embedded system applications. The built-in interface distinguishes the XC6200 family from previous generations of FPGAs (such as the XC4000 series). In the XC4000 series, the interface to the main processor bus must be implemented using programmable logic resources in the FPGA, and that may consume a significant portion of the FPGA resources. More importantly, the XC6200 interface provides high-speed access to all internal registers in the logic cells, that is, any register can be mapped into the memory address space of the host processor, allowing fast data transfers using simple hardware. In general, internal FPGA architectures are not always optimized for the data-path algorithms typical of coprocessing applications. The XC6200 FPGA is one of the first commercial products to address this problem effectively. In the following, we take a more detailed look at the XC6200 architecture, emphasizing its role as a transformable coprocessor.

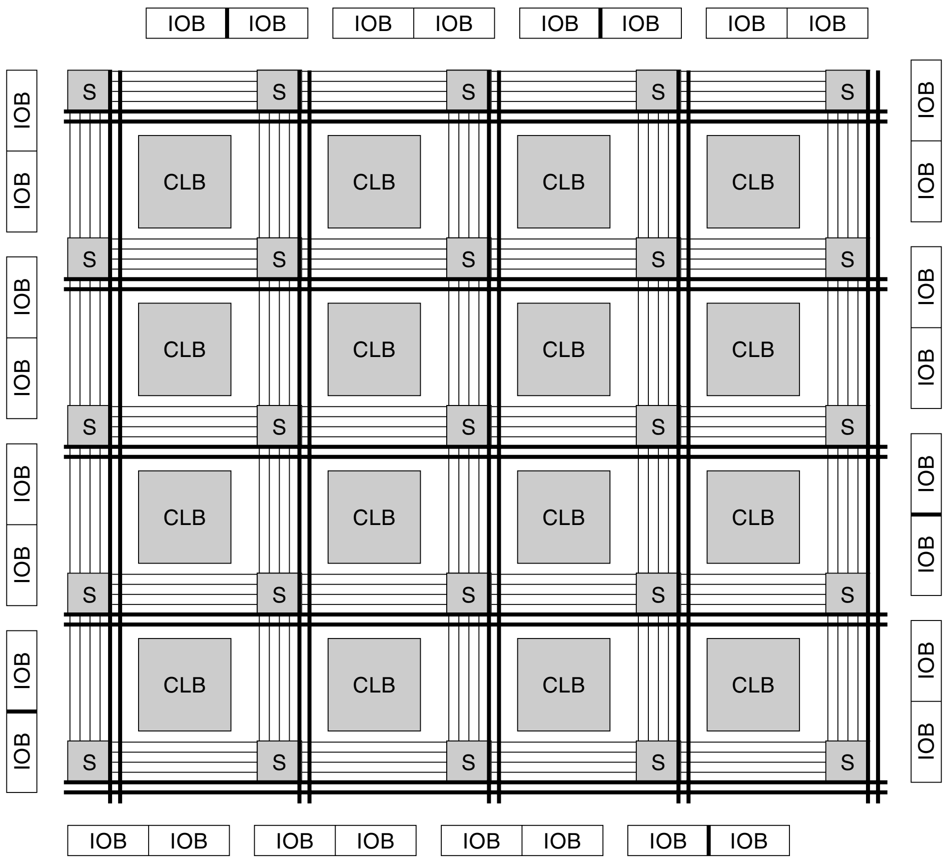

Figure 6. The Xilinx XC4000 FPGA structure (CLB denotes a configurable logic block; IOB an input/output block, and S a switch matrix block).

XC6200 Architecture. The XC6200 FPGA is equipped with simple function units and abundant hierarchical routing interconnect resources. The XC6200 FPGA is arranged as a hierarchy of *cells,* blocks of cells, blocks of blocks of cells, etc., with each level in the hierarchy having its own routing resources. At the lowest level of the hierarchy, neighbor-connected cells are grouped into blocks of size 4 × 4 cells. At the next level of the hierarchy, 16 of the 4 × 4 blocks are grouped in a 4 × 4 array to form a 16 × 16 block, as shown in Fig. 11. In the XC6216 FPGA, the 16 × 16 blocks are grouped in a 4 × 4 array to form a 64 × 64 block of logic cells. At each level of the hierarchy, blocks are interconnected by wires of appropriate length. The XC6200 FPGA employs wires of length 1 (cell), length 4, length 16, and chip length for the 64 × 64 block. Thus, each level of the hierarchy has its own routing resources. Each of the basic cells consists of a function unit as well as a reconfigurable switch capable of realizing any interconnection pattern among the cell ports. The detailed structure of a cell is shown in Fig. 12. Two sets of input multiplexers are used to connect a cell to its four nearest-neighbor cells and to the adjacent 4 × 4 blocks. The inputs from nearest-neighbor cells are labeled N, S, E, and W, corresponding to neighbor cells, respectively, to the north, south, east, and west of the cell shown. Inputs from cells connected to the shown cell by length-4 wires are labeled N4, S4, E4, and W4. Inputs from cells connected by wires of length 16, or even length 64, are also available as inputs. However, such inputs are not shown in Fig. 12 to maintain clarity. The *Magic* output in each cell provides an additional routing resource but is not always available for routing. The role of Magic outputs will be explained in more detail below. The function unit shown in the center of the cell of Fig. 12 is implemented using the logic circuit of Fig. 13. Clock and clear functions are required for the correct operation of the D flip flop in the function unit. Despite its simplicity, a function unit is capable of realizing over 50 distinct logic functions.

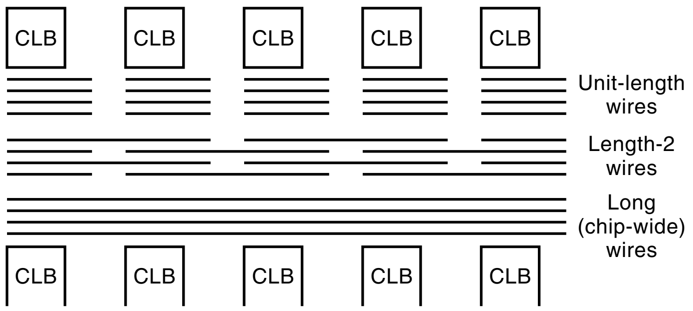

Figure 7. Wire segments of different lengths are available in the XC4000 FPGA to realize local and global interconnections among CLBs.

To support hierarchical routing resources in the XC6200 FPGA, additional *boundary switches* are provided around the periphery of larger blocks of cells. Figure 14 shows how boundary switches are placed around a 4 × 4 block. A cell's Magic output is routed to two distinct boundary switches. The Magic wire can be driven by one N, S, E, or W input from an adjacent cell or from the N4, S4, E4, and W4 signals passing over the cell. The Magic output is particularly useful for mak-

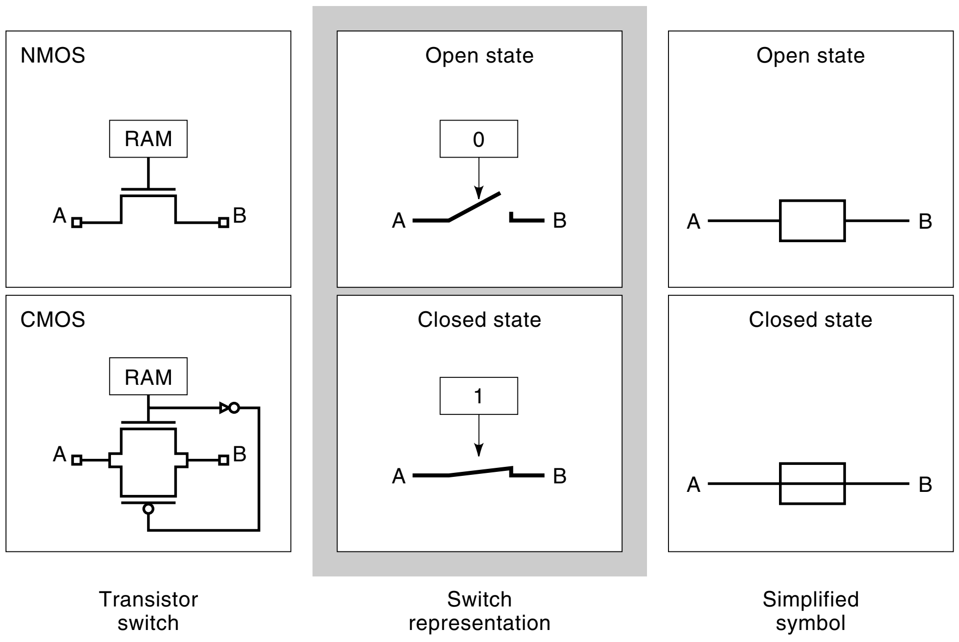

Figure 8. Pass-transistor switch controlled by a RAM configuration bit, NMOS = n-channel metal oxide semiconductor; PMOS = p-channel metal oxide semiconductor.

ing large buses turn around corners, as illustrated in Fig. 14. It is also useful for allowing the cell outputs to jump to the boundary switches of a 4 × 4 block and onto longer wires to other 4 × 4 blocks.

I/O Architecture of the XC6200 FPGA. The XC6200 FPGA employs user-configurable input/output blocks (IOBs) to provide the necessary interface between external package pins and the internal logic circuits. Basically one IOB is provided for every cell position around the array border. For example, 64 IOBs are provided along each of the four borders of a 64 × 64 block of cells. However, the number of IOBs is larger than the number of I/O pads available for the package, and therefore some IOBs will remain padless. The XC6200 FPGA incorporates a powerful I/O feature in that every IOB can route either a cell-array signal or a control logic signal to and from the device pin. This implies that all control signals can be routed into the cell array and incorporated in user designs. By the same token, user logic outputs can be used in the XC6200 internal control circuits. For example, a user-generated signal can be used to drive the internal chip-select (CS) signal rather than the CS pin on the package. Figure 15 shows how the interface circuitry between an XC6216 FPGA and a microprocessor can actually be placed within the FPGA. This greatly simplifies board design by eliminating the need for interface "glue" logic circuitry normally implemented by additional logic-array packages.

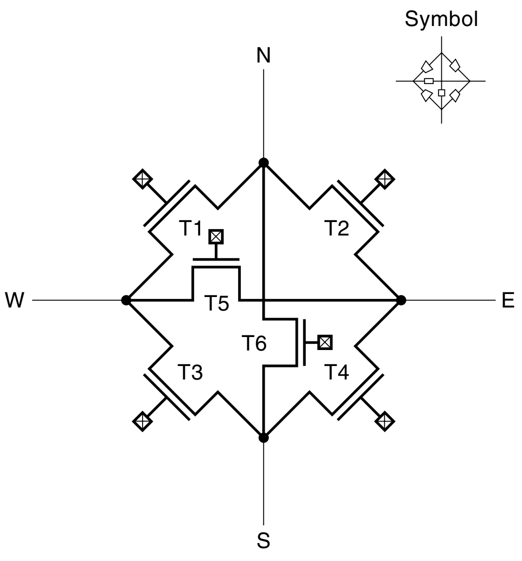

Figure 9. A four-port reconfigurable switch—the key element of any reconfigurable interconnect. Any subset of ports (N, S, E, and W) can be interconnected by closing one or more of the transistor switches (T1–T6) as shown in the example configurations.

The XC6200 FPGA as a Transformable Coprocessor. Several flexible and fast reconfiguration capabilities of the XC6200 FPGA make it suitable for realizing the concept of a transformable coprocessor. As a part, the XC6200 FPGA can be used as a microprocessor peripheral or as an application-specific device. When used as a microprocessor peripheral, the XC6200 interface contains the same data, address, and control signals as a conventional SRAM device. When used as an application-specific device, the XC6200 FPGA may require only user-defined I/O signals. The block diagram of Fig. 16 presents the cell array and I/O layout for the XC6216 part. Larger arrays can be constructed by tiling several XC6200 parts together. In some cases data and address buses may have to be used on every part of the large array. In the XC6200 FPGA, the control signals use every other IOB, leaving evenly distributed IOBs for interconnecting adjacent XC6200 chips.

If the XC6200 FPGA is to be used as a transformable coprocessor, the host-processor program must be designed so that it can interact with the design running on the FPGA. In this regard, the XC6200 FPGA provides several advanced processor-compatible features such as the following.

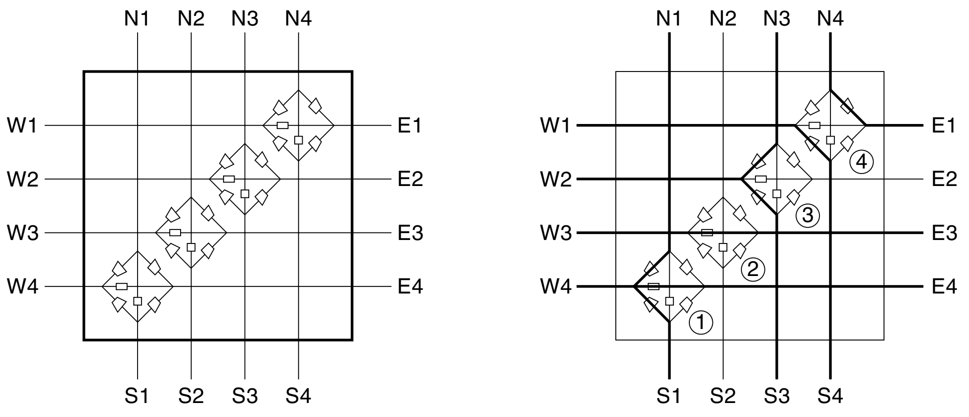

Figure 10. A simple switch matrix block (SMB), like that used in the XC4000 FPGA, can be configured to realize a large number of interconnections among its ports. The connected groups of ports are (W1,S4), (E1,N4), (W2,N3,S3), (W3,E3), (W4,N1,S1).

- Direct processor read and write access to all internal registers in the FPGA with no logic overhead, and support for 8-, 16-, or 32-bit data bus width. The XC6200 FPGA offers a flexible mechanism for mapping all the possible cell outputs from a column (in the cell array) onto the 8-, 16-, or 32-bit external data bus. This is illustrated in Fig. 17. It should be noted that the cells producing the outputs need not be adjacent. However, the output bits must appear in descending order of significance within a column of cells.
- All user registers and SRAM control memory are mapped onto the host-processor address space. In other words, the various registers within an XC6200 design appear as locations within the processor memory map. Also, the configuration memory of the FPGA appears within the processor memory map. Therefore portions of the XC6200 FPGA can be configured under the control of the host processor.

The features just noted demonstrate how the XC6200 family of FPGAs bring the concept of transformable coprocessors closer to reality. However, two major hurdles remain to be conquered before transformable coprocessors become a widely accepted concept. First, it is still time consuming to develop highly optimized custom hardware configurations for specific applications. Second, the process of deciding on how to partition task executions between the host and the coprocessor is mostly ad hoc and based on the user experience. The former problem is likely to become less serious as predefined device drivers and efficient run-time libraries of components for FPGAs continue to be offered by vendors. The latter problem, however, is more difficult and requires the development of intelligent compilers that are capable of optimizing the partitioning of tasks among the host processor (for execution in software) and the transformable coprocessor (for execution in hardware). This problem is harder than it may initially appear because the compiler must keep track of the state and gate usage of the transformable coprocessor, and it must also be aware of the specifics of the coprocessor performance.

DYNAMICALLY PROGRAMMED GATE ARRAYS AND MULTICONTEXT FPGAs

One problem with current FPGA architectures is the speed of reconfiguration. In such FPGAs, the function of a logic block, or logic cell, remains fixed between relatively slow reconfiguration sequences. This is caused by the time-consuming operation of loading configuration bit files from off-chip memory.

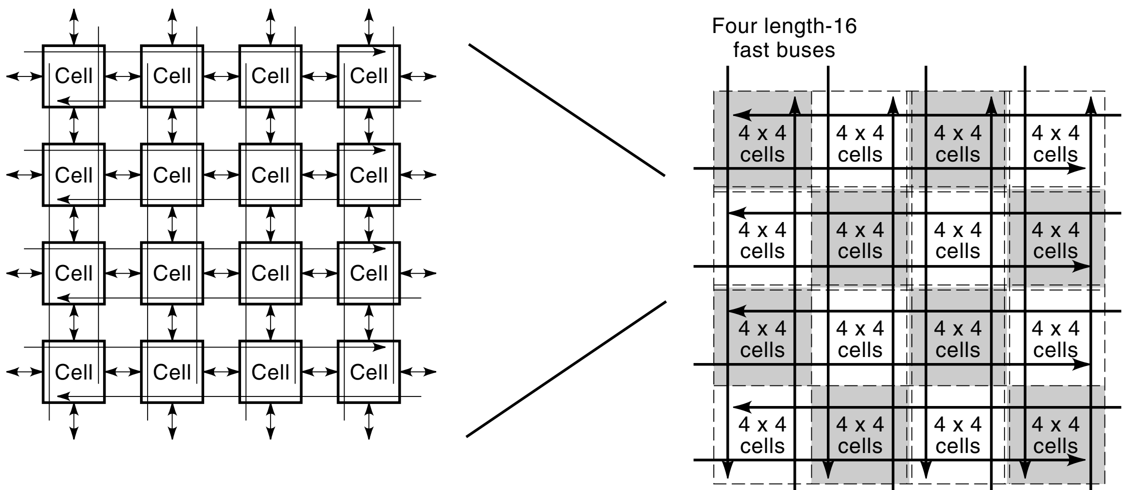

Figure 11. The basic layout of an XC6200 FPGA showing the hierarchical structure of logic cells and interconnects.

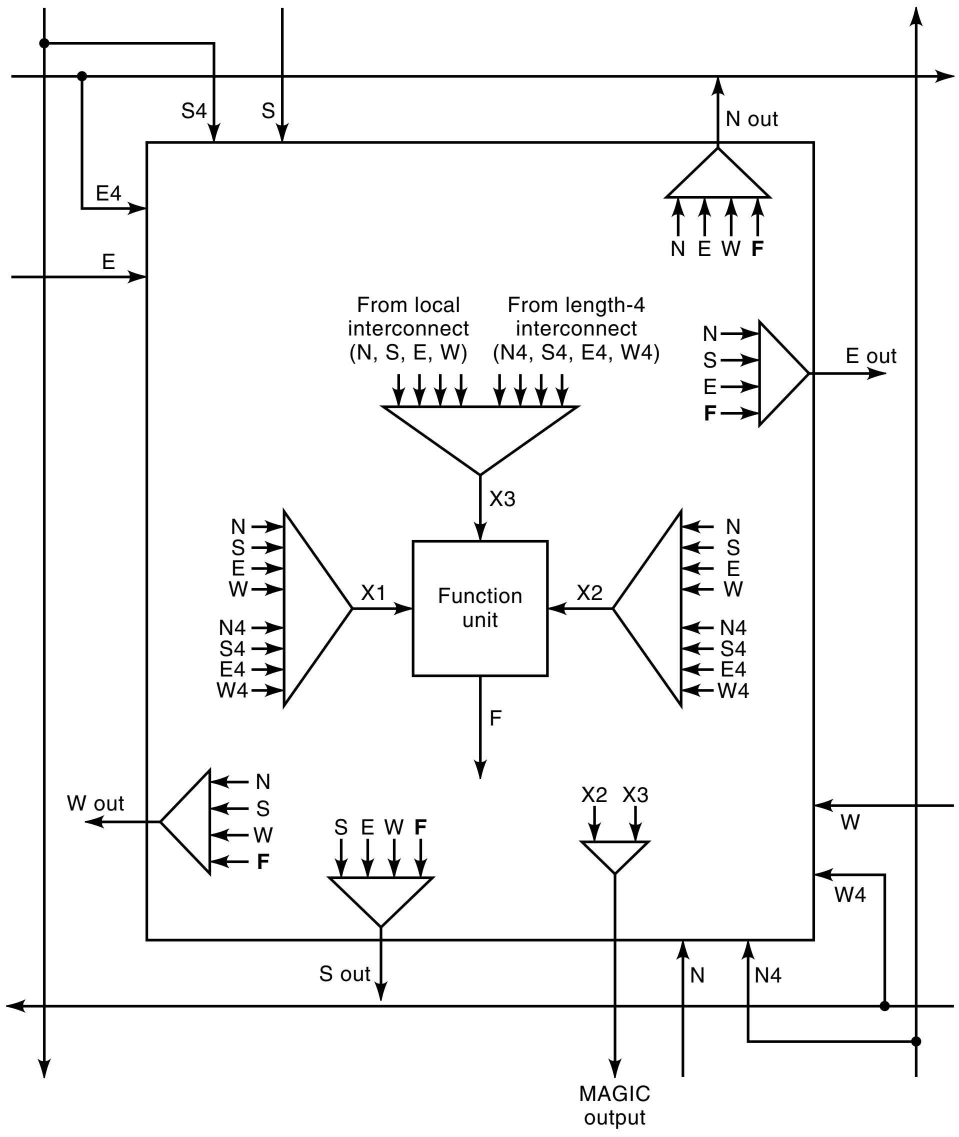

Figure 12. Basic cell structure in the XC6200 FPGA (for clarity, only a subset of the interconnect is shown).

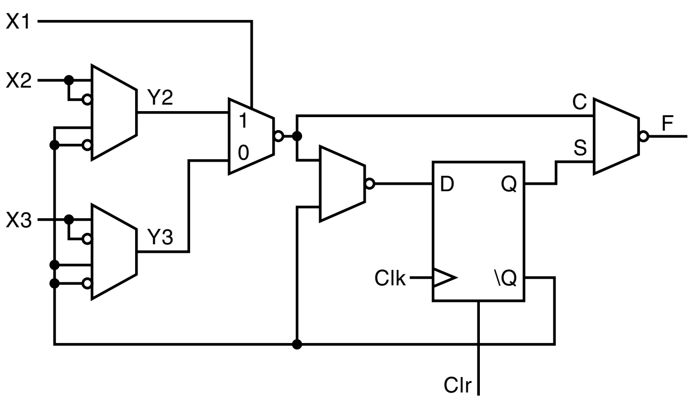

Figure 13. The XC6200 function unit consists of a simple logic circuit with one flip-flop and several configurable multiplexers. The SRAM bits that control the multiplexers are not shown for clarity.

Dynamically programmed gate arrays (DPGAs) present an enhancement over standard SRAM-based FPGAs towards realizing highly efficient configurable computers that are capable of changing a portion or all of their internal configuration on a clock-cycle by clock-cycle basis. A DPGA provides on-chip memory to allow multiple configurations to be stored in several memory banks within the chip. An application can store multiple customized array configurations into the same DPGA and switch rapidly (within one clock cycle) and dynamically among these configurations. This allows the DPGA to be reconfigured using its own local memory, thus eliminating several bottlenecks caused by limited I/O speeds and external memory access. With this method, full or partial DPGA reconfiguration can be achieved in one clock cycle, which is in the order of several tens of nanoseconds. In comparison, reconfiguration of the fastest current FPGAs requires a few microseconds. A DPGA is also called a *multicontext FPGA,* indicating the fact that a DPGA is an FPGA that can switch among multiple contexts (i.e., configurations) stored in its on-chip memory.

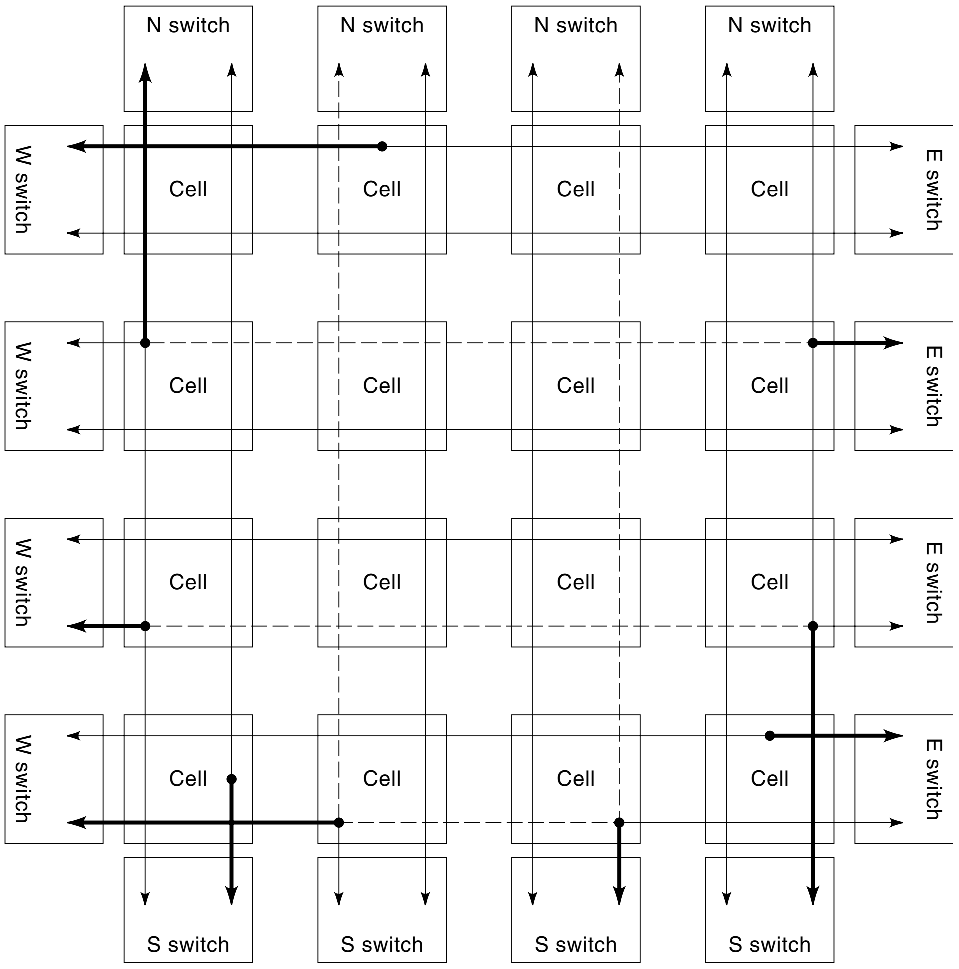

Figure 14. An XC6200 4 × 4 block with boundary switches for enabling global interconnects among blocks. The Magic outputs within the cells are used to enable long buses to turn corners within a cell.

The basic unit of the DPGA is an *array element,* which is basically a look-up table (or LUT) with a memory block that stores multiple configurations or contexts. Figure 18 illustrates the architecture of a DPGA array element based on the prototype reported in Ref. 16. The context decoder selects the appropriate configuration for the LUT from the memory based on a global *context identifier* distributed to all array elements. The DPGA employs a two-level routing architecture. At the lower level, array elements are grouped in square subarrays, with horizontal and vertical interconnects enabling communication among array elements in the same row or the same column of the subarray. At the higher level, neighbor-to-neighbor interconnection among subarrays is achieved by large crossbar switches.

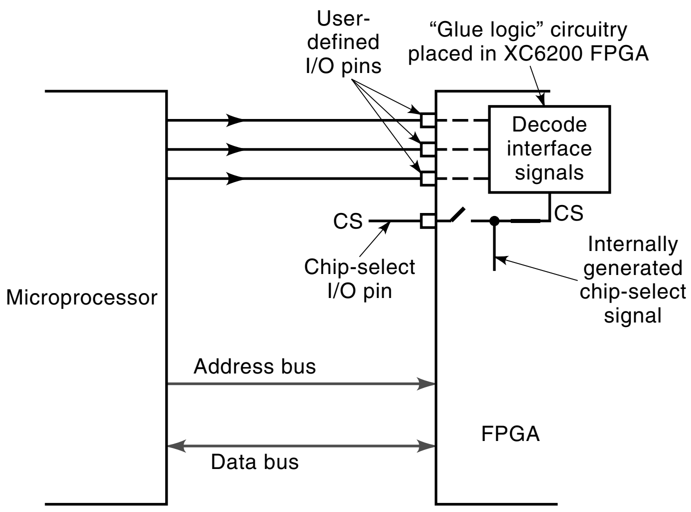

Figure 15. Microprocessor-FPGA interface.

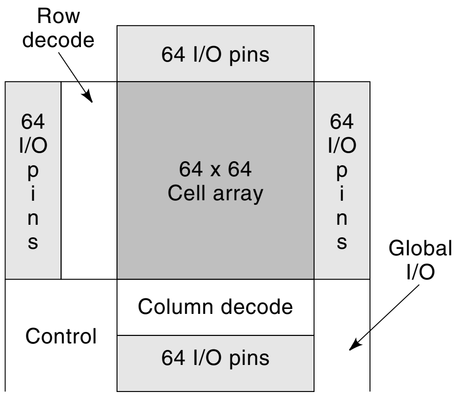

Figure 16. Cell array and I/O layout for the XC6216 part.

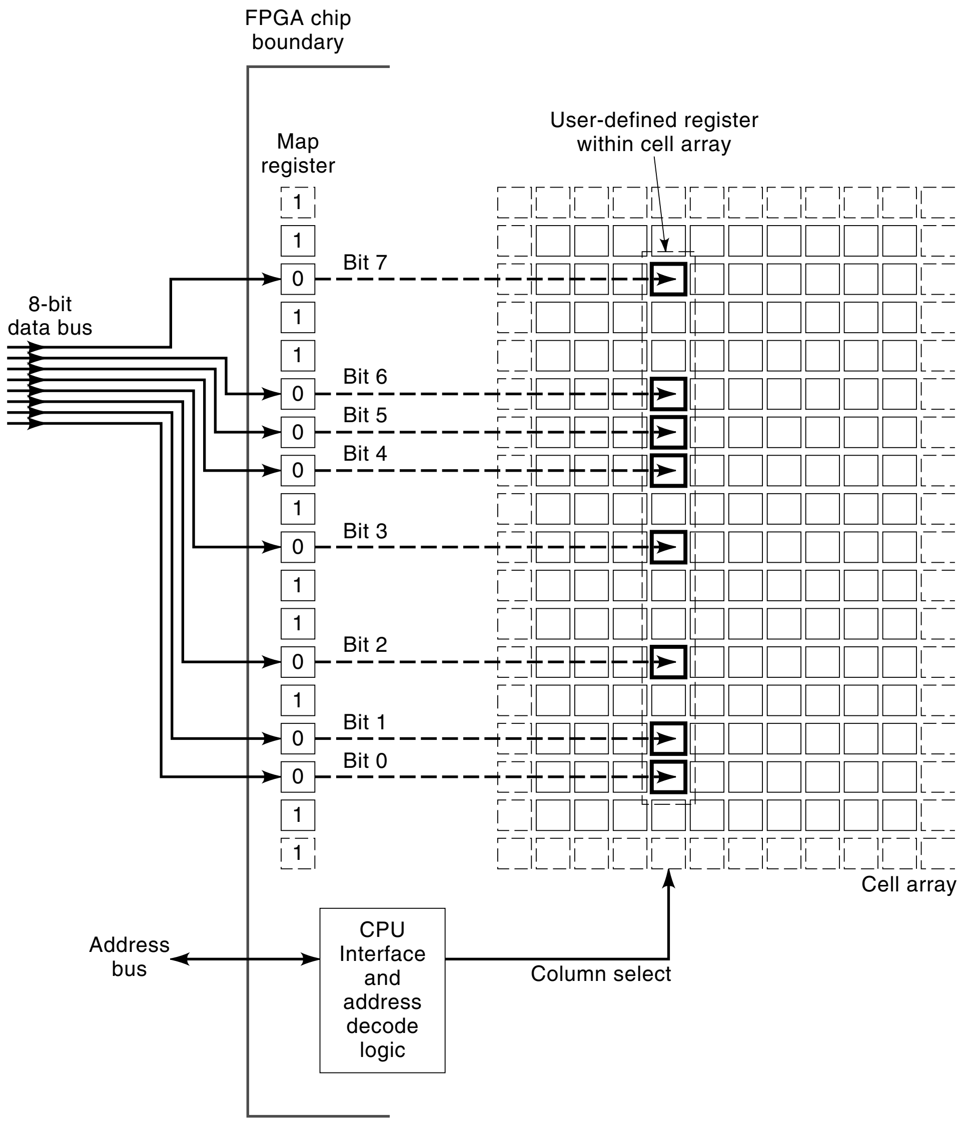

Figure 17. Distributed register access in the XC6200 FPGA.

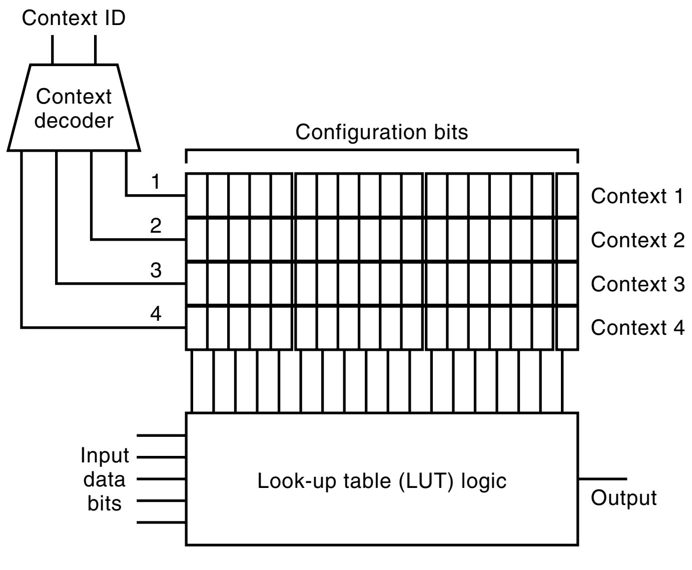

Figure 18. DPGA basic array element.

FPGA-COUPLED MICROPROCESSORS

The most common microprocessors nowadays are general purpose. They are configured for a specific application by their instruction streams. However, the instruction set as well as the computational resources of a microprocessor cannot be tailored to a specific application. To maintain operational diversity, microprocessor designs are almost universally characterized by a complicated control structure that aims at reusing the relatively small data-path portion of the processor for all types of instructions. Configurable computing aims at removing this rigidity by allowing the data-path and control logic resources to be reallocated, or reconfigured, for a specific application. In this section, we present a number of new perspectives on integrating configurable logic with microprocessor architectures, which will pave the way for a new generation of powerful, dynamically transformable architectures for future microprocessors.

The Coarse-Grained MATRIX Architecture

In contrast to the fine-grained architecture of the Xilinx XC6200 FPGA, MATRIX is a coarse-grained reconfigurable

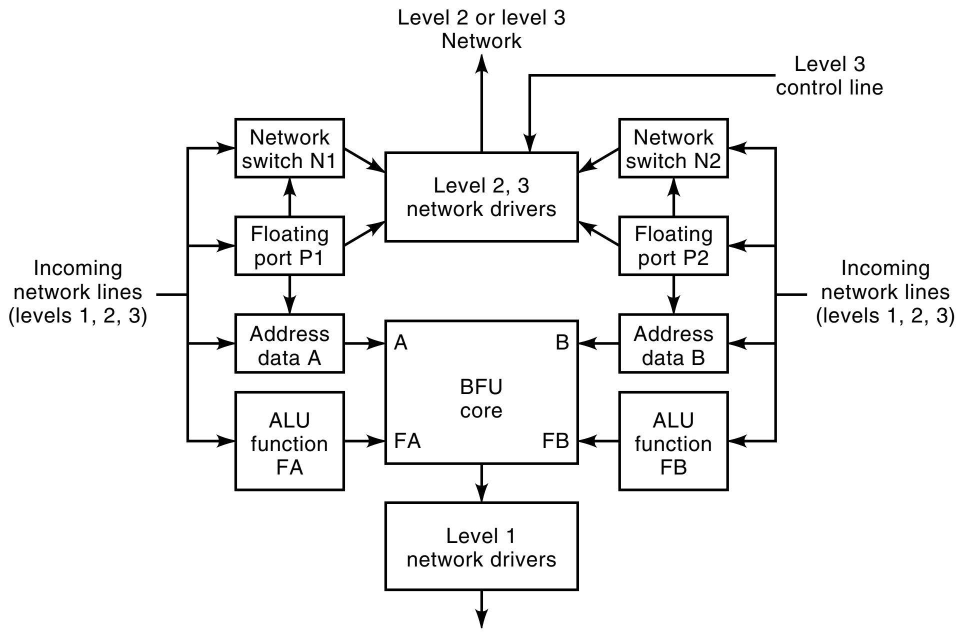

Figure 19. The main blocks of a BFU in MATRIX.

architecture, developed by A. DeHon and others at MIT, which specifically targets configurable instruction distribution (17). A typical MATRIX architecture consists of an array of basic functional units (BFUs) with a hierarchical (three-level) network of reconfigurable eight-bit buses. Each BFU is a powerful computational device containing an arithmetic or logic unit (ALU), a large register file (or memory), control logic, and reconfigurable network switches as shown in Fig. 19. The local interconnect (called a level-1 network) provides communication channels between each BFU and its 12 nearest-neighbor BFUs, within two Manhattan grid squares, as shown in Fig. 20. At the next level (level-2 network) length-4 bypass buses provide medium-distance interconnects among the BFUs as shown in Fig. 21. Level-2 networks also allow corner turns and some data-shifting operations. At the top level (level-3 network), global row and column buses span the entire chip width and length. Each BFU is connected to the level-3 network through special ports and network switches.

Pipeline registers are provided at each BFU input port, so that the operation of MATRIX can be pipelined at the BFU level. Pipelining is a particularly powerful feature of the MATRIX architecture that enables higher utilization of the BFUs as well as higher computational throughput. A BFU can serve as an instruction memory (for controlling the ALU and/or the interconnect), as a read/write data memory, or as an ALU/register-file slice. Thus a BFU can serve as a unit of the control logic or as a component of the data path. This flexibility is a key feature of the MATRIX philosophy, which is based on allowing the application to control the division of resources between control and computations according to its own characteristics. For example, regular computations may dictate allocating most BFUs to data-path logic, while irregular computations may dedicate most BFUs to control logic.

With current technology, it is possible to integrate hundreds of BFUs on a single silicon chip. Alternatively, a MATRIX array can be integrated on a single chip with a traditional microprocessor. In this case, the MATRIX array

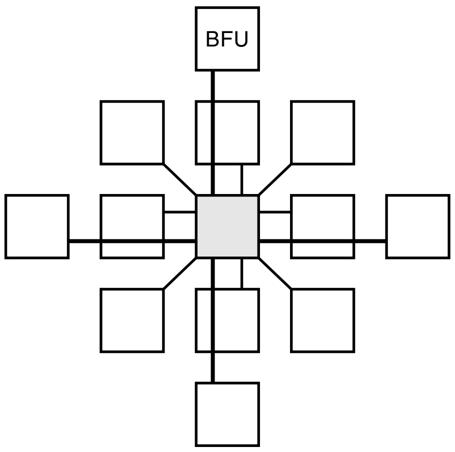

Figure 20. Nearest-neighbor interconnects among BFUs in MATRIX.

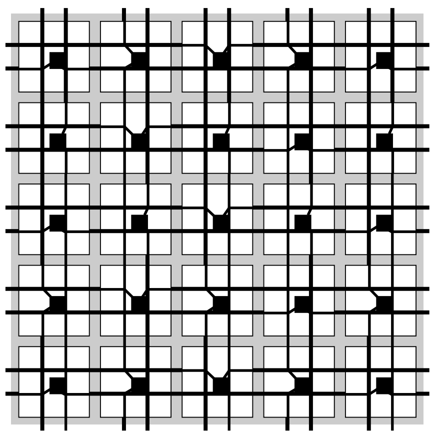

Figure 21. Length-4 bypass buses in MATRIX (black squares indicate BFUs).

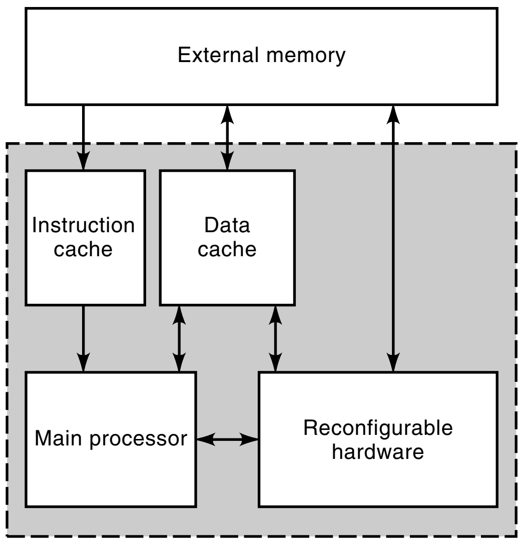

Figure 22. The main blocks of the Garp architecture.

provides a programmable function unit (PFU) that can be used as part of the data path, the control path, or both. When implemented as a part of the data path, the PFU can serve as an application-specific functional unit for executing operations that are not supported efficiently by a traditional microprocessor. For example, the PFU can be used to implement a parallel systolic array to process multidimensional arrays of data rapidly, such as those appearing in video processing or computer graphics applications. When implemented as a part of the control path, the PFU can be used, for example, to emulate and decode new instructions not supported by the microprocessor, or it can be used to customize instruction streams to a particular application.

The Garp Processor

The Garp architecture proposed in Ref. 18, combines a standard MIPS microprocessor (Silicon Graphics, Inc.) with reconfigurable hardware on the same silicon die. The goal of the Garp concept is to employ reconfigurable hardware in a processor architecture that fits into ordinary processing environments. Figure 22 shows the main blocks of the Garp architecture. The reconfigurable hardware used in the Garp processor employs reconfigurable logic blocks, which are very much like the CLBs in the XC4000 FPGA described earlier. However, the logic blocks are arranged in rows to allow parallel access to, and processing of, wide words of data as required by typical microprocessor operations. Observe from Fig. 22 that the instruction stream does not access the reconfigurable array directly, but rather through the MIPS processor. In the Garp processor, the loading and execution of configurations on the reconfigurable hardware are always done under the direct control of a program running on the MIPS processor. Therefore, the main thread of control in a program is always managed by the processor, with certain computational loops forwarded to the reconfigurable hardware for faster execution. In this respect, the Garp architecture presents an enhanced single-chip version of the transformable coprocessor concept developed in Refs. 10, 19, and 20. However, one interesting aspect of the Garp processor is the development of a software environment that links configuration files into C programs.

Dynamic Instruction Set Computer

The dynamic instruction set computer (DISC) presents another effort toward combining reconfigurable computing with microprocessors (21). DISC employs FPGAs to augment and alter the instruction set of the processor dynamically. The basic system is illustrated in Fig. 23. As shown, the DISC approach employs two FPGAs: a processor FPGA and a controller FPGA. The controller FPGA loads configurations stored in a special memory onto the processor FPGA in response to requests from the program running on a host computer. If the configuration memory does not contain the requested circuit, the processor FPGA initiates a request to the host computer, which loads the appropriate configuration.

PROCESSOR ARRAYS WITH RECONFIGURABLE BUSES

Configurable computing has been known to the parallel processing community since the mid-1980s. The work was initially pioneered by Miller, Prasanna-Kumar, Reisis, and Stout (8), and soon after, a large number of researchers contributed to this area (see Refs. 2, 3, 5, 7–9, 22–26). The bulk of the research work in this area has targeted developing ultrafast solutions for several basic problems such as sorting, searching, arithmetic computations, and problems from computational geometry. This research has laid out the theoretical foundations of configurable computing and established the justification for using processor-array models with reconfigurable interconnects.

A typical reconfigurable processor array consists of simple processors or processing elements (PEs), which are interconnected in a regular multidimensional structure by short links or bus segments. The distinguishing feature of such arrays is that each PE is capable of locally, or internally, reconfiguring the interconnection among its various ports, allowing data to transparently “pass through” the PE. Local reconfiguration can be used to realize other useful interconnection configurations such as crossover and broadcast interconnects. A subset of interprocessor links that are interconnected through local reconfiguration forms a single *bus* spanning the involved PEs.

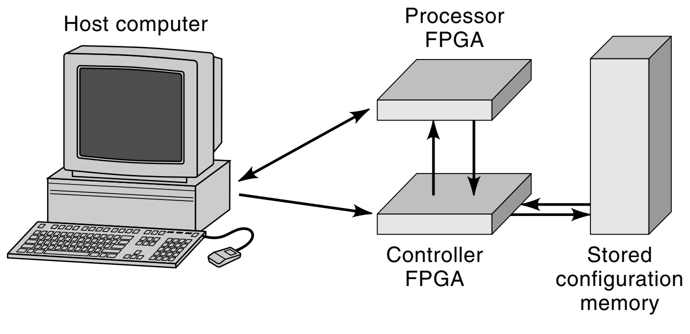

Figure 23. Basic organization of a DISC.

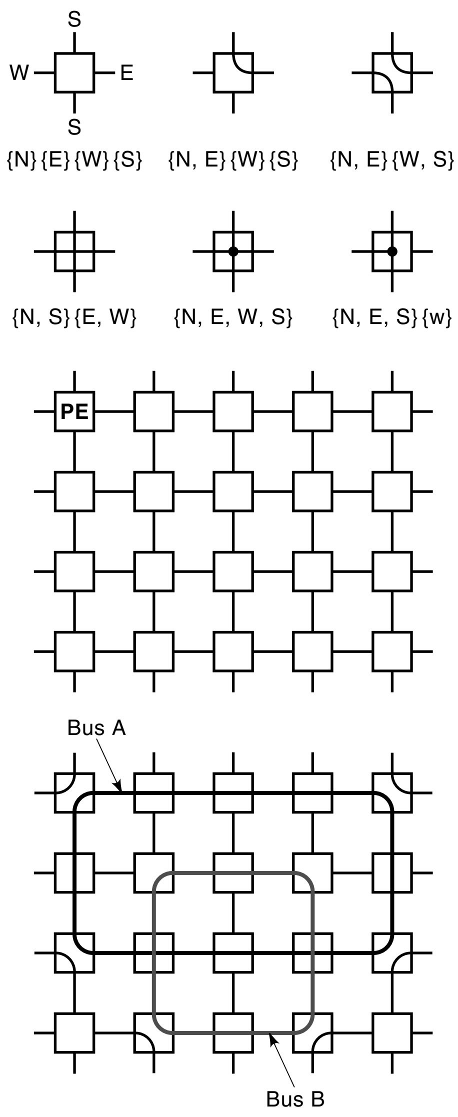

Figure 24. A reconfigurable network of processors showing a four-port PE with some allowable configuration, an uncommitted 4 × 5 RNP, and a 4 × 5 RNP configured to form two global buses (bus A and bus B).

Figure 24 shows a few possible local-switch configurations for a PE with four ports labeled N, E, W, and S, which can be used to connect the PE to its north, east, west, and south neighbor PEs, respectively. In representing the different switch configurations, we adopt the convention of placing within parenthesis the ports that are connected together within a PE. For example, the notation (N,S,W)(E) indicates that ports N,S, and W are connected together within a PE, while the notation (N,E)(S,W) indicates two distinct groups of connected ports within the same PE. Figure 24 also illustrates how a group of PEs can use their local reconfiguration capability to construct multiple global buses. Observe that more than one bus can pass through the same PE when the crossover local configuration [i.e., the configuration (N,S)(E,W)] is employed. It should be realized that all PE ports and interprocessor links can be n-bit wide. In this case each link, shown as a single edge in Fig. 24, actually represents an n-bit-wide bus segment.

In this class of reconfigurable architectures, the simple processors participate dynamically in the process of reconfiguring the network of buses interconnecting their ports. The dynamic reconfiguration process can alter the interprocessor topology on a per-instruction basis. Varying the interconnection network topology in this dynamic manner provably contributes to enhancing the computational power of such processor arrays. Indeed, such processor arrays are capable of solving many classes of problems in constant time, that is, in a fixed number of steps, which is independent of the problem size or the number of data items that must be processed by a parallel program.

Reconfigurable Network of Processors Model

The reconfigurable network of processors (RNP) models discussed in the following all fall under the single-instruction multiple data (SIMD) model of parallel processing architectures. In the SIMD model all PEs operate synchronously under the control of a single control unit that issues the same instruction to all PEs within each instruction cycle. However, a PE can modify the execution of an instruction based on its local information. For example, two different PEs may apply the same instruction on a different subset of ports based on some local state information.

To simplify presentation, only one- or two-dimensional RNP models will be discussed. However, one should bear in mind that the discussion can be extended in many cases to larger dimensions. It will be assumed that each PE has a fixed amount of local memory and a fixed number of ports, which are both independent of the RNP size. It is very important to realize that each PE employed in such models executes one of two types of activities within each instruction. The first activity is configuring the local interconnection among the PE ports; the second is executing arithmetic or logic operations on local data (in the PE memory or available at the PE ports). We assume that both activities can be completed in constant time for a single instruction.

It is interesting to observe that the close resemblance between RNP models and the MATRIX reconfigurable architecture described earlier. The RNP model is still slightly more powerful than what the MATRIX architecture can achieve, because the PEs in a RNP model can execute several types of conditional and unconditional instructions that the BFU of the MATRIX architecture cannot handle. However, the MATRIX architecture can be easily augmented with such capabilities.

Local Switch Models and Properties

At this point it may be useful to consider each PE to consist primarily of a reconfigurable switch connecting the PE ports, in addition to the arithmetic or logic processing hardware. The type of local interconnect function supported within a PE has a direct impact on the relative computational power of reconfigurable processor arrays. In the following, several switch models will be defined and their effect on global computations will be discussed.

Conditional versus Unconditional Switch Configuration. One important aspect of local switch configuration is whether a switch is controlled locally, that is, by the PE, or globally by the centralized control unit that issues instructions to all

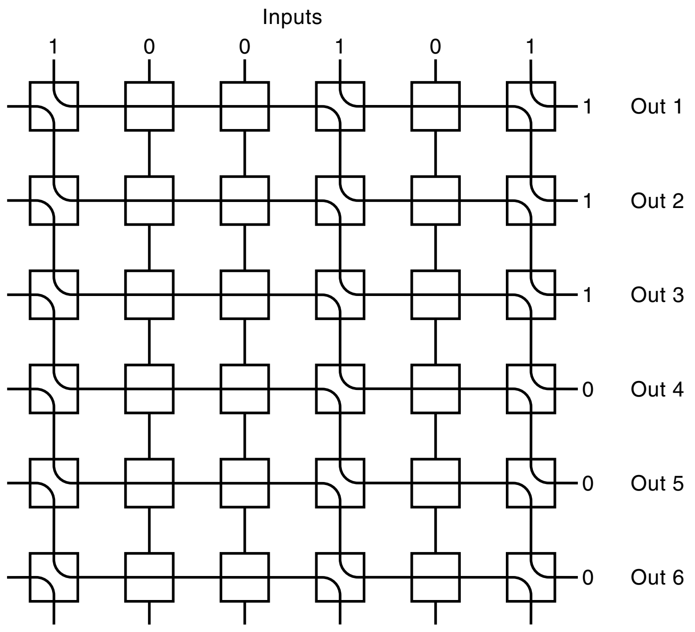

Figure 25. Counting 1's on a 6 × 6 RNP.

PEs. Local switch control provides each PE with a certain level of autonomy in the sense that different PEs, executing the same instruction, can select different switch configurations based on local state information or other local decisions made within each PE. Global, or unconditional, switch configurations can be also issued by the control unit to force all, or a selected subset, of PEs to select the same local switch configuration among their respective ports. The example given in Fig. 25 illustrates the interplay among global and local switch configurations in solving a simple, but important, counting problem. Here, a 7-bit binary input of 0's and 1's is input to a 6 × 6 RNP, such that 1 bit is supplied to port N of each of the top-row PEs. The RNP is required to count the number of 1 bits in the input. This problem can be solved using the following procedures. Initially, an unconditional instruction is issued to each PE to connect its N and S ports, which results in six column broadcast buses. Each one of these buses can be used to copy the input bit to all PEs in its column. The next instruction is executed conditionally by each PE as follows. Each PE that has received a 1 connects its N port to its E port and its W port to its S port, that is, the PE sets up a (N,E)(S,W) configuration. On the other hand, a PE that has received a 0 will set up a (E,W)(N)(S) configuration, that is, it internally connects its E and W ports. Counting is performed by observing that the output PEs, labeled Out 1 to Out 6 in Fig. 25, have the 1 bits and 0 bits appearing in sorted order on their E ports. To determine the actual number of 1's present in the input, each output PE with a 1 appearing on its E port determines whether its south neighbor PE has a 0 on its E port. Only one output PE will detect this condition. Then this PE can use its own row address to indicate the number of 1's in the input. In Fig. 25, the rightmost PE in row 3 determines that the total number of 1's is 3.

To perform arithmetic operations on a RNP in which each processor has four ports (N, S, E, and W), it is necessary that the numbers be represented in an efficient form depending on the ease with which they can be manipulated for a given arithmetic operation. The addition circuits shown in Fig. 26 use two different types of coding schemes, one for the digits entered from the leftmost column and the other for the digits entered from the bottom row of the RN. Inputs through the leftmost column and outputs from the rightmost column are represented using a *unitary coding* scheme in which n bits are used to represent an integer in the range $[0, n - 1]$. An integer I is represented by presenting a 1 signal to the W port of the lower $I + 1$ PEs in the leftmost column, and a 0 signal to the rest of the PEs in that column. Each input from the bottom row controls the column of PEs above it. In this case, it is sufficient to represent this digit using a nonpositional count-based code. Such a code represents an integer I in the range $[0, n]$ by presenting 1 signals to the S port of any subset of I PEs in the lower row of the array, as shown in Fig. 26. Note that the representation of a number by such a code is not unique. In Ref. 26, it has been shown that the combination of "adder" RNPs with the "divide-by-2" RNPs leads to constant-time algorithms for adding N k-bit numbers on a bit-model RNP with $2N \times 2kN$ PEs.

Bit versus Word Models. In general, parallel-processing computational models can be divided into bit models and word models. The difference between the two models depends mainly on how many bits of information a PE needs to access, within a single instruction cycle, before it can decide on how to configure its local switches. In a bit model, a PE only needs a fixed number of bits to make its decision independent of the problem size or the processor array size. In a word-model PE, the number of bits required is a function of problem size. For example, if a PE, in an array containing K PEs, needs to know its relative position among the other PEs before deciding on which ports to connect, then this is a word-model processor array even if the links and internal data paths within the PEs are 1-bit wide. This is because $\log K$ bits are needed to encode the position (or address) of each PE in the processor array, and this information must be stored within each PE.

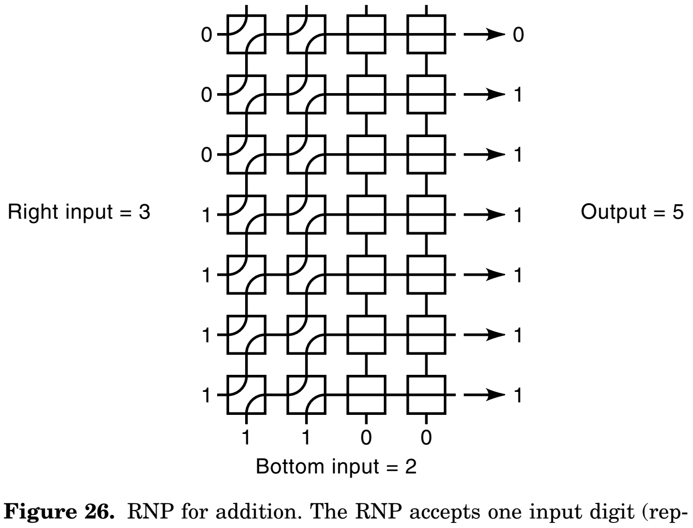

Figure 26. RNP for addition. The RNP accepts one input digit (represented in unitary code) from the rightmost column, and a second input (represented in nonpositional code) from the bottom row. The output is produced in unitary code format.

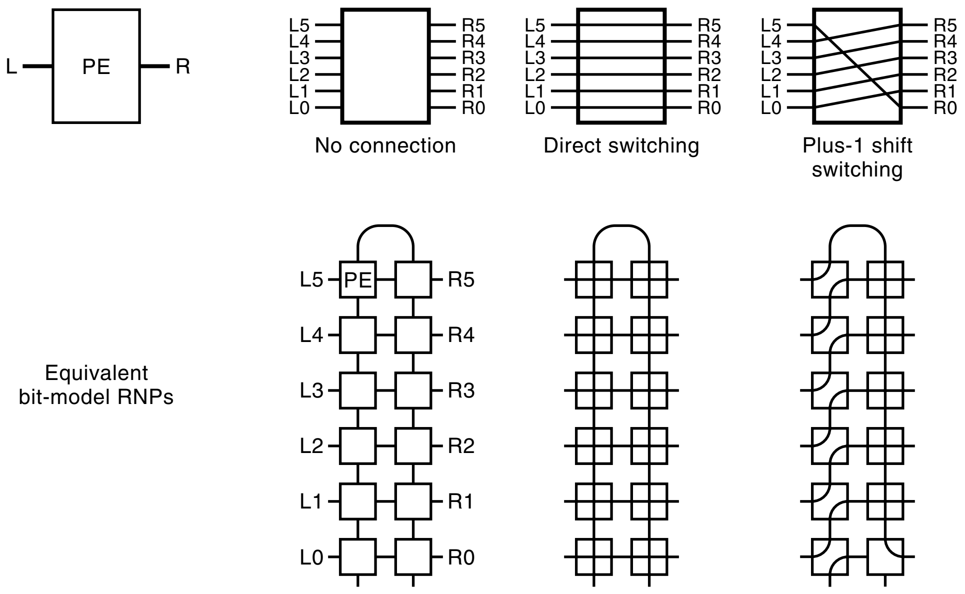

Figure 27. A two-port, plus-1 shift-switching PE and its equivalent bit-model RNP.

A two-layered bit model of the reconfigurable mesh was introduced in Ref. 7. This model is capable of simulating all other reconfigurable mesh models without an asymptotic increase in the size of the mesh, an increase in the size of the mesh, or an increase in its time complexity. It can be shown that a two-layered bit-model RNP of size $wK \times wK$ can simulate all arithmetic and logic operations performed by a corresponding word-model RNP of size $K \times K$, where w is the word length in bits.

A number of improvements in very-large-scale integration (VLSI) area and time complexity can be achieved with the bit model for several problems, such as counting 1's in a binary string, computing innerproducts, and radix sorting. For instance, the problem of counting 1's in a binary string of length K can be solved in constant time on a bit-model RNP with K log $2K$ PEs, while logarithmic time is required on a corresponding word-model RNP with K word-size PEs. Also, integer sorting on the bit-model RNP is faster by a factor of $O(w)$ over the algorithm reported for the RNP with a shift-switching word model (7).

Direct versus Shift-Switching Models. The computational power of a reconfigurable network depends directly on the basic capabilities of its local switches. For example, a RNP employing switches that allow several wires to cross over one another is more powerful than a RNP that employs noncross-over switches. Shift switching is another type of a local switch that can contribute to the computational power of word-model reconfigurable processor arrays. In shift switching, the data lines from one port can be cyclically shifted before they are connected to the data lines of another port. It should be emphasized that the shift-switching model is meaningful only within the context of a word model of computation. Specifically, shift switching provides additional computational power only when compared to standard word-model RNs. However, it can be shown that it is always possible to construct bit-model RNPs of equivalent computational powers as shift-switching RNs, and with comparable hardware complexity (2,7). Figure 27 shows that a two-port, plus-1 shift-switching PE with w-bit-wide ports has an equivalent bit-model RNP with $2wn$-bit-size PEs (each having four ports). This transformation that converts shift-switching PEs to bit-model RNPs can be generalized to shift-switching PEs with more than two ports. For example, if a two-port PE in a one-dimensional shift-switching RNP with a w-bit-wide bus allows q different shift states, then its function can be realized by at most $2wq$ bit-model PEs. Shift-switching models play a useful role in developing simple RNP algorithms with a small number of configuration states, for example, connect-with-shift, connect-with-no-shift, and do-not-connect states. Such algorithms can then be mapped onto their equivalent bit-model RNPs using standard transformations.

One particular useful application of shift switching is in bit counting, or addition, problems. For example, a prefix mod-k bit-counting RNP can be constructed from a linear connection of two-port shift-switching PEs with a k-bit bus, as shown in

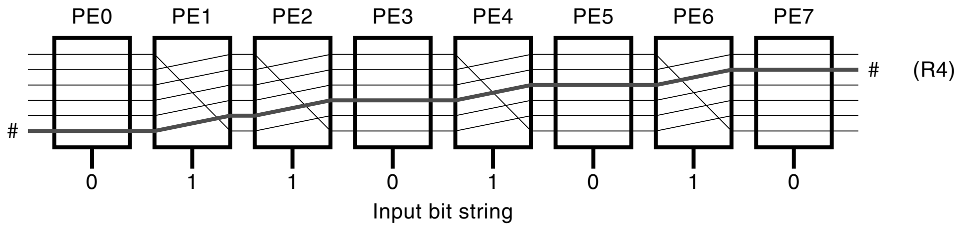

Figure 28. A mod-6 bit counter or adder. A PE reading a 0 selects a direct connection among its ports while a PE receiving a 1 selects a plus-1 shift connection among its port. Counting is achieved by passing a marker bit through the array. The final output position of the marker indicates the total number of 1's in the input string. In this example, the marker emerges from output port R4 indicating a sum of four 1's.

Fig. 28 for the case $k = 6$. Another important application of bit counting is in enumeration sorting. To sort N elements, the enumeration-sort algorithm starts by comparing all pairs of input numbers and produces a two-dimensional array of 0's and 1's based on whether a particular number is smaller or larger than its mate. The second step of the algorithm consists of computing the rank of each number by summing the 1's in each column of the 0-1 array. This summation can be implemented using mod-k shift-switching RNPs. The summation procedure continues in an iterative fashion and can be shown to require $\log N/\log k$ iterations on an $N \times N$ mod-k shift-switching RNP.

BIBLIOGRAPHY

1. G. Estrin et al., Parallel processing in a restructurable computer system, *IEEE Trans. Electron. Comput.,* 747–755, Dec. 1963.
2. H. Alnuweiri, M. Alimuddin, and H. Aljunaidi, Switch models and reconfigurable networks: Tutorial and partial survey, in *Proc. Workshop Reconfigurable Architectures,* 8th Int. Parallel Process. Symp., Cancun, Mexico, April 1994.
3. Y. Ben-Asher et al., The power of reconfiguration, *J. Parallel Distrib. Comput.,* **13** (2): 139–153, 1991.
4. P. Bertin, D. Roncin, and J. Vuillemin, Introduction to programmable active memories: A performance assessment, in J. McCanny, J. McWirther, and E. Swartslander (eds.), *Systolic Array Processors,* Englewood Cliffs, NJ: Prentice-Hall, 1989, pp. 300–309.
5. J. Elmesbahi, O(1) algorithm for image component labeling on a mesh connected computer, *IEEE Trans. Syst. Man Cybern.,* **21**: 427–433, 1991.
6. M. Gokhale et al., Building and using a highly parallel programmable logic array, *IEEE Comput.,* **24** (1): 81–89, 1991.
7. J. Jang, H. Park, and V. K. Prasanna, A bit model of a reconfigurable mesh, in *Proc. Workshop Reconfigurable Architectures,* 8th Int. Parallel Process. Symp., Cancun, Mexico, April, 1994.
8. R. Miller et al., Meshes with reconfigurable buses, in Proc. 5th MIT Conf. Advanced Res. VLSI, Cambridge, MA, 1988, pp. 163–178.
9. B. F. Wang, G. H. Chen, and F. C. Lin, Constant time sorting on a processor array with a reconfigurable bus system, *Inf. Process. Lett.,* **34** (4): 187–192, 1990.
10. S. Casselman, Virtual computing and the virtual computer, in *Proc. FPGAs Custom Comput. Mach.,* Los Alamitos, CA: IEEE CS Press, 1993, pp. 43–48.
11. P. M. Athanas and H. F. Silverman, Processor reconfiguration through instruction-set metamorphosis, *IEEE Comput.,* **26** (3): 11–18, 1993.
12. Xilinx, The Programmable Logic Data Book, 1994.
13. Xilinx, XC6200 Field Programmable Gate Arrays, Product Description (Version 1.10), April, 1997.
14. *AT&T Field Programmable Gate Arrays Data Book,* Allentown, PA: AT&T Microelectronics, April 1995.
15. J. E. Vuillemin et al., Programmable active memories: Reconfigurable systems come of age, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **4**: 56–69, 1996.
16. A. DeHon, DPGA-coupled microprocessors: Commodity ICs for the early 21st century, in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.,* April 1994, pp. 31–39.
17. E. Mirsky and A. DeHon, MATRIX: A reconfigurable computing architecture with configurable instruction set and deployable resources, in *Proc. FPGA's Custom Comput. Mach.,* Los Alamitos, CA: IEEE CS Press, 1996, pp. 157–166.
18. J. R. Hauser and J. Wawrzynek, Garp: A MIPS processor with a reconfigurable coprocessor, in *Proc. IEEE Symp. Field-Programmable Custom Comput. Mach.* FCCM '97, April 1997.
19. H. Chow, Transformable computing for MPEG video coding, Master's thesis, University of British Columbia, Vancouver, B.C., November 1996.
20. H. Chow, H. M. Alnuweiri, and S. Casselman, FPGA-based transformable computing for fast digital signal processing, *3rd Canadian Workshop Field Programmable Devices* FPD'95, 1995, pp. 25–31.
21. M. J. Wirthlin and B. L. Hutchings, A dynamic instruction set computer, in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.,* April 1995, pp. 99–107.
22. H. M. Alnuweiri, Constant-time parallel algorithms for image labeling on a reconfigurable network of processors, *IEEE Trans. Parallel Distrib. Syst.,* **5**: 320–326, 1994.
23. X. Jenq and S. Sahni, Reconfigurable mesh algorithms for the area and perimeter of image components and histogramming, in *Proc. Int. Parallel Process. Symp.,* 1991, pp. 280–281.
24. H. Li and M. Maresca, Polymorphic-torus network, *IEEE Trans. Comput.* **C-38**: 1345–1351, 1989.
25. R. Lin and S. Olariu, Short reconfigurable buses for computer arithmetic, in *Proc. Workshop Reconfigurable Architectures,* 8th Int. Parallel Process. Symp., Cancun, Mexico, April 1994.
26. K. Nakano, Efficient summing algorithms for a reconfigurable mesh, in *Proc. Workshop Reconfigurable Architectures,* 8th Int. Parallel Process. Symp., Cancun, Mexico, April 1994.

Reading List

M. Bolotski, A. DeHon, and T. F. Knight, Jr., Unifying FPGAs and SIMD arrays, Transit Note 95, MIT Artificial Intelligence Laboratory, September 1993.

W. S. Carter et al., A user programmable reconfigurable logic array, *IEEE 1986 Custom Integrated Circuits Conf.,* May 1986, pp. 233–235.

H. Chow and H. M. Alnuweiri, FPGA-based transformable coprocessor for MPEG video processing, *Photonics East '96—SPIE Int. Symp. Voice, Video, Data, Conf. 2914: High-Speed Comput., DSP, Filtering using Reconfigurable Logic,* November 1996.

S. A. Cuccaro and C. F. Reese, The CM-2X: A hybrid CM-2X/Xilinx prototype, in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.,* April 1993, pp. 121–130.

C. Ebeling, D. C. Cronquist, and P. Franklin, Rapid—reconfigurable pipelined datapath, *Proc. Field-Programmable Logic,* Heidelberg: Springer-Verlag, 1996, pp. 126–135.

B. Fawcet, FPGAs as configurable computing elements, in *Proc. Workshop Reconfigurable Architectures,* 9th Int. Parallel Process. Symp., Santa Barbara, CA, April 1995.

J. P. Gray and T. A. Kean, Configurable hardware: A new paradigm for computation, in *Proc. 10th Caltech Conf. VLSI,* 1989, pp. 279–295.

R. Hartenstein and R. Kress, A datapath synthesis system for the reconfigurable datapath architecture, in *Proc. Asia South Pacific Design Autom. Conf.,* 1995, pp. 479–484.

D. T. Hoang, Searching genetic databases on Splash 2, in *Proc. IEEE Workshop FPGAs Custom Comput. Mach.,* April 1993, pp. 185–191.

E. Lemoine and D. Merceron, Run time reconfiguration of FPGA for scanning gnomic databases, in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.,* April 1995, pp. 90–98.

D. Lpresti, Rapid implementation of a genetic sequence comparator using field-programmable logic arrays, in *Advanced Research in VLSI,* Cambridge, MA: MIT Press, pp. 138–152.

W. Luk, N. Shirazi, and P. Cheung, Compilation tools for run-time reconfigurable designs, in *Proc. FPGA's Custom Comput. Mach.,* Alamitos, CA: IEEE CS Press, 1997, pp. 56–65.

W. H. Mangione-Smith et al., Seeking solutions in configurable computing, *IEEE Comput.,* **30** (12): 38–43, December 1997.

M. Wazlowski et al., PRISM-II compiler and architecture, in *Proc. IEEE Workshop FPGAs for Custom Comput. Mach.,* April 1993, pp. 9–16.

R. D. Wittig and P. Chow, One chip: An FPGA processor with reconfigurable logic, in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.,* April 1996, pp. 126–135.

J. Villasenor and W. H. Mangione-Smith, Configurable computing, *Sci. Am.,* **276** (6): 54–59, 1997.

HUSSEIN M. ALNUWEIRI
University of British Columbia

STEVE CASSELMAN
Virtual Computer Corporation

CONFIGURATION MANAGEMENT FOR NETWORKS. See NETWORK MANAGEMENT.

CONFOCAL MICROSCOPY. See MICROSCOPE IMAGE PROCESSING AND ANALYSIS.