A Reprogrammable Gate Array and Applications

STEPHEN TRIMBERGER

Invited Paper

A field programmable gate array (FPGA) can implement thousands of gates of logic, has no up-front fixed costs, and can be programmed in a few minutes by users at their site. This paper describes an FPGA that is programmed by writing into on-chip static memory. This kind of FPGA can be reprogrammed any number of times, providing a versatile platform for rapid hardware implementation. Reprogrammable technology allows software-like design methodologies to be applied to logic design. This paper describes the construction of this kind of FPGA, design tradeoffs and examples of applications that take advantage of reprogrammability.

I. Introduction

FPGA's resemble traditional mask-programmed gate arrays in their modular, extensible structure, but differ in that their programming is done by end users at their site with no IC masking steps. FPGA's are currently available in densities up to 10 000 gates. This size is large enough to implement many digital systems on a single chip, and larger systems can be made up of multiple FPGA's. Although the unit costs of FPGA's are higher than mask programmed gate arrays for the same density, there are no up-front engineering charges to use an FPGA, so FPGA's are cost-effective for many applications [1].

The design process used to target an FPGA is the same as that used for a gate array. Input can come from a schematic netlist, from a hardware description language or from logic synthesis. Vendor-supplied software converts the design description into FPGA programming with processing similar to placement and routing for a gate array. The resulting programming code can be immediately loaded into the device and the design tested. This fast turnaround significantly reduces design risk because a design error can be corrected quickly and inexpensively.

The combination of multiple-thousand-gate density and fast turnaround allows hardware designers to use an interactive, incremental design methodology previously possible only in software development. A designer can design a system, debug it, then re-work and re-prototype to improve the design. When the design is complete, the "prototype" may be sufficiently cost effective and of adequate performance for production.

Mansucript received October 1, 1992.
The author is with Xilinx, Inc., 2100 Logic Dr., San Jose, CA 95124.
IEEE Log Number 9210744.

Section II of this paper describes the architecture of a reprogrammable FPGA, Section III describes the design environment, and Section IV describes innovative applications of reprogrammable FPGA's.

II. A Reprogrammable Gate Array

Traditional mask-programmed gate arrays have significant drawbacks. They have manufacturing times from weeks to months, and large up-front expenses for tooling. The up-front costs require that gate array designs be extensively simulated to verify correct functionality before manufacture. Simulation lengthens the development cycle, raising the cost of implementation further. Each design built in a mask-programmed gate array must have an accompanying test program to catch manufacturing defects. Test program development may add weeks of effort to the design and can require redesign. All FPGA's address the manufacturing issues, but to solve the simulation and testing issues requires a reprogrammable gate array.

This paper describes a family of FPGA's called Logic Cell Arrays (LCA's) that are configured by loading a set of configuration bits into on-chip static memory [2]–[5]. The memory implements logic and controls interconnections paths for signal routing. The memory is built from static RAM cells, so the chips are often termed "SRAM programmable."

SRAM programming has an obvious drawback, namely volatility. When the power is turned off, the IC loses its programming, so the chip must be reprogrammed every time power comes up. For systems that need active logic during power-up, an SRAM-programmed part is not a viable solution. However, SRAM-programmable FPGA's include logic to sense power-on and to initialize themselves automatically, providing "virtual nonvolatility," provided

0018–9219/93$03.00 © 1993 IEEE

the application can wait the tens of milliseconds required to program the FPGA.

Reprogrammability has some significant advantages. The FPGA manufacturer can test all paths in the FPGA by reprogramming it on the tester. Users need not verify proper manufacture of their individual designs on the part. They get well-tested parts and 100% "programming yield" with no design-specific test patterns and no "design for testability."

Since the on-chip programming is done with memory cells, the programming of the part can be re-written an unlimited number of times. Prototyping and in-circuit verification can replace extensive simulation for verification. Prototyping can proceed iteratively, re-using the same chip for new design iterations.

The FPGA described in this paper allows an application to read out the contents of all internal flip flops. The development system captures these values and displays them so users can examine the interior nodes to verify correct operation of their logic. These capabilities allow a system designer to experiment with a design, try an implementation and quickly re-implement the design to fix logic or timing problems. Since there is no cost to prototype, prototypes can replace simulation for the final stages of verification. This methodology is similar to software development methodology.

Reprogrammability has advantages in systems as well. In cases where parts of the logic in a system are not needed simultaneously, they can be implemented in the same reprogrammable FPGA, and the FPGA logic switched between applications. This paper describes applications for which the reconfigurability of the FPGA is essential.

How SRAM Programming Works

This section is a bottom-up description of a simple SRAM-programmable FPGA. First, the underlying programming technology and design methodology are reviewed without the details of a particular implementation. Following the generic description is a discussion of design tradeoffs. The final section describes the Xilinx XC4000 FPGA and implementation software, discussing how many of the design tradeoffs were resolved.

SRAM Programming: An SRAM-programmable FPGA contains memory cells that control the logic that performs the application function of the FPGA. There is no separate RAM area on the chip. The memory cells are distributed among the logic elements they control. Since FPGA memories do not change during normal operation, they are built for stability and density rather than speed. The Xilinx SRAM cell, shown in Fig. 1, is a five-transistor single-ended version of a six-transistor memory cell. Low-resistance paths to the power supplies provide good stability. In normal operation, the single sense transistor is off and does not affect the stability of the cell.

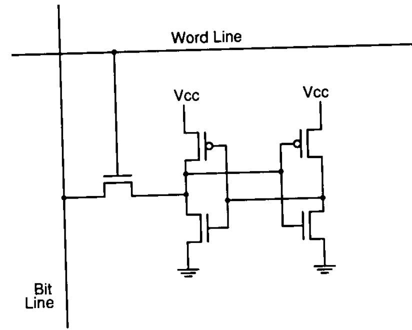


Fig. 1. The Xilinx five-transistor memory cells.

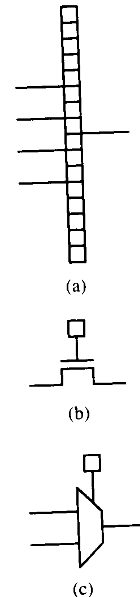


Fig. 2. Three important peices of an SRAM FPGA. (a) 16x1 lookup table; (b) pass transistor controlled by a memory cell; and (c) mux controlled by a memory cell.

Building Blocks: Figure 2(a) shows a four-input *function generator*. A function generator implements combinational logic as a $2^w \times 1$ memory. The memory is used as a lookup table, addressed by the w inputs. A designer can use a function generator to implement any of the 2^{2^w} functions of its inputs by preloading the memory with the bit pattern corresponding to the truth table of the function. For example, if all bits in the 16-bit lookup table in Fig. 2(a) are 0 except the high-order bit (binary address 1111), then the function generator implements a four-input AND, since the output of the function generator is zero unless the address is binary 1111, selecting the high-order bit.

All functions of a function generator have the same timing: the time to look up the result in the memory. Therefore, the inputs to the function generator are fully swappable by simple rearrangement of the bits in the lookup table. The routing software described later takes advantage of pin swappability to improve signal delay and routability.

The second building block is called a *programmable interconnect point*, or PIP. The PIP, shown in Fig. 2(b), is a pass transistor controlled by a memory cell. The PIP is the atomic unit of configurable interconnect. The wire

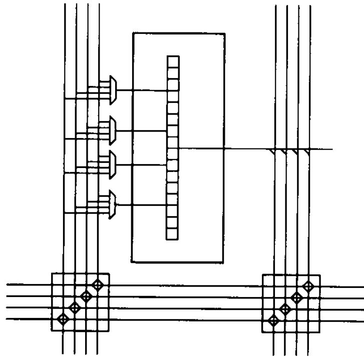


Fig. 3. A simple configurable logic block.

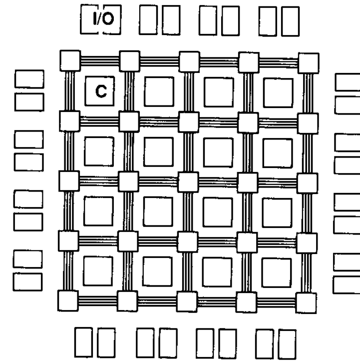


Fig. 4. An FPGA chip.

segments on each side of the transistor are connected or not, depending on the value in the memory cell. The pass transistor introduces resistance into the interconnect paths, and hence delay. Interconnect delay issues are addressed later in the section on design tradeoffs.

The third building block is a multiplexer controlled by a memory cell [Fig. 2(c)]. The multiplexer is a special-case one-directional routing structure. Multiplexers may be of any width, with more configuration bits for wider multiplexers.

The Logic Block: Figure 3 shows the building blocks from Fig. 2 combined into a *configurable logic block (CLB)* with wiring. The CLB in Fig. 3 contains a single four-input function generator surrounded by *wiring channels.* Each wiring channel contains several *segments.* Segments have connections to the CLB and to each other through multiplexers and PIP's.

In Fig. 3, multiplexers connect the segments in the left side channel to the CLB inputs. The CLB output can be connected to any segment at the right through PIP's (shown as diagonal connections). If the block is unused, all PIP's are turned off so the block does not drive its signal onto any wire.

Segments in channels connect to segments in other channels through PIP's in *switchboxes* at the intersection of the channels. The diamond-shaped connection represents the six PIP's that connect any pair of segments independently of all others. The output signal from the block may be routed through additional switchboxes to other blocks or back to the same block to implement sequential circuits.

The Chip: Figure 4 shows a chip composed by building an array of CLB's with surrounding wiring, then surrounding the array with configurable *I/O Blocks* (IOB's). The simplest form of an IOB is a three-state bidirectional pad with input, output and three-state connections to the adjoining interconnect. If the three-state control signal is always on, it is an output pad; if the three-state signal is always off, it is an input pad; otherwise the pad is a bidirectional pad.

Upon initialization, the contents of all memory cells are loaded to program the chip. Conceptually, they are connected in a single long shift-register. When the chip is powered up, the *configuration bitstream* is shifted into all programming cells from off-chip memory.

Design Tradeoffs:

Wiring structure versus size and speed: The fraction of delay incurred due to routing in an FPGA is significantly greater than that in a traditional mask-programmed gate array. In a traditional gate array, wiring is implemented on metal runs with low capacitance and resistance. On an SRAM-programmable FPGA, a signal must pass through a transistor resistance at each PIP. Each wiring segment is loaded with the capacitance of the PIP's attached to it.

The delay of a signal is due less to the distance it travels than to the number of PIP's through which it passes and on the load on the interconnect segments. Signal delay can be reduced by implementing longer interconnect resources without substantially increasing the load on those interconnects. Buffering in the interconnect eliminates the effect of side-branch loading, also improving delay.

Fewer PIP's in the interconnect reduce the load on the interconnect segments, but also reduce the utility of the routing. More PIP's improve routability, but also add loading and area. Each PIP includes a memory cell, so a large number of PIP's leads to a large area devoted to interconnect. Rose and Brown [6] investigated these tradeoffs and determined that an interconnect structure with good connectivity from CLB's to the interconnect is very routable even if the switchbox connectivity is low.

Block structure versus size and speed: Simple blocks are attractive conceptually, but can lead to slow logic if many blocks with associated delay-intensive wiring are required to implement a function. A more complex block, perhaps containing a larger lookup table, can implement more complex functions without connections to the wiring, but may not be fully utilized, leading to poor logic capacity and lower logic density.

Flip-flops and other sequential elements are very common in digital design. Although they can be built from function generators, a more efficient FPGA results from including them as an option on the output of every function generator [7]. When a flip-flop is needed, the output of the flip-flop is selected as the output of the block. Although the area due to the flip-flop is wasted when the registered

output is not needed; when the flip-flop is needed, the function generator and interconnect area are usable for other functions. If typical flip-flip usage is high enough, the result is a denser overall design. This argument can also be applied to other dedicated logic, such as multiplexers, memory and carry logic for arithmetic to determine if they should be implemented directly or if they should be built from function generators and flip-flops.

The arguments for adding special-purpose logic are based on analysis of typical designs, but the ideal balance of logic resources differs for every design. A design that uses none of the special-purpose logic will use only a fraction of the available capacity of the CLB. The capacity of a complex block can also be difficult to predict using gate-array or PLD measures because the FPGA does not implement logic with those primitives. Designs that differ by a factor of two using gate counts and product terms may fit into the same number of CLB's due to different CLB utilization. A proper estimate requires that the logic be mapped into CLB's.

Large, slow interconnect does not necessarily produce a large, slow FPGA. Large logic blocks contain more logic and reduce the need for interconnect. When carefully designed, the large blocks produce complex functions quickly, so over a range of applications, complex-block SRAM architectures have been shown to be the fastest FPGA's despite the delay-intensive interconnect [8].

Software versus architecture: The capacity and speed of the parts is related to both the architecture and the software that maps designs onto that architecture. Without software capable of automating the design implementation, the usable capacity and speed of the FPGA will be lower than that expected from an analysis of the architecture. Complex blocks require software to partition the logic into those blocks efficiently. This partitioning step is not part of physical design for traditional gate array and standard cell chips. Complex wiring complicates the routing cost estimate in the placement software. In mask-programmed gate arrays and standard cells, closer placement is nearly always better placement. When the FPGA wiring includes long wire segments, it may be preferable for the placer to align the cells along the segments rather than to cluster them.

Architectural Overview of the XC4000 Series FPGA

This section describes the Xilinx XC4000 LCA family [5], [9] as an example of a real-world architecture of an SRAM programmable FPGA. The XC4000 is derived from similar earlier architectures [2], [3] and shares many of their logic and interconnect structures. The concerns and tradeoffs described in previous sections are addressed in this architecture and lead to an interesting combination of special-purpose and general-purpose features.

The Logic Block

The XC4000 CLB, shown in Fig. 5, is substantially more complicated than the simple block in Fig. 3. The CLB contains three function generators, two flip-flops and several programming-controlled multiplexers. The two primary input function generators, labeled F and G, each implement a function of four variables. These two results can be brought out of the CLB independently or they can be combined in the H function generator into any function of five inputs and some functions of up to nine inputs. This allows functions such as nine-input AND, OR, XOR (parity) or decode to be done in one CLB. The flip-flops can take their inputs from the function generators or from an external signal. The clock, clock enable and reset signals are shared by the two flip-flops.

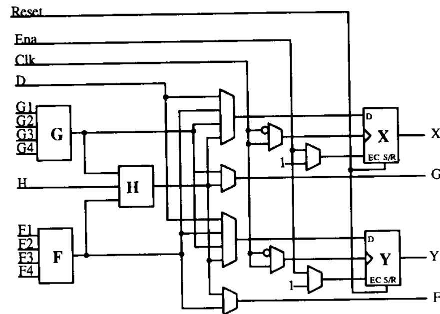


Fig. 5. The configuration logic block for the XC4000.

With more logic capability within a CLB, functions require less routing outside the block and circuits run faster. The direct implementation of flip-flops results in not only better logic density, but also good metastability behavior, controllable setup and hold times and a powerful pipelining capability.

The XC4000 allows application logic to write to the F and G function generator memories using the function generator inputs to address the bit, and additional inputs to the CLB for `write enable` and `data-in` [Fig. 6(a)]. Reading the memory is the same as using it to implement a function. One CLB can implement up to 32 bits of memory.

Figure 6(b) shows a configuration of the CLB in arithmetic mode. The F and G function generators compute the sum while special-purpose logic calculates the carry. Previous-generation FPGA's [2], [3] implemented arithmetic with sum in one function generator and carry in the other. Since the speed of arithmetic operations is dominated by the speed of the carry chain, special-purpose carry logic substantially speeds up arithmetic while doubling its density.

The I/O Block

Figure 7 shows the details of the IOB. Signals to be output from the chip can be registered before output and enabled by a separate control signal to reduce clock-to-output delays. Output pins can be optionally pulled up or down. The driver on the pad can be configured in either fast-slew mode for high performance, or slow-slew mode for less noise.

Inputs from the pad can be brought into the interior of the chip either directly or registered or both. By allowing both, the I/O block can de-multiplex external signals such

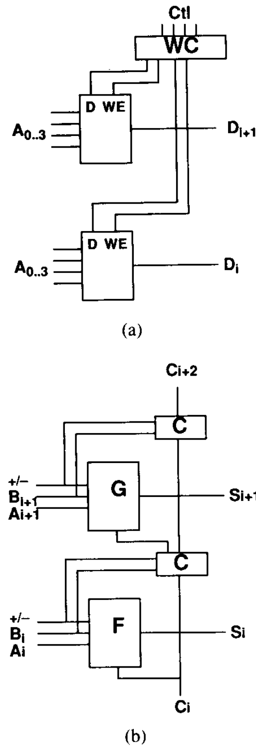


Fig. 6. Special CLB configurations: (a) memory function; (b) arithmetic function.

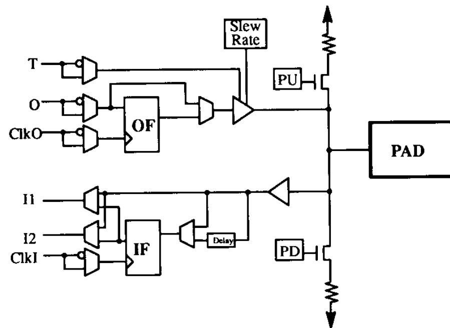


Fig. 7. The I/O Block for the XC4000.

as address/data busses, storing the address in the IO flip-flop and feeding the data directly into the wiring. To further facilitate bus interfaces, inputs can drive wide decoders, built into the wiring for fast recognition of addresses.

Wiring Architecture

The XC4000 design has a variety of wiring resources, targeted to different applications. Figure 8 shows an overview of the *general-purpose* wiring. This wiring includes *single-length lines*, wires that connect from one switchbox to the next. At each end of each wire segment in the switchbox, there are three PIP's controlling the connection of the segment to other segments from the other three directions

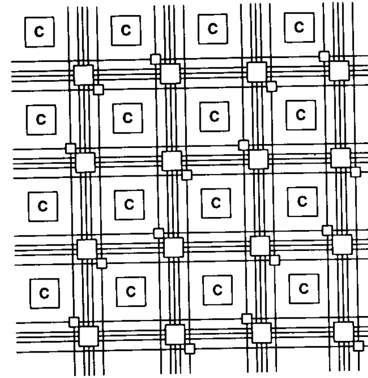


Fig. 8. General-purpose interconnect.

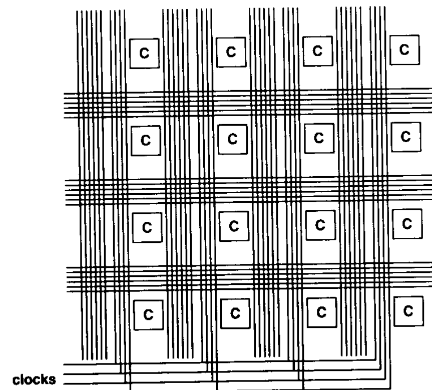


Fig. 9. Long-line interconnect.

in the switchbox. Figure 8 shows a few of the segments that bypass alternate switchboxes. Since wiring delay is more dependent on the number of segments through which signals pass than on the length of those segments, these *double-length* lines allow a signal to travel twice the distance in the same amount of time or to travel a given distance in half the time.

To reduce delay, signals on general interconnect may be buffered by routing them through unused CLB's. The choice of the CLB and the path through the CLB is left to the routing software. The router includes a delay estimator, and uses signal timing to decide when to repower a signal.

General interconnect can lead to significant skew for clock and other high-fanout signals despite delay-driven routing. The wiring architecture includes wire segments called *long lines* that span the entire height or width of the chip to support these signals (Fig. 9). Signals are buffered onto the long line to reduce skew. Six of the long lines in each channel are general-purpose for high-fanout, high-speed wiring. In each row of blocks, two of the horizontal long lines can driven through three-state buffers. Three-state buffers are an efficient alternative to multiplexers for large digital systems saving logic, interconnect and congestion.

Four of the vertical long lines in Fig. 9 are global signals optimized for clock routing. Clocks can originate on-chip or off-chip and are wired to all logic blocks through low-skew dedicated wiring. They are driven by dedicated high-drive clock buffers and pre-wired through dedicated interconnect to minimize clock skew.

The XC4000 family has members with different amounts of wiring for different size ranges. The amount of wire and the distribution among the different wire lengths is dictated by routability requirements of the FPGA's in the target size range. For CLB arrays from 14×14 to 20×20, each wiring channel includes eight single-length lines, four double-length lines, six long lines and four global lines. The distribution was derived from analysis of wiring needs of a large number of existing designs, from trial placements and global routing of those designs on the proposed arrays, and from statistical wirability analysis using Heller's techniques [10]. The general distribution was modified to conform with physical constraints from the IC layout and by the need to support memories and other multiple-block structures efficiently.

High-Level System Support

The array of blocks in the XC4000 naturally divides into two-bit horizontal slices for datapath operations. The logic blocks contain carry logic to support two bits of addition or subtraction per block, two 16×1 memories and two flip-flops per block. There are two three-state bus lines per row in the array. These bus lines connect to two I/O blocks per row. I/O blocks can be configured to continue the three-state bus off-chip. Interfaces to external systems are simplified by bus demultiplexers, on-chip address decoders and low-skew on-chip clocking.

The result is that it is easy to build bus-oriented datapaths, bit-sliced horizontally, two bits per row. Datapaths can contain arithmetic, memory or other functions. This functionality is available in addition to the ability to build random logic, interconnected in whatever fashion is needed, using function generators and flip-flops.

Configuration

Xilinx LCA's include dedicated circuitry to load the part from external memory. They can load either as a master, generating clock and enable signals; or as a slave, accepting those control signals from an external source. The programming data stream can be a serial bitstream or a parallel byte-wide stream. The data can come from a small-footprint serial PROM, or the LCA can be configured like a microprocessor peripheral, with data loaded from any other part of the system. LCA's can configure themselves when they sense power-up, or they can be re-configured on command while residing in the circuit. A designer can create a system in which the FPGA's program changes during operation.

The LCA can also *read back* its programming along with the contents of internal flip-flops, latches and memories. A working part can be stopped and its state recovered. The read-back facility is an extremely valuable verification and debugging tool for prototypes and is also used extensively in manufacturing test.

III. Design System

An application targeted to an FPGA can be designed on any one of a number of logic or ASIC design systems, including schematic capture and hardware description languages. To target an FPGA, the design is passed to FPGA-specific implementation software. The interface between design entry and design implementation is a netlist that contains the desired nets, gates, and references to hard macros.

The capacity of an FPGA and the performance of the logic in the FPGA are determined by the FPGA architecture, the design software and the interaction between them. Although many designs are implemented with manual tools, these designs typically have special density or performance requirements. Manual and automatic tools can be used in concert on a design, or an implementation can be done fully automatically. This section addresses automatic design implementation, the most common method of implementing logic on FPGA's. Implementation consists of three major pieces: partitioning, placement and routing.

Partitioning

Partitioning is the separation of the logic into CLB's. Partitioning has both a logical and physical component. The connections within a CLB are constrained by the limited intra-block paths and by the limited number of block outputs. A partitioning algorithm that ignores these constraints will build illegal blocks. However, the quality of the resulting partitioning depends on how well the subsequent placement can be done, so physically related logic should be partitioned into the same block.

The logical component has been investigated in the context of *technology mapping* in logic optimization, and work in this area has focused on collecting combinational logic into function generators [11]–[15]. However, these techniques do not handle the larger problem of mapping logic and flip-flops into CLB's.

Partitioning for the XC4000 has been addressed with a four-step algorithm that combines logical and physical optimization [16], [17]. First, a Chortle-based technology-mapping step builds function generators from combinational logic [12]. Then a pattern-matching step clusters patterns of logic that match the most-constrained paths inside a CLB. The third step is a min-cut-based placement step to cluster physically-related logic [18]. The final step is a greedy algorithm that builds legal CLB's from the partitioned, clustered logic; breaking physical and logical constraints when necessary to make a legal CLB.

Placement

Placement starts with CLB's, IOB's, hard macros and other structures in the partitioned netlist and decides which corresponding blocks on the chip should contain them. The

FPGA placement problem is very similar to traditional standard cell and gate array placement and many of the existing algorithms are applicable, such as simulated annealing [19], force-directed relaxation [20] and min-cut [18]. The rigid interconnect in the FPGA requires different cost functions in these placement algorithms. Traditional gate array interconnect is optimized when communicating blocks are close together, so total wire length is the dominant component of the cost function. Although wire length is a good metric for nets wired on LCA general interconnect, alignment is better for nets wired on long lines. The LCA placer builds an internal model of how nets are to be routed and scores its placement accordingly.

Routing

Most standard cell and gate array routers use a model that assumes a great deal of flexibility in routing—wires may bend anywhere and may connect to other channels at any point in the channel. In contrast, the FPGA routing in Fig. 8 shows very little connectivity between vertical and horizontal segments. For heavily-constrained problems of this type, A* maze routing with ripup-and-reroute [21]–[23] has been very successful and is the primary algorithm in use for FPGA routing, although other interesting routing techniques have been proposed, including coarse graph expansion [24].

In a mask-programmed gate array, interconnect delay tracks distance, so maze routers that optimize the length of nets also optimize their delay. This kind of simple resource-driven maze routing is not adequate for a technology where the delay of wiring is due not to the distance travelled, but to the kind of interconnect used to travel that distance. For example, crossing the width of a chip is faster on a long line than travelling only a few CLB's distance on general interconnect. LCA routers include an incremental delay estimator to evaluate timing during the maze expansion. The delay estimator uses the actual resistance and capacitance values of the segments and PIP's, derived from the FPGA layout. The router trades off timing and resource costs, improving both the skew and the delay of routed nets, while limiting resource requirements.

Interactive Design Editing

Interactive tools allow constraints on the automated algorithms, postroute improvements on the design and quick design iterations. The XDE editor provides this capability and allows users to program every detail of the design manually. The manual editing capability allows users to modify the configuration of any CLB or routing path. XDE also contains commands that allow interaction with an LCA in a system. These commands are discussed in the following section.

Design Environment Support for Rapid Prototyping

Download Hardware

The Xilinx design system includes a *download cable* to connect a PC or workstation to an LCA on a prototype test board or on the board-level prototype system. Designers can load the design over the cable, then run the prototype on the board. If needed, they can connect signal generators and analyzers to the LCA to verify the design at operating speed in the system.

Debugging

Interactive software allows designers to insert *probe points* into a design. A probe point can be any on-chip signal that a user wants to examine during operation. The editor routes the signal on unused interconnect segments to an unused IOB, and generates delay information for the path to the pin. Externally, a user may connect a scope or logic analyzer to the package pin on the LCA that corresponds to the IOB and observe the signal during operation. This debugging aid is the equivalent of examining a local variable in a program during debugging. A designer may move a probe point and re-load the part to watch some other internal node. Probe points and their routing are stored separately from the "real" design and are discarded after debugging.

Debugging logic may be placed and routed in unused CLB's. This logic may replace an external signal analyzer, performing buffering and checksums, triggering on internal signals or simply storing important data for a later read-back.

If there are not enough unused resources on a chip to support both the design and the debugging logic, a designer may simply use a larger capacity part available in the same package. Xilinx LCA's are available in compatible packages over a wide range of logic densities, so there can be a significant amount of space for debugging logic. The placement and routing software can restrict the "real design" to part of the larger chip to keep it separate from the "scaffolding logic." When debugging is finished, the debugging logic may be deleted and a smaller "production" part substituted for the larger "debugging" part.

Placement and Routing Support for Iterative Prototyping

In support of an iterative design methodology, Xilinx's automatic place and route system has built-in incremental design facilities. Small changes in a design are incorporated without changing unaffected parts of a design. Large, complex CLB's facilitate incremental changes because a small change can more easily be isolated to a change in a single CLB or a single new routing connection. Where the original placement and routing took hours, the incremental change may take a few minutes.

Design Flow Changes Allowed by Reprogrammability

This combination of moderate density, reprogrammability and powerful prototyping tools provides a novel capability for systems designers: hardware that can be designed with a software-like iterative-implementation methodology.

Figure 10(a) shows a typical ASIC design methodology in which the design is verified by simulation at each stage of refinement. Accurate simulators are slow; fast simulators

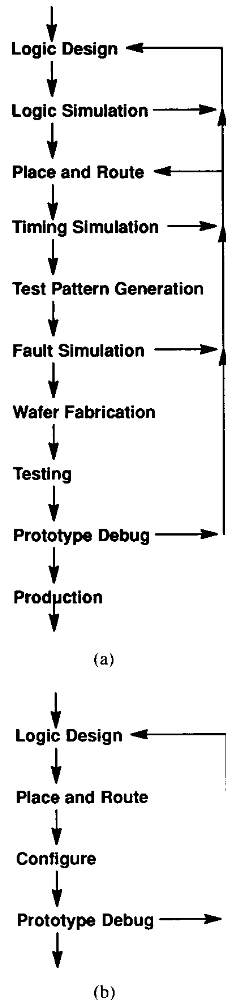


Fig. 10. Contrasting design methodologies: (a) Traditional gate arrays; and (b) FPGA.

trade away simulation accuracy. ASIC designers use a battery of simulators across the speed/accuracy spectrum in an attempt to verify the design. Although this design flow works with FPGA's as well, an FPGA designer can replace simulation with in-circuit verification, "simulating" the circuitry in real time with a prototype [Fig. 10(b)]. The path from design to prototype is short, allowing a designer to verify operation over a wide range of conditions at high speed and high accuracy.

This fast design-place–route-load loop is similar to the software edit-compile-run loop and provides the same benefits. Designs can be verified by trial rather than reduction to first principles or by mental execution. A designer can verify that the design works in the real system, not merely in a potentially-erroneous simulation model of the system. This makes it possible to build proof-of-concept prototype designs easily.

Design-by-prototype does not verify proper operation with worst-case timing, merely that the design works on the presumed-typical prototype part. To verify worst-case timing, designers may check speed margins in actual voltage and temperature corners with a scope, speeding up marginal signals; they may use a software timing analyzer or simulator after debugging to verify worst-case paths; or simply use faster speed-grade parts in production to ensure sufficient speed margin over the complete temperature and voltage range.

Prototype versus Production

As with software development, the dividing line between prototyping and production can be blurred with a reprogrammable FPGA. A working prototype may qualify as a production part if it meets cost and performance goals. Rather than re-design, an engineer may choose to substitute a faster speed FPGA using the same programming bitstream, or a smaller, cheaper compatible FPGA with more manual work to squeeze the design into a smaller IC. A third solution is to substitute a mask-programmed version of the LCA for the field-programmed version. All three of these options are much simpler than a system redesign. Rapid prototyping is most effective when it becomes rapid product development.

Field Upgrades

Reprogrammability allows a systems designer another option: that of modifying the design in the FPGA by changing the programming bitstream after the design is in the customer's hands. The bitstream can be stored in a dedicated PROM or elsewhere in the system. For example, an FPGA used as a peripheral on a computer may be loaded from the computer's disk. In some existing systems, manufacturers send modified hardware to customers as a new bitstream on a floppy disk or as a file sent over a modem.

IV. Reprogrammable FPGA Applications

Most applications of SRAM-programmable FPGA's are straightforward logic integration and, although reprogrammability may be crucial to verification of the design, they do not use reprogrammability in the system. However, this section is devoted to designs that use reprogrammability in an interesting manner. Therefore, these designs are not necessarily typical of the designs that are implemented in SRAM-programmable FPGA's, but serve to emphasize the unique capabilities of reprogrammable FPGA's.

Reprogrammability for Board-Level Test

The most common system use of reprogrammability is for board-level test circuitry. Since FPGA's are commonly used for logic integration, they naturally have connections to major subsystems and chips on the board. This puts the FPGA in an ideal location to provide system-level test access to major subsystems. The board designer makes one configuration of the FPGA for normal operation and a separate configuration for test mode. The "operating" logic and the "test" logic need not operate simultaneously, so

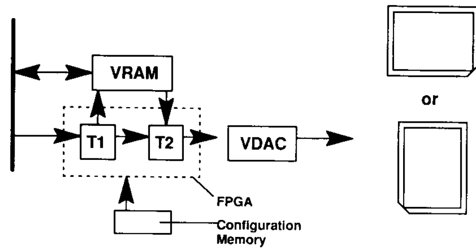


Fig. 11. Pivoting display system block diagram.

they can share the same FPGA. The test logic is simply a different configuration of the FPGA, so it requires no additional on-board logic or wiring. The test configuration can be shipped with the board, so the test mode can also be invoked as a diagnostic after delivery without requiring external logic.

Rosendahl [25] exploited reprogrammability-for-test in a bus interface built with a reprogrammable FPGA. One configuration of the FPGA is the bus interface. Additional test configurations of the FPGA include an IEEE 1149.1 boundary scan Test Access Port for the board. Specific tests are loaded with the TAP, including a *13N* pattern test for on-board RAM. The RAM tester generates addresses and patterns, captures and compares the results and keeps an error vector for later analysis.

Another configuration of the FPGA performs parametric tests. The FPGA test logic samples the input data at various times during the read cycle and compares it with the expected pattern. The earliest match gives the speed of the memory subsystem. Without a reprogrammable FPGA, the test and parameter measurement logic would have required thousands of additional gates of on-board logic as well as a significantly more complex board layout.

Reprogrammable Logic in a Configurable Display Device

The Radius Pivot monitor is a Macintosh-compatible display device that can display data either in portrait mode, which is preferred for word processing; or landscape mode, which is preferred for spreadsheets. A user of the display manually rotates the dislay to select the display style. Because the scan direction in the monitor is the same regardless of the orientation of the display, the display hardware must change the order of presentation of bits to the display in portrait mode to maintain Macintosh software compatibility.

The display interface board contains the frame buffers in video RAM, a Xilinx FPGA and the digital to analog converter for preparing the signal for display [26] (Fig. 11). Part of the bitstream rotation is done when the pixels are stored in the dispay memory, part is done when the pixels are sent to the monitor. The VRAM addresses generated by the FPGA are different for portrait and landscape orientation. In addition, the data is scanned out of the video memory differently in each of the two display modes. The final bitstream sent to the monitor is 50MHz.

The Pivot display actually has six formatting modes: one, two or four bits per pixel, aligned either vertically or horizontally. The modes are all mutually exclusive, so one reprogrammable XC2018-100 FPGA with six different programs implements all six options. When the pivoting display orientation changes, an orientation-sensitive switch in the cabinet starts the reconfiguration process and the FPGA is reconfigured with the appropriate bitstream from the PROM that contains all six bitstreams. The XC2018 is rated at about 1800 gates, but this application replaces about six thousand gates of logic.

Teaching Advanced Logic Design

Instructional labs need reprogrammability and fast prototyping. All student projects are prototypes, and the more laborious the design implementation, the less students learn about design. Since 1988, Professor Pak Chan at the University of California, Santa Cruz has been using Xilinx FPGA's in an Advanced Logic Design class [27]. Professor Chan interfaced tools from the UC Berkeley Oct tool set including `bdsyn`, `misII` and `mustang` with placement and routing tools `APR` and `XACT` from Xilinx. Students use hardware description languages and schematic entry to specify the design, and use simulation to verify the designs. They implement the logic using placement and routing and load the design by programming an EPROM that they plug into the prototyping board with the LCA.

This teaching method has significant advantages over the more traditional solder-and-wirewrap method. Students use a design style very much like commercial ASIC design: they spend more time designing and less time implementing. Class projects include high-speed addition, multiplication, division and bit-slice processors. When the course is finished, the hardware used by students is reused.

The prospect of "soft hardware" for teaching may be upsetting to those who believe it is important for students to have a good, hard, practical understanding of the implementation technology. Although this understanding is important, so is exposure to a large range of design issues. The less time students spend on the mechanics of implementation, the more time they spend doing design. Computer science curricula have already addressed this issue: nearly all computer science projects use a high-level language, even though it is machine code that actually gets executed. Once or twice in a student's career, he or she is required to actually deal with assembly language and machine code, but the bulk of the teaching is intended to broaden the student's experience with large-scale "real world" problems. Reprogrammable FPGA's may allow electrical engineering courses to broaden the scope of exposure they give their students.

Large-Scale Applications

Several large-scale uses of reprogrammable gate arrays have been reported in the literature recently [28]–[33]. In this section, we describe a few of these systems and their

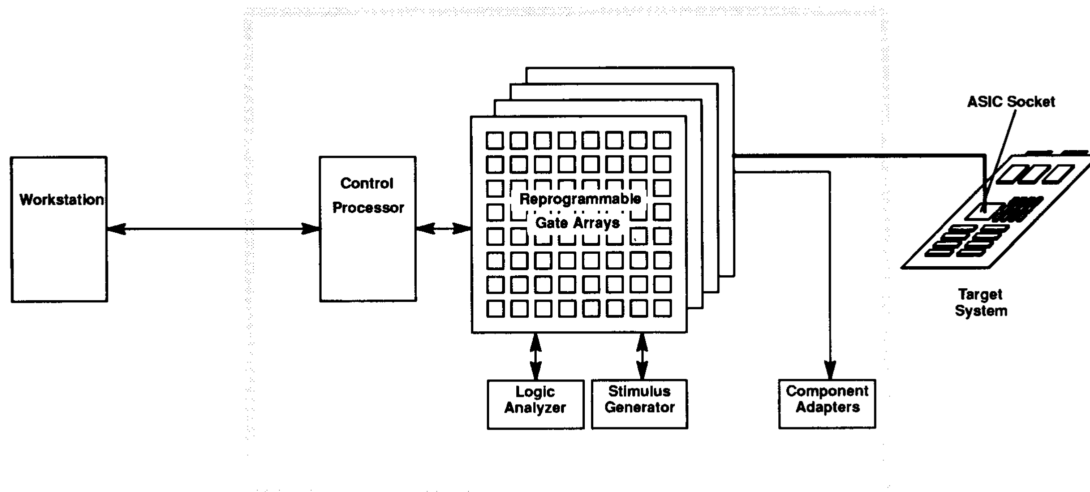


Fig. 12. RPM emulation system.

use of reprogrammable FPGA's. Readers are directed to the references for more details.

The Quickturn Logic Emulator

Quickturn Systems, Inc. addressed the problem of prototyping large-scale ASIC's and systems in their RPM Logic Emulator by using an array of LCA's [28]. The RPM hardware includes a SCSI or Ethernet connection to a control processor, an array of Xilinx XC3090 FPGA's and external component adapters (Fig. 12). The FPGA's in the array are connected in a hypercube arrangement with fixed interconnects at the board level.

The RPM software accepts a netlist of hundreds of thousands of gates, partitions the gates into FPGA's to optimize interconnection and density, places and routes designs on the FPGA's and adjusts the resulting timing to match that of the incoming netlist. The emulated design executes up to a million times faster than simulation. Small changes are handled incrementally across all FPGA's in the system, limiting re-implementation times to minutes. The RPM includes adapter cards that allow a designer to interface with external logic—IC's or complete systems. It includes configurable signal generators and analyzers also built out of LCA's to monitor the running system.

Prototyping Hardware/Software Algorithms

Perle-0 is a platform for experimentation with logic and architectures for highly-parallel computation [29]. The board consists of an array of 25 reconfigurable FPGA's with local memory and bus interface circuitry. It has been used for a variety of algorithms, including image filtering, very-long word size arithmetic, RSA encryption and data compression. Changes in the algorithm may take weeks to describe and optimize, and design iterations can take several days. Compared to turnaround times for custom ASIC's, these delays are short, there are no tooling charges, and the hardware can be re-used. Because the designers can continually hone their algorithms, Perle-0 has achieved performance on some problems superior to the best custom IC solutions.

Another example of this kind of system is the Splash processor [30], [31]. Splash consists of an array of LCA's with external memory and a fixed interconnection pattern. The Splash processor is targeted to one-dimensional systolic problems and has been used for pattern matching DNA sequences. The general-purpose Splash board out-performs a Cray supercomputer by as much as a factor of 300 and out-performs a custom single-chip integrated circuit by a factor of 45.

The Anyboard is a general-purpose board for rapid-prototyping [32]. The hardware consists of several FPGA's with fixed connections to each other and to on-board memories. The FPGA array has a fixed interface to a host processor bus. The host loads programming and passes other instructions to the board. To support large-scale logic implementation, the Anyboard development environment includes a logic partitioner to divide the prototype system among the FPGA's and to generate programming for each part. The Anyboard has been tested with such diverse elements as a systolic linear convolver and an ALU.

Key to attaining high-quality designs in all the above systems is the ability to repeatedly prototype algorithms and to bring to bear large amounts of reusable hardware. The system designers built powerful debugging features into their systems to debug their highly-parallel designs. The debuggers access the read-back facilities on the LCA's to observe interior nodes. In some cases, read-back is used to unload the results of computations, saving the need to generate extra upload logic in the FPGA array.

A Flexible Processor

Wolfe and Shen [34] used several reprogrammable FPGA's to implement a "flexible processor." Eight reconfigurable FPGA's are used to implement instruction

decoding, address generation and datapath operations in a single-board computer. They can be configured to implement a wide variety of virtual processor architectures with different instructions, addressing mechanisms, pipelining schemes and ALU operations. Wolfe and Shen used the flexible processor for prototyping processor architectures for solving systems of linear equations, and the same flexible processor computer can be used to prototype other processor architectures.

Flexible processors may change our definition of algorithmic complexity. For example, a searching algorithm that requires $O(n^2)$ operations on a standard processor may require only $O(n)$ time on a flexible processor configured with an array of comparators. The optimal choice of processor configuration may be dynamic and problem-size dependent. The processor configuration may change during operation. Such a development has been called a new paradigm of computation [34], [35]. Reprogrammability gives a system designer new capabilities: reconfigurable hardware or even virtual hardware, similar to virtual memory in computer systems. Eventually such virtual hardware may be swapped or timeshared in a manner analogous to multiple-process-management in computer systems, employing a software-like operational methodology as well as a software-like design methodology.

V. Summary

Reprogrammable FPGA's provide a new implementation target for logic and digital systems. They can implement thousands of gates of logic at clock speeds in the tens of megahertz, and both these numbers are improving rapidly. New generations of FPGA's provide both powerful logic and system implementation features.

Reprogrammable FPGA's allow a change in the IC and systems design methodology to emphasize prototyping rather than simulation. FPGA's and the software that surrounds them include features that facilitate this methodology, and many current users of FPGA's have adopted these techniques.

Designers are using reprogrammable FPGA's to replace multiple functions on a board. The systems allow experimentation with large amounts of logic, systolic processors and computer architecture. These systems require reprogrammable logic, since they would not be cost effective if the hardware could be used only once.

References

[1] *The Programmable Gate Array Data Book.* Xilinx, 1989, 1991, 1992.

[2] W. Carter, K. Duong, R.H. Freeman, H.-C. Hsieh, J.Y. Ja, J.E. Mahoney, L.T. Ngo, and S.L. Sze, "A user programmable reconfigurable gate array," in *IEEE 1986 Custom Integrated Circuits Conference*, pp. 233–235,1986.

[3] H.C. Hsieh, K. Duong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, R.H. Freeman, "A Second generation user-programmed gate array," *IEEE 1987 Custom Integrated Circuits Conference*, pp. 515–521, 1987.

[4] H.C. Hsieh, K. Duong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, R.H. Freeman, "A 9000 gate user-programmable gate array," in *IEEE 1988 Custom Integrated Circuits Conference*, pp. 15.3.1–15.3.7, 1988.

[5] H.C. Hsieh, et. al., "Third generation architecture boosts speed and density of FPGAs," in *IEEE 1990 Custom Integrated Circuits Conference*, 1990.

[6] J. Rose, S. Brown, "Flexibility of Interconnection structures for field-programmable gate arrays," *IEEE J. Solid-State Circuits*, vol. 26, no. 3, Mar. 1991, pp. 277–282

[7] J. Rose, R.J. Francis, D. Lewis, P. Chow, "Architecture of field programmable gate arrays: the effect of logic block functionality on area efficiency," *IEEE J. Solid-State Circuits*, vol. 25, no. 5, Oct. 1990.

[8] J.-M. Vuillamy, Z.G. Vranesic, R. Rose, "Performance evaluation and enhancement of FPGA's," in *Proc. Int. Workshop on Field Programmable Gate Arrays*, Oxford, 1991.

[9] S. Trimberger, "Beyond logic—FPGA's for digital systems," in *Proc. Int. Workshop on Field Programmable Gate Arrays*, Oxford, 1991.

[10] W.R. Heller, W.F. Mikhail, W.E. Donath, "Prediction of Wiring Space Requirements for LSI," *J. Design Automation and Fault Tolerant Computing*, vol. 2, no. 2, pp. 117–144, May 1978.

[11] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in *Proc. 27th Design Automation Conference*, 1990.

[12] R.J. Francis, J. Rose, K. Chung, "Chortle: A technology mapping program for lookup table-based field-programmable gate arrays," in *Proc. 27th Design Automation Conference*, 1990.

[13] R.J. Francis, J. Rose, Z. Vranesic, "Chortle-crf: Fast technology mapping for lookup table-based field-programmable gate arrays," in *Proc. 28th Design Automation Conference*, 1991.

[14] [K. Karplus, "Xmap: A technology mapper for table-lookup field-programmable gate arrays," in *Proc. 28th Design Automation Conference*, 1991.

[15] N.-S. Woo, "A heuristic method for FPGA technology mapping based on edge visibility," in *Proc. 28th Design Automation Conference*, 1991.

[16] S. Trimberger, "Placement-driven partitioning for lookup-table-based field-programmable gate arrays," in *Proc. 1992 ACM Workshop on Field Programmable Gate Arrays*, 1992.

[17] S. Trimberger, *Field Programmable Gate Arrays.* Kluwer Academic Press, 1992.

[18] M.A. Breuer, "Min-cut placement," *J. Design Automation and Fault Tolerant Computing*, Oct., 1977.

[19] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by simulated annealing," *Sci.*, 13 May 1983.

[20] S. Goto, "An efficient algorithm for the two-dimension placement problem in electrical circuit layout," *IEEE Trans. Circuits Syst.*, Jan., 1981.

[21] N.J. Nilsson, *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[22] J. Soukup, "Circuit Layout," *Proc. IEEE*, Oct. 1981.

[23] M. Palczewski, "Plane Parallel A* Maze Router and its Application to FPGA's," in 29th Design Automation Conference, 1992, submitted for publication.

[24] s. brown, j. rose, z. vranesic, "A detailed router for field-programmable gate arrays," in *IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, IEEE 1990, pp. 382–385.

[25] G. Rosendahl, T. Paille, D. Freiling, R. McLeod, "In system reprogrammable lca's provide a versatile interface for a dsp based parallel machine," in *Proc. Int. Workshop on Field Programmable Gate Arrays*, Oxford, 1991.

[26] J. Tan-Nguyen, T. Oyama, N. Moss, "Pivoting monitor increases versatility of workstations," *Computer Techn. Review*, Fall 1990.

[27] "Using field programmable gate arrays in a second course on logic design," in *Proc. 3rd Microelectronics Systems Education Conference and Exposition*", San Jose, CA, pp. 37–42, July 1990.

[28] S. Walters, "Reprogrammable hardware emulation automates system-level ASIC validation," *Wescon/90 Conf. Record*, 1990.

[29] M. Shand, P. Bertin, and J. Vuillemin, "Resource tradeoffs in fast long integer multiplication," in *Proc. 2nd Annual ACM Symp. Parallel Algorithms and Architectures*, 1990.

[30] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *Computer*, Jan. 1991.

[31] D. Lopresti, "Rapid implementation of a genetic sequence comparator using field-programmable logic arrays," in *Advanced*

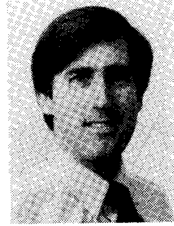
Research in VLSI: Proc. 1991 University of California Santa Cruz Conference, The MIT Press, 1991.

[32] D. Thomae, T. Petersen and D. Van den Bout, "The anyboard rapid prototyping environment," in *Advanced Research in VLSI: Proc. 1991 University of California Santa Cruz Conference*, The MIT Press, 1991.

[33] C.E. Cox and W.E. Blanz, "Ganglion, A Fast Hardware Implementation of a Connectionist Classifier," in *Proceedings of the Custom Integrated Circuits Conference*, 1991.

[34] A. Wolfe and J.P. Shen, "Flexible Processors: A promising application-specific processor design approach," Technical Report, Carnegie-Mellon University, 1988.

[35] J.P. Gray and T.A. Kean, "Configurable Hardware: A New Paradigm for Computation," in *Advanced Research in VLSI: Proc. 1989 Decennial Conference on VLSI*, The MIT Press, 1989.



Stephen Trimberger received the M.S. degree from UCI, where he worked in the dataflow group, and a Ph.D. in computer science from the California Institute of Technology, where he worked in the areas of nondeterministic computation and design automation.

He was a member of the original design team for the VLSI Technology ASIC design software and has worked on software for nearly every aspect of ASIC design. He has written two books on computer-aided design for integrated circuits and teaches computer-aided design at Santa Clara University.

From 1988 to 1990, Dr. Trimberger was a member of the architecture definition group for the Xilinx XC4000 FPGA and the technical leader for the XC4000 design automation software. He is currently Manager of Advanced Development at Xilinx where he is responsible for innovative and long-term FPGA architectures and software. His interests include design automation at all levels, software development methodology and innovative uses of reprogrammable FPGA's.