EMULATORS

An emulator (specifically a logic emulator) is a completely programmable hardware system, which can be programmed to emulate a large digital design, and operate that design in real time, as if it is real hardware. Logic emulators are used for real-time design verification, debugging, and analysis, for software development before actual hardware is available, and for architectural experimentation and development.

Emulated logic designs as large as many millions of gates can run at a multimegahertz clock rate, directly connected to the surrounding hardware system and also running actual applications and data. Internal signals are easily observed for debugging analysis. Design changes can be made quickly, without hardware modifications, and the emulator is reprogrammed with the new version. Emulation covers orders of magnitude more verification cycles than simulation, and its ability to verify in the real system environment with real code and data is unique. Emulators have become mainstream, commercially available and supported development tools used by hundreds of projects, for application-specific integrated circuit (ASIC) and full custom chip and board-level system designs.

From tens to thousands of field-programmable gate array (FPGA) chips, field-programmable interconnect device (FPID) chips and static random-access memories (SRAMs) are combined with software to translate, partition, and route logic design netlists into the hardware, and they are also combined with instrumentation for observation, testing, and debug. Logic emulators are the first widely used large-scale dynamically reprogrammable hardware systems. Emulators are intrinsically able keep up with the explosive verification demands of digital technology, even as design sizes double every 18 months according to Moore's law, because they are based in the same silicon technology that drives the design sizes themselves.

DEFINITION

Specifically, a logic emulator is a system of three major components: (1) programmable hardware, which consists of pro-

grammable logic and programmable interconnect, (2) compiler software, which automatically programs the hardware according to a gate-level or higher-level-language description, and (3) instrumentation and control hardware and software to support operation of the emulated design.

USAGE

Logic emulators are usually connected to a workstation computer or a local area network (LAN) of workstations. The design compiler and the graphical user interfaces for run-time instrumentation and control generally run on one or more of these workstations. A typical system is shown in Fig. 1.

The emulator may be connected to the target hardware, in the place that the emulated design will operate after it is built, and the entire system, including the emulated design, can then actually run in live hardware form. Commonly the design for a custom chip or ASIC is being emulated, and the emulator is plugged into the socket which will hold the actual chip after fabrication.

Since the hardware is programmable, the emulated circuit's speed is considerably lower than the real circuit's speed. Typical clock frequencies are between 100 kHz and 10 MHz, depending on the emulation technology used. Many techniques have been developed to slow down the target system's clock rate to match the emulator. Otherwise, the circuit operates the same way in emulation as in reality. The design may be operated with real applications and real data, just as the final permanent version of the hardware will be.

Logic emulators are also commonly used as ultrafast test vector evaluators. Rather than being connected to other hardware, a series of vectors of input values are applied to the inputs of the emulated design, and output vectors are collected from its outputs. These vector sets might be either the test vectors needed for testing the chip after fabrication or predefined vector sets for compatibility and regression testing. These vectors can be applied at very high speed, since the emulated design is operating at hardware speed, so very large vector sets may be evaluated in a short time. Emulators used for vector evaluation typically have very deep and wide vector memories, supporting thousands of channels for tens or hundreds of thousands of vectors, with connections for streaming more vectors in and out of the host computer's disk storage.

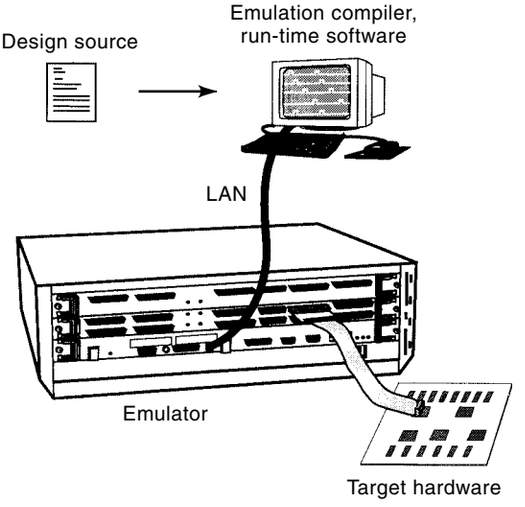

Figure 1. Typical logic emulation system.

Emulators are usually used to verify a logic design after some amount of software-based logic simulation has been done. First, the design is loaded by the emulation compiler software. The user specifies how the design's inputs and outputs should map to pins on the in-circuit cable, specifies any internal signals to be connected to the emulator's logic analyzer, and identifies critical paths and clock nets. Then the compiler automatically translates the design into the binary programming for the FPGAs and FPIDs in the hardware, and it creates an emulation design database for use in operation and in recompiles. If the user has input and output vector sets available from earlier simulation, these may be run on the emulated version of the design to validate it. Then the emulator is connected in-circuit to the target hardware, and the emulated design is operated in real time. The emulator's built-in logic analyzer displays any internal signals that the user wishes to observe, using complex user-defined trigger conditions. Small design changes can be made by incremental recompilation, which takes less time than the initial full compile. Once the user is satisfied with the design, it can be released for fabrication. Even after the real chips are available, the emulator provides a real-time in-system analysis environment that provides internal visibility to the design.

TYPES OF EMULATORS

There are currently two major types of logic emulators: FPGA-based and processor-based. Most emulators are based on FPGA, FPID, and SRAM chips. Every gate, flip-flop, memory cell, and wire in the emulated design is mapped onto a specific programmable logic, memory, or interconnect element. FPGAs may be interconnected by FPIDs. Once the hardware is programmed, it is a live instance of the design in actual hardware, a kind of automatically generated prototype. FPGA-based emulators have the fastest operating speed, and they can emulate practically any logic structure or clocking scheme in the input design.

The processor-based type of emulator is actually a very high-speed hardware-accelerated logic simulator. Dedicated parallel processors repetitively execute the logic equations of the design. Input signals, from the in-circuit connection or from vectors, are continuously translated into input data for the processors, and processor output data are continuously driven onto in-circuit or vector outputs. The processors are fast enough to emulate real-time operation of the design. Processor-based emulators generally have large capacities and fast compile times, but they are much slower than FPGA-based emulators, and they are not capable of emulating designs with complex clocking or unclocked internal feedback paths.

CAPABILITIES AND COMPARISONS

Design Verification Tools

It is vital to verify the correctness of a chip design before it is fabricated. Substantial amounts of time and money stand

between releasing the logic design to the chip foundry and operating the resulting chip in the system. Once operating, internal signals are not directly available for observation and analysis, so diagnosing errors after fabrication is often very difficult. Even the slightest design error must be corrected by going through another fabrication cycle, at considerable expense and delay. Studies have proven that a few months of delay in getting a new product to market can cost a large fraction of the product's total lifetime sales. The premium on getting the first silicon fabricated correctly is high.

Logic simulation programs, running on desktop workstation computers, are widely used to verify designs. A logic simulator takes design netlist files, along with signal inputs in the form of vector data files, and calculates how the logic design would behave over time, given those inputs. The designer observes the outputs predicted by the simulator to see if the design is operating correctly. If incorrect operation is observed, the simulated internal circuit activity can be displayed, design errors found, and corrections made to the design, rapidly. Once enough operation has been simulated to give confidence in the design's correctness, it may be released for fabrication.

Verification Coverage

Simulation provides enough verification for designs with tens of thousands of gates; but by the 1990s, designers were faced with verifying chip designs with hundreds of thousands and even millions of gates, operating in systems with other such chips. Simulation remains a valuable tool during the design process, for initially verifying each module of a design, and for doing the initial verification of the entire design. However, many logic designs are now too large to completely verify using simulation alone. This is because the simulation workload is increasing much faster than the processing power of conventional computers. Logic emulation is capable of providing the trillions of cycles required to fully verify current and future logic designs.

When the size of a logic design doubles, the amount of computing work to sufficiently simulate the design roughly quadruples. Doubling the number of gates roughly doubles the amount of processor time required to simulate each cycle of operation. But doubling the size of the design also roughly doubles the number of operation cycles needed to verify its operation. The result is that the amount of processor work to fully verify a logic design by simulation goes up as the square of the size of the design. As observed by Moore's law, design sizes grow by a factor of two every 18 months, four every 3 years, and 100 every 10 years. Thus simulation processor work grows by a factor of four every 18 months, 16 every 3 years, and 10,000 every 10 years. This is a much faster pace than even the rapid growth in processor performance over the same time.

Each logic gate is represented by programmable silicon in the logic emulator. As design sizes grow due to Moore's law, the capacities of FPGA and FPID chips also grow in the same proportion. Logic emulation technology is intrinsically able to keep up with design size growth, as well as maintain system emulation cycle rates over 1 MHz.

Real-Time Operation

Some applications must be verified in real operation in real time. Video display outputs cannot be verified by inspecting simulator output in the form of vector files or waveform traces, but instead require actual observation. Extremely large amounts of output data are needed for even a small amount of display operation. More and more systems, such as audio and video compression, depend on qualities of human perception. It is difficult to verify such designs with software simulation alone. Real-time operation is a natural characteristic of hardware emulation. It is directly capable of supporting real-time video displays, audio devices, and perception-based verification. There have been a number of cases where a subtle design error was identified within an hour of emulated operation, by directly hearing or seeing its effect on the design's output, which the users have said could never have been caught in simulation.

Test System-Level Interactions

A common problem in developing large chip designs that run in complex systems is when the chip design meets specifications but fails in the system, due to misunderstandings or unanticipated situations. A specification only represents its writer's understanding, and misunderstandings between designers result in system-level malfunctions. Sometimes other parts of a system aren't well-specified. Frequently, real system operation presents unanticipated situations that aren't covered in the test vectors. As complex chips interact in real systems, the number of combinations of operational situations explodes combinatorially. For example, in a networked virtual memory computer system, there could be an error that only occurs when the Ethernet driver interrupts a page fault, which is servicing a floating-point exception. With logic emulation, the design is being verified in the actual hardware environment in which it will be used. No human assumptions are involved in this verification. Trillions of cycles of verification are available to cover situational combinations. Verification has much higher reliability as a result.

Internal Design Visibility

Once the chip design is fabricated and placed in a system, if it fails, internal probing is impossible. It may be hard or impossible to get the simulator into the failing state, because it depends on a complex set of conditions or takes many millions of cycles of operation to get to. Internal nets may be connected to the emulator's logic analyzer via the programmable interconnect, so the design's internal operation may be observed and analyzed during real operation with real applications and data. This is a powerful capability for both chip and system-level debugging. Emulators usually provide hundreds or thousands of channels of logic analysis, providing rich visibility inside the design. This unique capability is widely used, even after the design has been fabricated, to analyze system-level bugs from inside the design, which is otherwise impossible. Some emulator users have found post-silicon emulation for analysis as valuable as the pre-silicon emulation for verification (1).

FPGA-BASED LOGIC EMULATORS: HARDWARE ARCHITECTURE

FPGA-based logic emulators (Fig. 2) typically have one or more board-level logic modules, each of which has a large

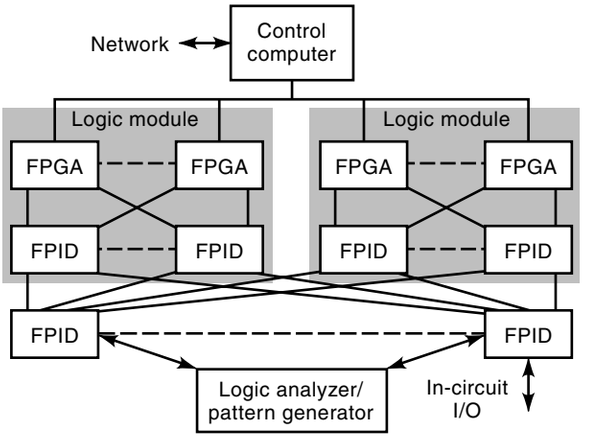

Figure 2. Block diagram of an FPGA-based logic emulation system.

number of FPGAs; usually each module has FPIDs to interconnect them. The logic modules are often interconnected by a system-level set of FPIDs. Several levels of programmable memories, along with facilities to connect user-supplied hardware (such as a processor core), inside the emulation are often included. Pattern generators to provide input vectors, along with logic analyzers to capture result vectors, are also usually included, and they are specialized for the logic emulation application. An in-circuit input–output (I/O) cable connects the emulator to the larger hardware system in which the emulated design is destined to be installed. One or more network-accessible control computers oversee all this hardware, programming the FPGAs and FPIDs and controlling the instrumentation.

Logic

Most emulators use reprogrammable FPGAs to emulate the design's logic gates and registers. An FPGA is a very flexible, completely programmable logic chip (2,3). FPGAs contain programmable logic blocks, programmable interconnect, and programmable I/O pins. While some types of FPGAs use nonvolatile programming, naturally emulators only use electronically reprogrammable FPGAs based on SRAM technology. To be useful in an emulator, an FPGA needs to have reprogrammable logic gates and registers, a reprogrammable way to interconnect them, and a way to freely program connections to I/O pins.

An FPGA has few actual gates at all. It is really an array of programmable logic blocks, usually in the form of RAM lookup tables (LUTs) and flip-flops, interconnected by metal lines and RAM-controlled interconnect cells (Fig. 3). An LUT is a 2^n-by-1-bit RAM whose address inputs are connected to the LUT signal inputs. It is programmed with a truth table to act as an arbitrary n-input logic function. Typical FPGA LUTs have three, four, or five inputs. One to four of these LUTs and a similar number of flip-flops or latches are interconnected together with programmable multiplexers to form a logic block.

Typically, a two-dimensional array of logic blocks is interconnected by metal lines of various lengths; these blocks pass either transistors or multiplexers, controlled by SRAM programming cells. Programmable I/O pin buffers line the FPGA's perimeter.

Computer-aided design (CAD) software is used to compile arbitrary netlists into programming binaries. The FPGA compiler maps netlist gates into LUTs and flip-flops, partitions them into logic blocks and places them in the array, maze-routes the interconnect, and generates the binary programming file.

Beyond the basic framework of logic blocks, programmable interconnect, and I/O, additional features, such as memory (either in dedicated blocks or by using the LUTs as read/write random-access memory (RAM), interblock arithmetic carry structures and wide decoders, and internal tri-state bus drivers, are usually included.

The die area of an FPGA chip is dominated by the SRAM programming cells and metal interconnect that make them field-programmable. Actual speed and capacity vary over a wide range depending on design characteristics. From 10 to 20 programming cells per equivalent logic gate are required. Of these, typically only 10% define logic functions; the other 90% are needed for programmable interconnect. Consequently, the total area penalty of an FPGA over hardwired logic in the same process is on the order of 15 to 30 times.

Programmable interconnect also makes FPGAs slower than hardwired logic. Worst-case delay through a logic block is in the 2 ns to 3 ns range, and interblock wiring delays of up to 10 ns or more are common (as of 1998). The speed penalty is very design-dependent, but is substantial, in the 3× to 10× range.

Because of the speed and area penalties of dynamic reprogrammability, FPGA cost/performance is usually one to two orders of magnitude worse than that of an ASIC or full-custom chip made with a similar process. This translates into a similar difference between the cost and performance of a logic emulator and the chip being emulated. The cost is more than justified by the logic emulator's verification capabilities.

Interconnect

The most challenging and important aspect of logic emulator design is the interconnect architecture. Logic emulators must

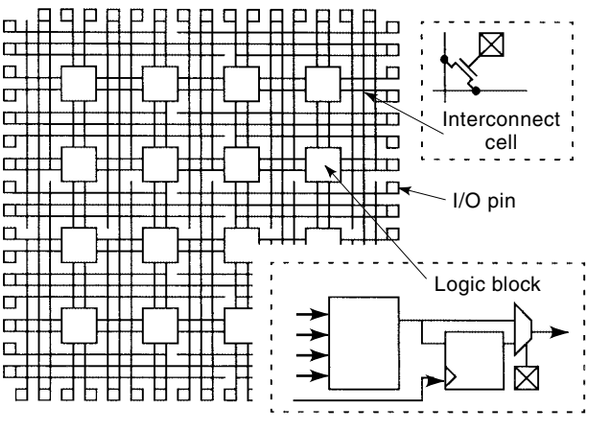

Figure 3. Generic representation of the internal architecture of an FPGA, showing a two-dimensional array of logic blocks and programmable I/O pins, interconnected by metal lines of various lengths, with programmable interconnect cells. The inset shows a simplified logic block, with a lookup table and a flip-flop.

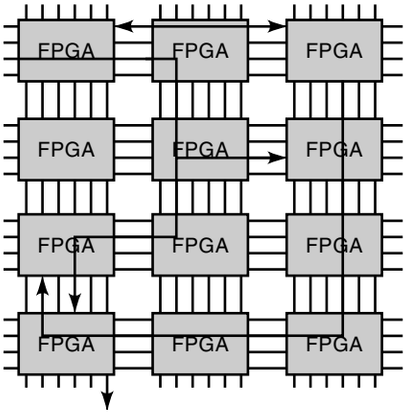

Figure 4. FPGAs in a nearest-neighbor interconnect. The bold lines represent example design nets, along with the paths they follow through intermediate FPGAs as they are routed from source to destination FPGAs.

always use multiple FPGAs, since a single FPGA can never have as many gates as an ASIC or full-custom chip design, made with the same process. This is because of the innate programmability of the FPGA. The pass transistors or multiplexers that carry an FPGA's signals, and the programming cells that control them, are much larger than a simple metal line. This is always true, regardless of the semiconductor process used. As silicon technology grows, the capacity of FPGAs will grow. But the size of designs to be emulated will grow as well, for the same reason. Therefore, logic emulators will always have multiple FPGAs.

These FPGAs must be interconnected in a way which is completely programmable, capable of interconnecting any logic design without introducing excessive delay or skew, scalable to a wide range of design sizes, and affordable. A number of architectures have been developed to address the multi-FPGA interconnect problem.

Nearest-Neighbor Interconnect. A simple way to interconnect an emulator's FPGAs is to continue the FPGA's two-dimensional array internal architecture and place a similar array of FPGAs, connected in a "nearest-neighbor" fashion, on the logic module board (Fig. 4). The interconnect I/O pins of each FPGA are connected to pins of nearby FPGAs. Most pins are connected to the pins of immediately neighboring FPGAs. Some may be connected to the next-most neighboring FPGAs for longer distance runs across the array. Logic modules are connected to one another in a similar fashion.

In the nearest-neighbor interconnect (4), the FPGAs are used both for emulating logic and for interconnecting signals. After the design has been technology-mapped into FPGA primitive form and then broken into FPGA-sized partitions, these partitions are placed into specific FPGAs in an optimized placement to minimize the routing distances in the array of inter-FPGA nets.

The earliest commercial logic emulator, the Quickturn RPM, used this architecture (5). It successfully emulated Intel's first Pentium CPU design, running an operating system and real applications many months before first silicon was available (6).

Rent's Rule Limitations. Unfortunately, there are a number of limitations and disadvantages with the nearest-neighbor architecture. FPGAs have a very limited number of I/O pins, since chip bonding pads and packages are much larger than the metal lines inside the chip. Logic emulators must also deal with the fact that when a complete chip-level logic design is automatically partitioned into many FPGA-sized pieces, each piece will usually have many more pins than a complete FPGA-sized design will. This is because logic designers naturally organize their designs to match the constraints of the chip packages they will reside in. Inside a chip-level module, interconnections are rich. When this chip-level module is cut by software into many FPGA-sized partitions, in a way unforeseen by the designer, each partition will cut many internal signal nets which must pass through FPGA pins. This effect is quantified by Rent's Rule (7), which is an empirically determined relation between the number of gates in a subpartition of a module and the number of pins required for the signals passing in and out of it. In FPGA-sized and board-sized partitions for emulation applications, experience has shown this form of Rent's Rule applies:

$$P = KG^r \tag{1}$$

where P is the number of pins, G is the number of gates, reduced to the equivalents of 2-input nand gates, r is the Rent exponent, typically between 0.5 and 0.7, and K is a constant, typically between 2.5 and 3.

Frequently the gate capacity of an FPGA in an emulator is limited more by this I/O pin constraint than by the FPGA's internal logic capacity. Therefore, the I/O pins of an emulator's FPGAs are a precious resource.

Nearest-Neighbor Interconnect Characteristics. Nearest-neighbor interconnected emulators use only FPGAs and few, if any, FPIDs. The printed circuit board is simple and inexpensive, since the wiring is short and regular. Interconnects which need only one direct path between neighboring FPGAs are fast and inexpensive.

Since the FPGAs must be used for routing inter-FPGA signals, as well as for logic, each FPGA's pins are in demand for two purposes: routing signals in and out of the logic partitioned into the FPGA and through-routing inter-FPGA signals of other FPGAs. In practice, this pin demand is a severe constraint on using the available logic capacity, and FPGAs are badly underutilized as a result. This overwhelms the savings from simple circuit boards and avoiding FPIDs, since many times more FPGAs are required for a given emulation capacity than the FPGA capacities alone would indicate.

Interconnection paths vary over a wide range, depending on the distance needed through the array. A placement program is required as part of the emulation compiler, which can take a long time to execute. It is never able to keep all inter-FPGA routes short, since logic circuits have a very irregular topology, little constrained by wiring, since permanent wire traces on chips are plentiful and inexpensive. Some routes end up taking long and circuitous paths through many FPGAs in the array, which results in very long interconnect delays on some nets. Not only does this slow operation, but the wide variance among net delays can induce incorrect behavior in some designs.

Emulation is most beneficial in verifying the largest designs, but the long routing paths make it impractical to scale

a nearest-neighbor interconnected emulator up to many hundreds of FPGAs, which is needed to handle the largest chip designs and multichip systems.

Full and Partial Crossbar Interconnects. The recognition that interconnect architecture is the key problem in logic emulation technology, because of the scarcity of FPGA pins and the cost and delay of programmable interconnects, motivated development of a different architecture. The partial crossbar interconnect made large-scale, efficient logic emulation practical and is the most widely used architecture today.

FPIDs. With crossbar-type interconnects, the emulator's FPGAs are interconnected by FPIDs. FPGAs themselves may be used as FPIDs, since they have an internal programmable interconnect among their I/O pins. However, they require CAD routing, and propagation delays may be difficult to predict.

There are also special-purpose FPID chips. These usually contain a single complete crossbar, which is an array of programmable switches than can interconnect all the pins of the FPID. Programming a crossbar is a simple table-lookup operation, and propagation delays are usually constant regardless of routing or fanout.

Full Crossbar Interconnect. To maximize the use of the FPGAs' scarce I/O pins, and thus their logic capacity, the pins should only be used for interconnections in and out of the logic in each FPGA. A separate structure for interconnecting FPGA pins is called for. This interconnect should be capable of automatically routing all logic design networks with nearly 100% success, with minimum and bounded delay, should be scalable to interconnect up to thousands of FPGAs, and should be economical.

In theory, a crossbar is the most complete and ideal interconnect. Crossbars are well known in communications technology, deriving originally from telephone central office switches. A crossbar consists of a regular array of programmable crosspoint switches, connecting each pin with all other pins. By definition, a crossbar can route any network with only one stage of delay. Figure 5 shows a very simple example (to fit into the figure) of four FPGAs with eight I/O pins each, interconnected by a full crossbar.

The problem in practice is that the size of a crossbar increases as the square of the number of pins. The number of crosspoint switches S in a bidirectional crossbar, where each switch can pass signals in either direction, which interconnects P pins, is

$$S = P(P-1)/2 \tag{2}$$

Since the switches that connect pins of the same FPGA are unnecessary, this can be reduced slightly. The number of crosspoint switches S in a bidirectional crossbar that interconnects N FPGAs with P pins each is

$$S = N(N-1)P^2/2 \tag{3}$$

To interconnect 20 FPGAs, each with 200 I/O pins, as would be used on a single board, a 4000 pin crossbar is required, which must have 7,600,000 crosspoint switches. For a system of 400 FPGAs, each with 200 I/O pins, an 80,000 pin crossbar is required, which must have 3,192,000,000 crosspoint switches. This is far in excess of what is practical, both in switch count and in pins, in a technology comparable to that of the FPGA.

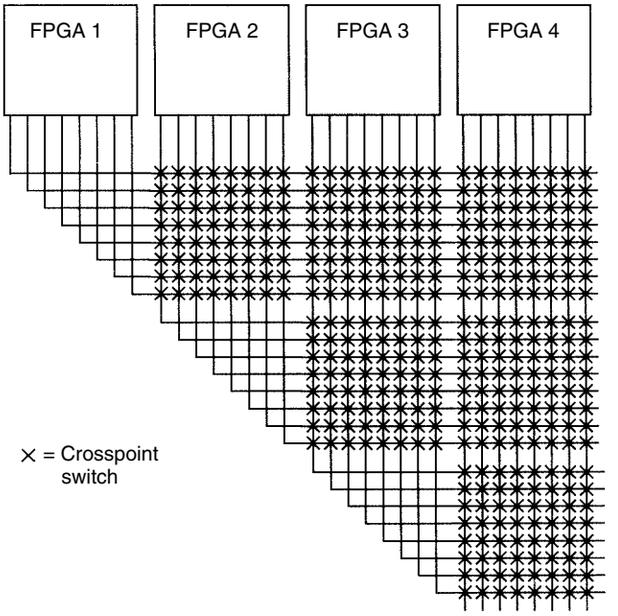

Figure 5. Four eight-pin FPGAs interconnected by a full crossbar. All the crosspoint switches that make up the full crossbar are shown.

Partial Crossbar Interconnect. A full crossbar can route much denser networks than are needed for normal logic designs. It can connect any pin with any or all other pins with equal ease. Typical nets in logic designs connect an output with a few inputs. A tiny fraction of crosspoint switches would ever be turned on to route a logic design.

Since an FPGA can freely interconnect internal signals to any of its I/O pins, there is flexibility available in the FPGA, which is not taken advantage of by a full crossbar. The partial crossbar interconnect (8,9) takes advantage of both these facts.

Figure 6 shows the earlier full crossbar example of four FPGAs, with eight I/O pins each, interconnected by a partial crossbar interconnect. In the partial crossbar interconnect, the I/O pins of each FPGA are broken into subsets. Only the crosspoint switches that interconnect FPGA I/O pins of the same subset are used. In the figure, the FPGAs' eight I/O pins are broken into four subsets, A, B, C and D, of two pins each. Each subset's crosspoint switches have FPGA I/O pins in common, so they may be grouped together into crossbars. Each resulting crossbar interconnects the pins of one subset of FPGA I/O pins.

Figure 7 has this same simple four-FPGA example, redrawn to show the partial crossbar interconnect in crossbar form. Each subset's crossbar is in the form of an FPID, which interconnects two pins from each of the four FPGAs. Any FPID may be used to route a net from one FPGA to others. Choosing an FPID for the route determines which I/O pin subset is used on the FPGAs. For example, a net running from FPGA 4 to FPGA 1 may be routed through any of FPIDs 1, 2, 3, or 4, using the FPGA I/O pins that connect to the FPID selected. In the figure, subset C is the choice, so one of the I/O pins from subset C is assigned to the net in FPGAs 1

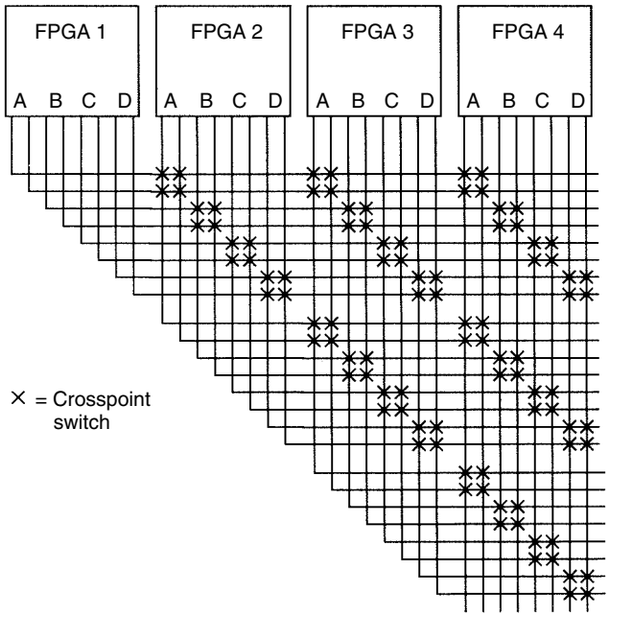

Figure 6. The same four FPGAs from Fig. 5, interconnected by a partial crossbar interconnect. The FPGA's pins are broken into four subsets of two pins each. Only the crosspoint switches that interconnect pins in the same subset are used.

and 4, and FPID 3 is programmed to interconnect the wires leading from those two FPGA I/O pins. All the inter-FPGA nets in a design are routed this way, one by one, largest first.

Additional pins on each FPID are used for external I/O connections in and out of the multi-FPGA network, for connections to in-circuit cables, instrumentation, and additional interconnect.

Using a partial crossbar interconnect, the board-level example of 20 FPGAs with 200 pins each can be interconnected by 50 FPIDs with 80 pins each for FPGA I/Os. The subset size is four pins, which is a size that has been successful in production experience. Each FPID has four pins connected to each of the 20 FPGAs; and 24 pins remain for external I/O, totaling 1200 for the board. Each 104-pin FPID has 5356 crosspoint switches, which is an easily built device. The total of 267,800 crosspoint switches among all 50 FPIDs is 30 times less than the full crossbar's 7,998,000, plus it is broken into easily packaged FPIDs.

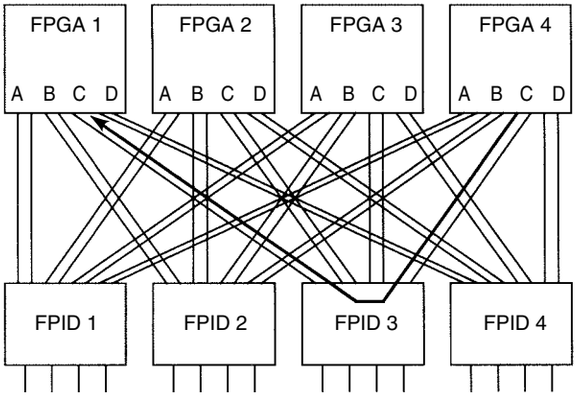

Figure 7. Four FPGAs in a partial crossbar interconnect. The same FPGAs and interconnect from Fig. 6 are redrawn to show the partial crossbar interconnect in crossbar form, with the crosspoint switches collected into FPIDs.

Partial Crossbar Interconnect Characteristics. Partial crossbar interconnects maximize the use of the FPGAs' logic capacity by preserving an FPGA's I/O pins for only nets that connect with its own logic. It maintains the full crossbar's ability to route all nets with one stage of delay. Routing the network is a simple tabular-based process, with some ripup and retry at the end for dense cases, which is very successful in practice. Since the network is fully symmetrical, no placement stage in the compiler is needed to decide which partition to put in each FPGA. The partial crossbar interconnect is economical and very scalable.

The penalties are (1) the extra cost and size of the FPIDs, (2) the fact that many wires on the printed circuit board are long, making it more expensive, and (3) the fact that direct connections between FPGAs are not available.

Hierarchical Partial Crossbar Interconnects. Large multiboard emulators with hundreds of FPGAs cannot be reasonably interconnected by a single partial crossbar interconnect. Since each FPID is connected to every FPGA, the network cannot be broken into multiple boards without cutting a large number of wires. Instead, the partial crossbar interconnect architecture can be applied recursively, in a hierarchical fashion.

Each group of partial crossbar interconnected FPGAs and FPIDs is itself like a very large FPGA. It has I/O pins, on the FPIDs; these pins can be freely used to connect with logic inside, as with an FPGA. A set of such groups can be interconnected by a second level of FPIDs, as shown in Fig. 8. Four first-level partial crossbar interconnected groups, each like the one in Fig. 7, have their external I/O pins broken into subsets of two each, which are interconnected by the second-level FPIDs. Each group could be on one board, and the boards interconnected in a card cage, by FPIDs on the backplane itself, or on additional boards which are mounted at right angles to the FPGA boards on the other side of the backplane. The number of external I/O pins in the group is determined by applying Rent's Rule to the expected logic capacity of the group.

Nets which pass between FPGAs in different groups are routed through three FPIDs: the one on the source board, the second-level one, and the one on the destination board. This

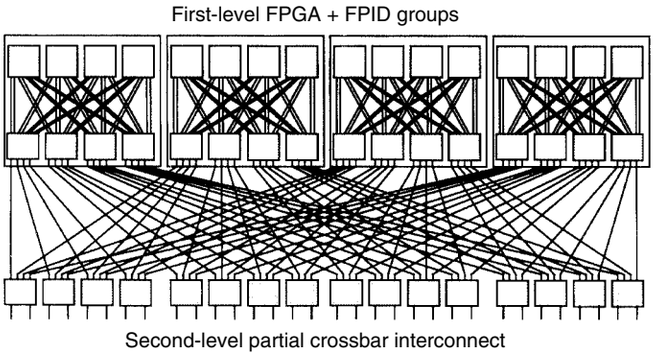

Figure 8. Hierarchical partial crossbar interconnect.

way, hundreds of FPGAs can be interconnected: Most routes take a single stage of FPID delay, and the rest only need three stages.

The second-level FPIDs may also have additional pins for external I/O connections. As many levels of hierarchy as needed may be used to interconnect a system of FPGAs of any size, efficiently and economically.

The earlier example of 400 FPGAs with 200 pins each can be effectively interconnected by a two-level partial crossbar interconnect. The earlier partial-crossbar interconnected board of 20 FPGAs and 50 FPIDs, with 1200 external I/O pins, is taken as the first-level group. Twenty such boards, making up the total of 400 FPGAs, are interconnected by a second level of 300 FPIDs, with 80 pins each, again connecting four pins to each of the 20 boards. There are 20 times 267,800 crosspoint switches in all the first-level FPIDs, plus 300 times 3160 crosspoint switches in the second-level FPIDs, totaling 6,304,000, which is 506 times fewer than the full crossbar would require. Interconnects of this type and size have been used very successfully in production logic emulators, containing over 1000 large FPGAs.

Memory

Often designs to be emulated include random access memory (RAM) and read-only memory (ROM). These memories can take a very wide variety of shapes and sizes from design to design. Emulator hardware usually includes a range of facilities to emulate all memories in the design.

Most LUT-based FPGAs offer the ability for logic block LUTs to be used directly as small RAMs, instead of for logic. These are small, usually only 16 or 32 bits each, but they are very flexible. Very tall or wide memories can be constructed out of smaller RAM primitives, with additional logic programmed into other logic blocks for decoding addresses and multiplexing data outputs. The main limitation is total size. A few thousand bits of memory, along with the logic required to assemble it into one memory block, can consume an entire FPGA.

Conventional SRAM chips are also included to emulate larger memories more efficiently than FPGAs can. They are connected to FPGAs and FPIDs in many ways in different emulators, to take actual SRAMs with a fixed number of words and bit width and make them emulate a wide variety of different memories in designs.

More and more multiported memories are coming into use, where the same memory array can be read and/or written to, in many locations at once. Memories in full-custom chip designs often have extreme bit widths and large numbers of ports, features that are not directly realizable in FPGA or standard SRAM devices. Emulation of these memories can be a complex task. Multiporting can be emulated by rapidly time-multiplexing a single or dual-ported RAM. Each port is serviced in a "round-robin" fashion at a high enough rate (compared with the speed of the emulated design's clock) that multiport operation is accomplished.

Control and visibility features are also generally included to load and unload data to and from the memories and to provide interactive visibility into the memories, in the manner of a debugging console. If the visibility port is emulated as an additional port of a multiport memory, then the debugging visibility can be freely used during operation without interfering with the design.

User-Supplied Hardware

Often the emulated design contains one or more modules that already exist in hardware form. There is no point in consuming emulator capacity for these, and usually their internal logic designs are not available anyway. One example is an ASIC that contains a core, such as a processor or a bus interface, which is available in "bonded-out" form as a real chip. Another is a board-level design containing off-the-shelf chips. Emulators contain facilities for these to be mounted on cards and connected to the hardware. Programmable connections using FPIDs interface the fixed I/O pin locations of the user-supplied hardware to the emulator's interconnect.

Instrumentation

Two main facilities are usually provided for connecting the programmable logic and interconnect with inputs and outputs: (1) in-circuit cables for real-time operation in the target hardware and (2) logic analyzer/pattern generator facilities for running test vector sets and for observing signals during real-time operation.

In-circuit cables directly connect the emulated design with the actual target hardware that the design will run in once it is fabricated. The emulated hardware receives signals from and drives signals to its live, running hardware surroundings. When the emulated design is to be a packaged chip, adapters are available to plug the in-circuit cable into the actual socket of the actual board where the chip will be. Alternatively, some emulation users choose to build an emulation-specific prototype board for the target, and they provide flat cable connectors for in-circuit cables. Programmable FPIDs interface the fixed in-circuit I/O pin locations with the emulator's interconnect.

Emulators usually include pattern generators and logic analyzers. They are used in stand-alone operation, without the in-circuit connection, to drive the emulated design with test vector inputs and capture test vector outputs for analysis and comparison. While this capability is similar to that of a simulator, the emulator's megahertz speed allows very large vector sets, for regression and compatibility testing, to be run in a far shorter time than on a simulator. The logic analyzer is also used during in-circuit operation as debugging instrumentation. Internal signals may be identified, and automatically routed out to logic analyzer channels.

Logic emulation places different demands on logic analyzers and pattern generators than ordinary benchtop operation with conventional instruments. Since emulation speeds are slower than real hardware, the capture rate can be lower, typically no more than 20 MHz. Only simple logic levels need to be observed. On the other hand, many hundreds of channels, with very complex triggering conditions, are called for to use the rich visibility into design internals that the emulator's programmable interconnect can provide. In contrast, standard benchtop instruments are very fast, have relatively few channels, and would call for cumbersome cabling to connect to the emulator. Therefore, most emulators include built-in logic analyzer and pattern generator facilities, tightly integrated with the emulator's interconnect, with hundreds or thousands of channels and hundreds of thousands of vectors

of depth. Emulation compilers program the interconnections to these instruments automatically, and they allow a well-integrated emulation run-time environment to be used with the same signal names as in the design source.

Control Facilities

One or more local control microcomputers directly control the FPGA and FPID programming process, control and access the pattern generator, logic analyzer, and memory visibility ports, run diagnostic programs on the hardware, and do any other low-level control, visibility, programming, or diagnostic functions. They are usually connected to the local area network (LAN) for communication with the users' run-time control and debugging programs. Usually one or more programmable clock generators are provided as sources for clock signals for the emulated design, pattern generator, and/or logic analyzer.

FPGA-BASED LOGIC EMULATORS: SOFTWARE ARCHITECTURE

Compiler Software

The logic emulator's design compiler is among the largest and most complex of all electronic design automation tools. Its major components and execution flow are shown in Fig. 9. The compiler accepts an input design, expressed as many files in many different libraries and/or hardware description languages, assembles a single fully expanded design representation, maps it into the FPGA logic technology, analyzes it for potential timing problems, partitions it into boards and FPGAs, places the FPGAs if necessary, routes the board-level interconnect, and then runs a chip-level place and route for each FPGA and FPID, finally creating a comprehensive emulation database for the design containing the FPGA and FPID programming binary bitstreams and reference information about the design to support the user's debugging at run-time. Some emulation compilers now can accept a register-transfer-level representation in a hardware description language (HDL), such as Verilog or VHDL. They have an additional front-end synthesis step, and create additional HDL linkage information in the emulation database for use at run-time. The compiler must do all this completely automatically, completely reliably, and making efficient use of the hardware capacity. Since the emulation user is only interested in using the emulation, not in internal details of the FPGAs and FPIDs, complete automation is desired, making this a more challenging task than the usual chip design tool faces.

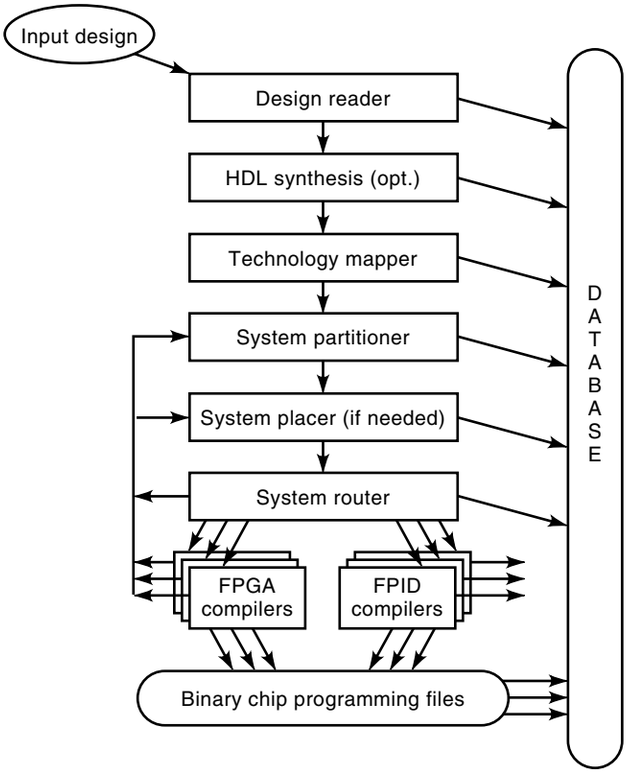

Figure 9. Major components and execution flow of the FPGA-based logic emulation compiler.

Compilation begins with a front-end design reader and checker. It reads in the design files, which may be a large hierarchical collection of netlists, and builds a completely expanded single-level version of the design in the emulation database. Usually one or more ASIC or cell libraries are called for by the design. The emulation compiler includes these libraries and expands library elements out to the fully primitive level. The design is checked for internal consistency. Nets which are to be connected to in-circuit cable pins, or to logic analyzer or pattern generator channels, are called out by the user and included in the design database at this stage. Some compilers include an HDL synthesis capability, which is discussed in the section entitled "advanced topics."

Technology mapping is done to translate from the ASIC or cell-specific logic primitives into FPGA-compatible primitives. For example, if the FPGA-level place and route tool only recognizes logic gates with five inputs or less, larger gates in the design are broken down into FPGA-acceptable smaller ones. Most FPGAs do not directly include transparent latch primitives, so if necessary latches in the design are translated into an equivalent network of cross-coupled gates. If the design includes nets with multiple drivers, such as tri-state or bidirectional nets, some emulation compilers will translate those nets into a logically equivalent unidirectional sum-of-products form. Even if the FPGA has internal tri-state buffers available, using them often severely constrains internal logic placement in the FPGA, thereby impacting logic capacity. Such nets often span many FPGAs, and maintaining tri-state form is difficult to accomplish across many FPGAs and FPIDs. Translation into sum-of-products form makes the net like any other, so it can be efficiently partitioned and interconnected. The technology mapping stage also does a design rule check to flag illegal logic networks, and it eliminates or optimizes unused or constant logic inputs and outputs to minimize the size of the network.

Often designs to be emulated include RAM and ROM memories. These memories can take a very wide variety of forms from design to design. In addition to the number of locations and their bit width, memories have different numbers and types of write enables and output enables. During technology mapping, a memory compiler can automatically generate the FPGA logic block or board-level SRAM primitives required to emulate the particular memory in the source design. It will

also add information about the memory to the design database, for use by the memory visibility tool at run-time.

Once a complete netlist of FPGA-compatible primitives is available, it must be broken into board-level and chip-level partitions by the system-level partitioner. The partitioner's job is to map primitives into FPGA chips and into board-level collections of FPGA chips, optimizing according to size, pin count, and timing constraints. The number of chips and boards must be minimized while observing the logic capacity and number of I/O pins available in each FPGA and the I/O pins available in each board. Better partitioners will also seek to minimize the number of interchip I/O pin cuts imposed on time-critical design nets. The partitioner may also have a role in timing correctness management (for details see the section entitled "Advanced Topics").

Multilevel multiway partitioning of millions of primitives into thousands of partitions is well known to be a very difficult computing problem (10), for which there is no known technique to directly arrive at the optimum result in polynomial time (i.e., it is an NP-hard problem). Emulation compilers use a combination of the heuristic techniques developed in the academic and industrial communities over the years, which arrive at acceptably near-optimal solutions in a reasonable time. Min-cut (11) and ratio-cut (12) techniques work from the top down, cutting the whole network into smaller and smaller partitions. Clustering techniques work from the bottom up, building partitions out of tightly interconnected primitives. Either or both approaches are generally used, alternately and in sequence. Simulated annealing optimization is often done at the end to improve the results.

Once the design is partitioned, the partition must be placed onto specific FPGAs and boards. The difficulty of this step depends completely on the interconnect architecture. Since the partial crossbar interconnect is completely symmetrical, any placement of partitions into FPGAs is equally valid, so no placement step is needed. When a nearest-neighbor architecture is used, placement is critical to maintaining any hope of accomplishing the routing task without needing too many FPGA pins for long-distance inter-FPGA routing. The placement program must be very sophisticated and powerful, demanding a substantial amount of run-time.

System-level interconnect routing is the final system-level compiler step. The partial crossbar interconnect router works by ordering the nets according to difficulty, mainly fanout, and then assigning them one by one to subsets, specific FPIDs, and specific I/O pins, keeping track in a table. Once most of the nets have been routed, there may be routing failures, where each of the source and destination FPGAs have I/O pins still available, but they are not all in the same subset. This can be cured by ripping up previously routed nets and rerouting. In extreme cases, a maze router completes the routing by taking multiple-stage paths through both FPIDs and FPGAs to complete the final routes. In the nearest-neighbor interconnect, the routing problem is more like that found in a gate array or printed circuit-board router, and similar maze-routing techniques are used. If the router fails to find routes for all nets, the emulation compiler will go back to the placement stage (if a nearest-neighbor interconnect is used) or back to the partitioning stage, to modify the placement and/or partitioning to improve its routability, and routing is rerun.

Once each FPGA and FPID has its logic content, interconnect, and I/O pins fully defined, then each chip is ready for chip-level compilation. The FPGA vendor's placement, routing, and bit generation software is generally used, bound into the compiler such that it is not separately visible to the emulation user. FPIDs are easily compiled, since they are usually built with a single full crossbar. Since each chip-level compile job is independent, they may be done in parallel. When the emulation user has a number of workstations available across the LAN, some emulation software can farm the jobs out onto the network for parallel execution. If any FPGA compile jobs fail to complete, again the compiler must go back and incrementally repartition, replace, and reroute the design, to place less demand on that particular FPGA. Finally the binary programming files for each chip are stored in the emulation database, and the design is ready to be emulated.

Run-Time Software

When the emulated design is ready to be downloaded into the hardware and operated, a number of programs running on users' workstations across the LAN can be run to program and run the emulation and instrumentation. A controller program will direct the emulator's control computer what files to download into the FPGAs and FPIDs to program them with the desired design. It will also define and control any programmable clock inputs that may be used. For pure in-circuit emulation, this is all that is needed.

To run the logic analyzer, a graphical logic analyzer front-end program is used. It can control which of the predefined observable internal and external signals are to be captured, and it can set the trigger conditions for starting the capture. Once triggered, interactive graphical waveform or tabular displays may be used to observe the signals and save them to output vector files. Likewise, if the emulation is driven by vectors from the pattern generator, input vector files can be selected and loaded with this user interface and then displayed in parallel with the captured logic analyzer vectors. Some emulators include the facility to automatically compare captured output vector files with predefined reference vector files and flag any differences. This is useful for validating the emulation against previous simulation, as well as for running regression tests to revalidate after changing the design.

FPGA-BASED LOGIC EMULATOR EXAMPLE

A representative FPGA-based logic emulator is the System Realizer, of Quickturn Design Systems. The System Realizer can emulate up to 250,000 gates in its benchtop form, or up to 3,000,000 gates in the full-size system. Emulation speeds up to 8 MHz are typical. Its Quest II compiler and run-time software can accept designs in structural Verilog, EDIF, TDL, NDL, or any of over 50 ASIC libraries. With the HDL–ICE version, it can accept designs in synthesizable register-transfer-level Verilog or VHDL. It maintains the HDL view of the design throughout the compilation and run-time process.

System Realizer hardware (Fig. 10), introduced in 1995, is based on the Xilinx XC4013 FPGA and a full-custom 168-pin FPID, with a two-level partial crossbar interconnect. Each FPGA has 1152 four-input LUTs, which can also be used as 16-bit RAMs, 576 three-input LUTs, and 1152 flip-flops. The 250,000 gate logic module is a pair of boards, each with a

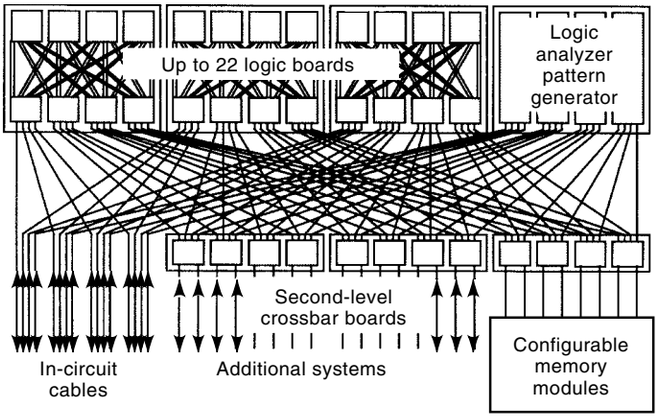

Figure 10. Block diagram of the Quickturn System Realizer logic emulation system hardware.

single-level partial crossbar interconnect of FPGAs and FPIDs. About 1200 of the FPID external I/O pins on each board are connected together, making a directly connected board pair. Of the remaining I/Os, 900 are available for connection to in-circuit cables, the logic analyzer and/or pattern generator, and 2500 go to the backplane.

In the full-size version, up to 22 logic boards are interconnected by a second-level partial crossbar, made up of additional FPIDs on boards on the other side of the backplane from the logic modules, and crosswise to them, to facilitate the partial crossbar wiring pattern. This system interconnects nearly 1000 large FPGAs, with no more than three FPIDs between any two pins. Fourteen thousand external I/Os are available for interconnection of multiple systems for even larger capacity.

Built-in logic analyzer and pattern generator facilities can connect to over 2000 design signals, and generate and capture vectors to a depth of 128,000 vectors, at up to 16 MHz. Complex trigger conditions with up to eight sequential events may be defined. Another form of visibility uses the FPGAs' internal readback scan chains to allow observation and recording of all signals in the design, at a slow or single-step emulation clock rate. Small memories are emulated by the LUT RAMs in the FPGAs. Larger ones are emulated by configurable memory modules (which can hold up to 14 Mbytes), and they emulate memories with as many as four write ports and 16 read ports. All memories may be initialized, read, and written during emulation.

The Quest II compiler automatically compiles multimillion gate designs into this hardware in a single pass. It does timing analysis on the clock systems in the emulated design to guarantee correct-by-construction timing (see section entitled "Timing Correctness"). Quest II can compile designs at a rate of 100,000 gates per hour. Its incremental capabilities can change an internal logic analyzer connection in a few minutes and can recompile a 5000 gate design change in less than an hour.

PROCESSOR-BASED LOGIC EMULATORS

Algorithm

If a simulation algorithm is sufficiently simplified and executed, not by a conventional microprocessor but by application-specific, highly parallel hardware processors, it is possible for the simulator to run fast enough to be used in-circuit like an FPGA-based logic emulator. Inputs are continuously converted into input data for the processors, and processor output data are continuously converted into outputs. Processor-based emulators generally have large capacities and fast compile times, but they are much slower than FPGA-based emulators, and they are not as effective at emulating designs with complex clocking.

Logic simulation executed by the processor-based emulator is reduced to simulating only two logic states, zero and one, and only complete clock cycles, not nanoseconds of gate delay. A levelized compiled code simulation algorithm is used (13). The algorithm works by simulating one clock cycle at a time. Starting with stored values of the clocked register outputs and external inputs, the logic gates are all evaluated, level by level, down to the combinational logic network's outputs, which are the clocked register inputs and external outputs. These values are loaded into register and external output storage, and the process repeats for the next cycle.

To ensure correct evaluation, the logic networks are levelized according to their position in the signal flow. Gates are assigned to levels, such that all inputs to a gate in any given level have been evaluated in previous levels (see Fig. 11). In this example, gates a and b are driven only by register outputs and external inputs, so they may be evaluated first. Gates d and e drive register inputs, so they must be evaluated last. Gate c must be evaluated before d and e, but after a. Three levels are needed, and the gates are assigned to them as shown.

The compiler analyzes the logic network for such dependencies to assign each gate to a level and find the minimum number of levels needed. It also seeks to balance the number of gates in each level, to minimize the amount of hardware needed to execute the simulation. Usually there is a choice of levels to which a gate may legally be assigned. In the example, gate b could be in either level 1 or level 2, since its inputs are all primary, and its output is not needed until level 3.

Hardware Architecture

Processor-based emulation hardware consists of a very large number of very simple processors, each of which can evaluate one gate, enough storage bits to hold the register contents and external I/O values, and a communications network that allows logic values to pass from one gate to the next. If

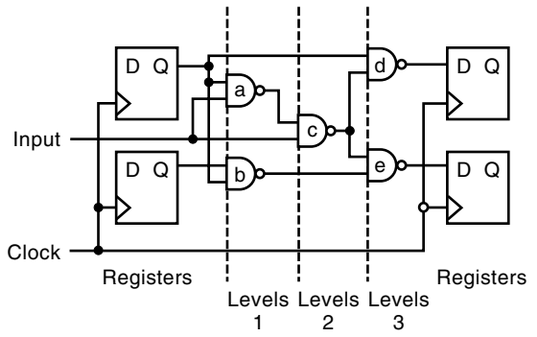

Figure 11. Levelization of logic gates for evaluation by the processor-based logic emulator.

enough gate processors are available to evaluate all the gates in a level at once, then that level can be simulated in one hardware clock cycle. Thus, the three-level example can be simulated in three hardware clock cycles for every emulated clock cycle in the design. If there are more gates in a level than processors available, the level can be split into two levels. This way more emulation logic capacity is available in exchange for emulation speed.

All the hardware is controlled by a single, extremely wide instruction word with a small field for each gate processor, each storage bit, and each stage in the communications network. Since the simulation algorithm evaluates all the gates the same way every cycle, there are no data dependencies in the program and no conditional branches in the instruction set. The simulation executes as a single short loop, with one instruction per logic level and one iteration per emulated clock cycle.

Since all the gates are evaluated only once per emulated clock cycle in a forward sequence, emulated designs may not contain internal feedback loops, since they won't be evaluated correctly. Because a loop of processor instructions corresponds directly to an emulated clock cycle, there can only be one emulated clock signal or, at most, a set of clocks that are all locked to the same master clock. Designs with multiple unrelated clocks cannot be emulated with a single processor-based emulator.

Software Architecture

Software for compiling designs into a processor-based emulator is similar, even identical, to that of the FPGA-based emulator, down through the technology mapping, which targets the gate-level processor primitives rather than the FPGA's. Then the logic network is levelized and scheduled onto specific processors and logic levels. The communications network is scheduled to make sure the proper signals are available to the proper processors at the right times. If conflicts occur in communication requirements between signals in the same level, signals are held in intermediate storage, gates are moved to other levels, and/or additional levels are introduced, to achieve successful communication of all signals between their processors and levels. Resolving this for a large design can be a challenging and time-consuming compiler task. The resulting instructions may then be loaded into the hardware for execution. Since this process is still simpler and faster than multi-FPGA partitioning, routing, and all the FPGA-level place and route jobs, the execution time of the compiler can be much shorter for the processor-type emulator.

Example

An example of the processor-based emulator is the concurrent broadcast array logic topology (CoBALT) system, of Quickturn Design Systems. The capacity of a single CoBALT system is between 500,000 and 8,000,000 gates. Typical emulation clock rates are between 250 kHz and 1 MHz. Each CoBALT board includes 2 Mbytes of on-chip memory and up to 8 Mbytes of additional memory cards, for emulating design memories. CoBALT can be operated in-circuit, can be vector-driven, or can be operated in co-simulation with another simulator. Its logic analyzer and pattern generator system has up to 2048 channels per board, each with a depth of up to 512,000 vectors. CoBALT's software is completely integrated with the same Quest II compiler front-end, including VHDL and Verilog emulation, and the same run-time debug tools as the FPGA-based System Realizer. It can compile a one-million-gate design in less than one hour on a single workstation.

CoBALT hardware is based on a 0.25 μm full custom chip, which has 64 logic processors, each of which can evaluate any three input logic gate. Each board has 65 processor chips. Each chip has direct connections to all 64 other chips, for rapid communications between the processors in different chips. The hardware clock rate, and thus the instruction rate, is 100 MHz.

ADVANCED TOPICS

HDL Emulation

Lately some logic emulators have developed the ability to handle designs in register-transfer-level (RTL) hardware description language (HDL) form, instead of being restricted to designs represented at the structural gate level. This reflects the increasing practice of doing RTL designs and using reliable synthesis tools that translate the RTL down to the gate level. HDL logic emulators accept the same synthesizable subsets of VHDL and/or Verilog that the silicon-targeting synthesis tools accept. They internally synthesize the HDL into a form that is optimized for emulation. At run-time, the emulator's debugging tools operate at the HDL level as well, allowing the user to identify signals and modules with the same names used in the source HDL code.

Synthesis. Logic synthesis in a logic emulation compiler has a different set of requirements than a synthesis tool that is targeted to silicon. Rather than synthesizing to gates, it synthesizes directly into the FPGA's logic primitives, such as the LUT, for which the cost depends only on the number of LUTs, not their logic functions. The emulation HDL compiler is set for rapid execution time, rather than taking a long time to get the smallest possible logic size or the fastest possible logic delay. For example, using the Quickturn HDL-ICE system, on design modules in the 30,000 to 80,000 gate range, synthesis for emulation takes one half hour or less, compared with six to twelve hours for the silicon-targeting synthesis tool. Design change iterations take much less time as a result.

Operation. The compiler saves the HDL source code and source variable and module names in the emulation database, and it keeps track of the mapping between source code elements and their emulated form. During operation, the user interface to the logic analyzer and pattern generator displays the HDL source code files and source module structure. Signals are selected and identified by their source code names. This greatly simplifies the debugging task compared with dealing with the postsynthesis gate-level version of the design.

Timing Correctness

Since an emulated design is translated from its silicon-targeted form into FPGAs and FPIDs, the logical function can be maintained to be identical, but the internal delays must be different. In particular, the proportion of delay between logic and interconnect is fundamentally different. In permanent

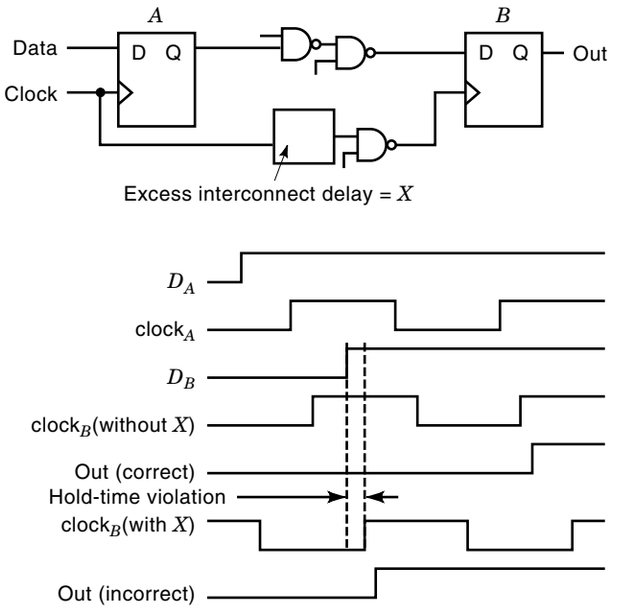

Figure 12. Example of a hold time violation in an emulated design, introduced by excessive delay in a gated clock path. The timing diagram illustrates the correct operation, without excess delay X, in the first five waveforms. The lower two waveforms show the clock delayed by X, the hold-time violation between input and clock, and the incorrect output that results.

silicon, the logic delays usually are longer than the wire delays, although this balance is shifting as processes shrink far below 1 μm feature size. In the emulator, interconnect delays in the FPGAs and FPIDs dominate, since signal paths pass through many programmable switches between logic blocks.

Built-in low-skew clock distribution networks in the FPGAs and the emulation system hardware ensure that clock delays are minimal and uniform, so the internal delays mainly affect only the data paths. In fully synchronous designs with a single clock, the only problems excess data path delays can cause are setup time violations on flip-flop inputs, when the clock arrives early, before the data are ready. These are easily cured by slowing the clock frequency.

However, many designs have logic in the clock paths (i.e., gated clocks), asynchronous feedback, multiple unrelated clock domains, and other deviations from pure synchronous timing. Delay differences can introduce hold-time violations, when the clock in one stage arrives late due to logic delays, after the previous stage has been clocked and has already changed that stage's input data (see Fig. 12). Flip-flop B is clocked by a gated clock. The path from the clock at A to the clock at B is designed to be faster, in the real implementation, than the data path from A to B. But suppose that in emulation the clock path to B is cut by the partitioner, and a substantial delay X is introduced, which makes the clock path too slow. When the clock edge occurs at A, the resulting change in the data input arrives at B before the same clock edge has arrived at B, resulting in error. Clock frequency adjustments cannot cure hold-time violations, since they are entirely due to imbalance between the internal clock and data paths.

More sophisticated emulation compilers (14) can conduct clock tree and timing analysis to avoid or even correct such delay imbalances and to determine a safe clock frequency to ensure correct emulation. Given information about which design nets are primary clock inputs, clock enables, and so on, each clock tree—that is, each tree of logic that feeds into a clock input—is automatically identified and analyzed. If the possibility of a clock path delay exceeding a data path delay is identified, then additional delay elements are programmed into the FPGAs and/or FPIDs in the data path. This will correct the imbalance and avoid hold-time violations.

Since interconnect delays are introduced when logic is split between FPGAs, clock tree logic that drives many clocks can be duplicated in each FPGA where the clocks appear, so the clock tree need not suffer inter-FPGA delays. Some emulators provide a special FPGA for clock tree logic, with low-skew clock distribution paths from the clock tree FPGA to the other FPGAs, again to avoid unnecessary delay in the clock paths. The partitioner may also be called upon to manage clock tree networks, maintaining clock tree logic uncut in the same chip with clock logic and duplicating clock tree logic when necessary.

OTHER USES OF THE TERM

The term *emulation* has become primarily associated with logic emulation, but the term is also used in a number of other senses in the computing field. An in-circuit emulator (ICE) is a debugging tool, which replaces a microprocessor chip with a plug and cable to a benchtop device or PC. It is usually used to debug an embedded control application where a screen and keyboard are not otherwise available. The ICE allows debugging and monitoring of the microprocessor's software execution. In-circuit emulators are passing out of common usage, because microprocessors now provide built-in debugging facilities which connect to a PC, and benchtop logic analyzers can interpret processor instructions, bus states, and data.

An instruction set emulator is software, which executes instructions of a different instruction set than the one that is native to the hardware doing the execution, to run programs written for a different processor. An example would be an instruction set emulator that runs on the processor of a Macintosh computer and emulates Intel x86 instructions, to allow software written for the PC to run on the Macintosh.

BIBLIOGRAPHY

1. J. Gateley, Logic emulation aids design process, *ASIC & EDA,* July: 1994.
2. S. Trimberger (ed.), *Field-Programmable Gate Array Technology,* Boston: Kluwer, 1994.
3. S. Brown et al., *Field-Programmable Gate Arrays,* Boston: Kluwer, 1992.
4. S. Sample, M. D'Amour, and T. Payne, *Apparatus for emulation of electronic hardware system,* US patent 5,109,353, 1992.
5. S. Walters, Computer-aided prototyping for ASIC-based systems, *IEEE Design & Test,* **8** (1): 4–10, 1991.
6. A. Wolfe, Intel's Pentium parry, *Electron. Eng. Times,* December **5**: 1, 1994.

7. E. Rymaszewski and R. Tummala, Microelectronics Packaging—An overview, in R. Tummala and E. Rymaszewski (eds.), *Microelectronics Packaging Handbook,* New York: Van Nostrand Reinhold, 1989, p. 13.
8. M. Butts, J. Batcheller, and J. Varghese, An efficient logic emulation system, *Proc. IEEE Conf. Comput. Design,* October 1992, p. 138.
9. M. Butts and J. Batcheller, *Method of using electronically reconfigurable logic circuits,* US Patent 5,036,473, 1991.
10. N.-C. Chou et al., Circuit partitioning for huge logic emulation systems, *Proc. 31st Des. Autom. Conf.,* June 1994, p. 244.
11. C. Fiduccia and R. Mattheyses, A linear time heuristic for improving network partitions, *Proc. 19th Des. Autom. Conf.,* 1982, pp. 175–181.
12. Y.-C. Wei and C.-K. Cheng, Ratio cut partitioning for hierarchical designs, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.,* **10**: 911–921, 1991.
13. M. Denneau, The Yorktown simulation engine, *Proc. 19th Des. Autom. Conf.,* IEEE, 1982, pp. 55–59.
14. W.-J. Dai, L. Galbiati, and D. Bui, Gated-clock optimization in FPGA technology mapping, *Proc. Electron. Des. Autom. Test Conf.,* Asia, 1994.

MICHAEL BUTTS
Quickturn Design Systems, Inc.

EMULATORS. See RAPID PROTOTYPING SYSTEMS.