Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

ELIXENT EXPANDS SOCS

Licensable Array Processor Has Massively Parallel Architecture

By Tom R. Halfhill {7/21/03-01}

Seeking a soft spot between a rock and a hard place, U.K.-based Elixent is introducing a massively parallel processor core that strives to combine the programmability of a general-purpose processor with the performance of a hard-wired ASIC. The goal: a more flexible system-on-chip (SoC) processor that consumes less power and adapts quickly to different tasks, amortizing the development costs of an SoC over multiple projects.

Elixent, a three-year-old spinoff from Hewlett-Packard Labs, described its new D-Fabrix architecture at **Embedded Processor Forum 2003**. The D-Fabrix is based on a concept that Elixent calls reconfigurable array processing (RAP). It's similar to the reconfigurable compute fabric (RCF) in the MRC6011 processor that Motorola also presented at **Embedded Processor Forum**. (See *MPR 7/14/03-01*, "Motorola Attacks ASICs.") A key difference is implementation: Elixent licenses the D-Fabrix as a hard macro for SoC integration, whereas Motorola offers the MRC6011 as a standard part.

Both Elixent and Motorola use the term "reconfigurable" rather loosely. Neither processor has reprogrammable gate-level logic, like that found in FPGAs. The Elixent core is structurally configurable by SoC developers at design time, but the logic remains static at run time. The Motorola processor, available only as an off-the-shelf chip, isn't structurally configurable by customers at all. Instead, both processors have arrays of small function units with local registers and tightly coupled memories, all woven together by a fabric of datapaths. The function units are independently programmable at run time and execute their tasks locally, without fetching instructions from off-chip or distant on-chip memory over a bus. This frees the fabric's I/O bandwidth for data throughput.

Another feature common to both the Elixent and Motorola architectures is an integrated RISC microprocessor core, which acts as a central controller for the processing array. In addition to supervising the flow of data, the RISC controller can load new programs to reprogram the array processors for different tasks or for different parts of a single task. In most cases, this can happen fast enough to make changes on the fly.

MPR prefers to call these processors reprogrammable or run-time-programmable rather than reconfigurable, because their logic is static at run time. (See the sidebar, "Defining Reconfigurable Processing" in *MPR 7/14/03-01*, "Motorola Attacks ASICs.")

Nevertheless, the Elixent and Motorola processors offer a different model of execution than do general-purpose microprocessors and programmable DSPs. Conventional processors execute a common stream of instructions fetched over a bus from on- or off-chip memory that is global to all the function units. In addition, function units of the same data type share a global register file. The Elixent and Motorola processors can simultaneously execute independent streams of instructions fetched from on-chip memory that's local to each function unit, and each unit has its own local register file.

A Checkerboard of ALUs

At the heart of Elixent's D-Fabrix architecture is a scalable array of muxes and four-bit ALUs with local registers and

closely coupled memories, all connected together by a fabric of datapaths. There are no other types of function units—no adders, shifters, or multipliers. The ALUs perform those arithmetic functions, singly or in concert. To carry out 32-bit integer operations, for example, an application could reprogram the array to combine eight of the four-bit ALUs into a pipelined 32-bit integer unit.

The basic D-Fabrix building block is called a tile, which contains two four-bit ALUs and six local registers. Elixent says there are no architectural or electrical limits to the number of tiles a designer can integrate in an SoC; the practical limits are power consumption, cost, and die size. A typical SoC will have 128 to 2,048 tiles, although much larger devices are possible. As a rough guide, a JPEG encoder capable of compressing 100 million pixels per second could be implemented with 256 tiles in a chip that runs at 100MHz. Dedicating more tiles to a task that can execute in parallel will increase performance at a near-linear rate.

Each ALU in a tile has about 100 ASIC-equivalent gates. However, gate-count comparisons with ASICs are not very useful. Because the ALUs are programmable, they can perform multiple tasks that would require many more gates to implement in a fixed-function ASIC.

On the other hand, a D-Fabrix SoC probably won't run as fast as an ASIC. Due to the architecture's complexity, the worst-case clock frequency is only about 100MHz at 0.18 micron and 150MHz at 0.13 micron. Elixent has characterized the hard macro for some common 0.18- and 0.13-micron CMOS processes at leading foundries, and Toshiba is producing a D-Fabrix processor (the ET1) at 0.13 micron. Porting the core among different 0.13- or 0.18-micron processes is relatively straightforward, but moving it to a 90-nanometer process would require a re-layout, as with any hard macro. The core is a fully static design, so lower clock frequencies are feasible if power consumption becomes a more important issue than performance.

Elixent provides the core as a GDS-II macro with register-transfer-level (RTL) Verilog and VHDL models. It's more flexible than a conventional hard macro, which usually has a fixed layout that customers cannot change. The D-Fabrix core is perhaps better described as a firm macro, because designers can scale the number of tiles in the core with a tool that Elixent calls the Array Generator. The Array Generator is like a sophisticated memory compiler, because it allows designers to combine several basic elements of a structure (in this case, processor arrays instead of memory arrays) to build a larger structure.

As Figure 1 shows, the physical layout of the D-Fabrix resembles a checkerboard, with each ALU occupying a square surrounded by the switchboxes of datapaths connecting it to surrounding ALUs. Each ALU has three four-bit registers and 36 four-bit buses (18 buses in each perpendicular direction) connected to the switchboxes. The smallest possible array is a 2×2 matrix of tiles, which contains eight ALUs and eight switchboxes. In practice, a pair of these arrays is tightly coupled to 256 bytes of locally shared SRAM over an eight-bit-wide data bus. This yields a basic array unit, or repeating block, of two arrays plus local memory, with a total of 16 ALUs. Using the Array Generator tool, SoC designers can combine as many repeating blocks as necessary for the application.

The on-chip controller for the D-Fabrix network is Toshiba's Media Embedded Processor (MeP), a synthesizable 32-bit RISC core with a VLIW coprocessor. (See *MPR 6/10/02-02*, "New Processors for New Media.") The controller sits outside the fabric but is tightly coupled to it by a 32-bit interface. Around the periphery of the fabric is more local memory, arranged in dual banks. SoC designers can

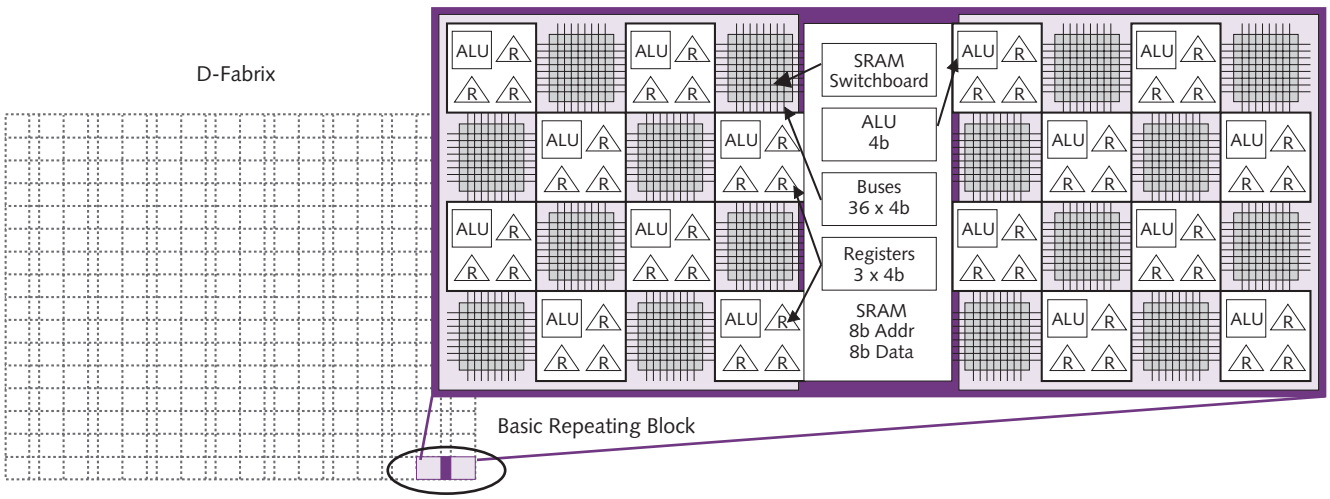

Figure 1. A basic repeating block in the D-Fabrix architecture has a pair of tile arrays surrounding 256 bytes of SRAM, with each array containing eight ALUs and eight interconnecting switchboxes. Any ALU in an array block can access the shared local memory or can exchange data with other ALUs in a single clock cycle.

implement different amounts of memory to suit their application (the Toshiba ET1 has 108KB). An application can use this memory as an I/O buffer, as temporary local data storage, or to store program code for the arrays. As Figure 2 shows, an internal bus connects all these elements—the D-Fabrix, the RISC controller, and the dual-banked memories—to a DMA controller and memory interface.

Using the dual-banked memories for I/O buffers or temporary data storage is the same as using local memory for those purposes on other SoCs. Using the memories as what Elixent calls a "reconfiguration buffer" is a distinguishing feature of the D-Fabrix. In theory, reprogramming the arrays might never be necessary if an SoC designer made the fabric large enough to execute all the tasks required for the application without reprogramming. However, that would inflate the fabric's size and negate the main advantage of using a programmable processor in the first place.

To make the most of the architecture's capabilities, designers should scale the fabric to meet the peak *performance* requirement of their application (and leave some headroom for the future), not necessarily the maximum *size* requirement. Ideally, the fabric should be large enough to perform the application's most demanding data-processing task in the required amount of time, but should also be too small to handle the whole application without reprogramming itself at run time. Such a configuration will minimize the number of redundant ALUs in the fabric.

Reprogramming the arrays at run time allows a downsized processor to tackle larger jobs by dividing the work into multiple phases—as long as the time required to reprogram the processor for each phase doesn't prevent it from meeting the application's performance target. It's the same principle as hiring temporary contract labor for different phases of a design project instead of employing a team of full-time engineers large enough to carry out the entire project. The more-flexible design team can expand, contract, and adapt its breadth of expertise to match changing demands.

Juggling the Trade-Offs of Reprogramming

Meeting an application's peak performance target while keeping the fabric as small as possible requires careful analysis by SoC designers. First, they have to divide the application into tasks the processor can perform in parallel and those it can perform sequentially. Parallel tasks require a larger fabric with more arrays of ALUs; sequential tasks might be able to reuse the same ALUs by reprogramming them between steps. Next, the designers must determine whether the processor can reprogram itself quickly enough at run time to carry out some sequential tasks in the same ALUs without missing the application's performance target.

The reprogramming speed varies according to where the application stores the configuration code. For the fastest access, code can reside in the dual-banked SRAMs surrounding the fabric, visible in Figure 2. Accessing this memory is almost as fast as accessing the local memory in each block, because it doesn't consume any bus bandwidth; all transactions flow through the fabric. Alternatively, the configuration code can be stored in off-chip system DRAM. Accessing this memory requires a bus transaction, so it's the slowest place to store the code, but it has the largest capacity—perhaps megabytes.

Deciding where to store the configuration code depends on the SoC designer's objectives, and all the code need not reside in the same place. One goal might be to design an SoC that, unlike a hard-wired ASIC, is suitable for several different applications or a line of related products; in that case, programmability at run time isn't a factor. Storing all the configuration code in off-chip memory and loading it during boot-up would be fast enough. Another goal might be to replace multiple ASICs in a system with a single SoC that the application reprograms at run time to handle different tasks; in that case, reprogramming must be fast enough to keep up with user input or data I/O. At least some of the configuration code would have to remain in the dual-banked memories tightly coupled to the fabric.

Obviously, the answer depends greatly on the application. Elixent says a D-Fabrix can be reprogrammed fast enough to handle the number-crunching tasks typically encountered in digital cameras (JPEG compression), digital video products (MPEG encoding/decoding), and cellular telephony (3G and GSM baseband processing). In general, these tasks might require the D-Fabrix to reprogram its

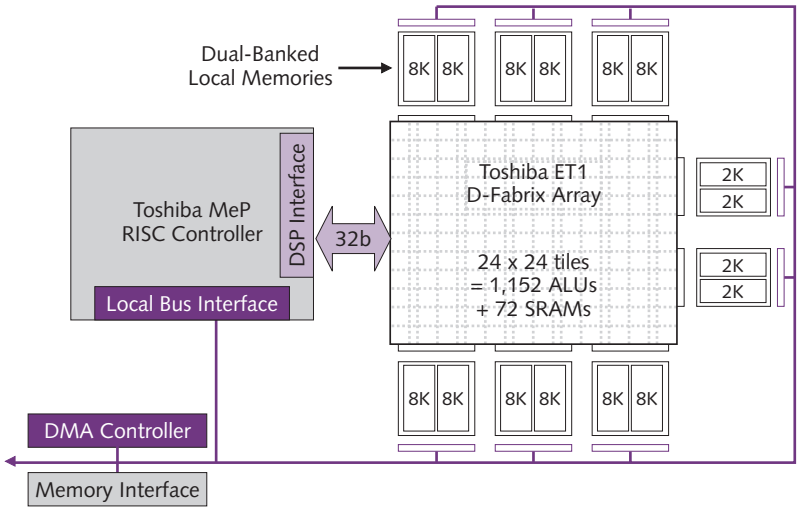

Figure 2. This higher-level block diagram of Toshiba's ET1 implementation shows all the processing and local-memory components of the D-Fabrix architecture. SoC developers can license this architecture as a hard macro, scale the number of arrays in the fabric to fit their applications, and integrate any additional components they wish, such as more on-chip memory and application-specific peripherals.

arrays in time frames ranging from 40 milliseconds (4 million clock cycles at 100MHz) to 10 microseconds (1,000 clock cycles at 100MHz)—well within the processor's capabilities.

Another trade-off designers have to make is determining how much tightly coupled memory to use for configuration code and how much to reserve for data. Using more memory for code speeds up reprogramming at the expense of data throughput, and vice versa. In one implementation of an MPEG encoder, Elixent found that the amount of execution time the fabric spent reprogramming itself varied from about 1% to 55%, depending on where the code was stored. Elixent determined that the best balance was to share the tightly coupled configuration buffers between the code and the data in proportions that reduced the reprogramming overhead to 5% of the execution time.

ROSS MEHAN

Alan Marshall, CTO of Elixent and a 15-year veteran of HP Labs, describes the D-Fabrix architecture at EPF2003.

Evaluating Performance

Elixent has no formal benchmark results for a D-Fabrix processor. So far, the only silicon implementations are test chips; Toshiba expects to receive the first silicon for its ET1 soon. Elixent says the test chips are fully functional, but neither those chips nor the RTL models have been subjected to EEMBC or any other industry-standard benchmark testing.

One problem is that EEMBC requires vendors to run the "out-of-the-box" (standard) benchmarks without modifying the C source code, which a D-Fabrix processor cannot do; the array processors require special programming. However, Elixent could run the EEMBC benchmarks under the "full-fury" or optimized rules, which allow the vendor to modify the benchmark code. Elixent is negotiating this issue with EEMBC. (See *MPR 6/21/99-01*, "Embedded Benchmarks Grow Up.")

Elixent has conducted some informal tests with the types of algorithms that customers are likely to run on a D-Fabrix processor. In one example, Elixent compared the simulated performance of a 100MHz D-Fabrix processor with that of a 100MHz DSP when both were running a common algorithm for a digital camera. The algorithm performs RGB color interpolation—estimating the RGB color of an image pixel by sampling the colors of nearby pixels. (Color interpolation is necessary because conventional CCD and CMOS image sensors record only one of the three RGB color components per pixel; only the Foveon X3 is a true three-component RGB sensor.)

To interpolate the two missing color components for a pixel, the DSP must execute 20 instructions: eight loads, eight adds, two shifts, and two stores. For this comparison, Elixent configured a D-Fabrix with 1,024 ALUs, although it needs only 24 ALUs to perform two color interpolations in parallel. (The remaining ALUs would be available for other tasks.) Elixent also added the latency of configuring all 1,024 ALUs at run time—assuming they weren't already programmed—but it was an insignificant amount of the total execution time (0.05%). Table 1 shows the results of this comparison.

Using only 24 of its 1,024 ALUs, or 2.4% of the fabric's capacity, this hypothetical D-Fabrix processor easily outruns a DSP that must dedicate all its resources to the same task. Of course, Elixent contrived this example to demonstrate the architecture's superiority for algorithms with a great deal of exploitable parallelism. And in practice, a properly programmed DSP could hide some of the load latencies and achieve other efficiencies. Still, the general point is that the D-Fabrix architecture is very good for data-intensive code that can execute in parallel.

Conversely, an SoC designer could implement a smaller D-Fabrix configuration that delivers the same performance as a DSP while consuming less power. To illustrate this, Elixent compares JPEG still-image compression on the D-Fabrix with a Texas Instruments TMS320C54x DSP. According to TI's data, the 'C54x can compress 2.58 megapixels per second while consuming 90mW. At the same power level, a D-Fabrix processor could compress the image almost 20 times faster (51 megapixels/sec). At the same performance level, the D-Fabrix could compress the image while consuming only 1/20th as much power (4.55mW).

Task (2 Megapixels RGB)	D-Fabrix Execute Time	DSP Execute Time
Array Programming	0.01ms	—
Interpolate R Color	20ms	400ms
Interpolate G Color	20ms	400ms
TOTAL	20ms	800ms

Table 1. The D-Fabrix processor could interpolate the missing red and green color components for two million RGB pixels in parallel, so the total elapsed time is the same as for interpolating each color component separately. It could finish the job 40 times faster than a DSP running at the same clock frequency (100MHz).

The Catch: Software Development

The biggest advantages of the D-Fabrix accrue from its massive parallelism and run-time programmability (Elixent's "reconfigurability"). Consider the previous example of using a D-Fabrix processor instead of a fixed-function ASIC in a digital camera. Simply by reprogramming the array, the camera manufacturer could adapt the chip for the wavelets of JPEG 2000 instead of the discrete cosine transforms (DCT) of regular JPEG encoding. That might allow the manufacturer to reuse the same SoC in multiple cameras, saving the considerable cost of another ASIC project.

Of course, general-purpose processors are reprogrammable, too. The camera manufacturer could just as easily replace the ASIC with an SoC built around a RISC core licensed from ARC, ARM, MIPS, Sun, Tensilica, or another vendor. It needn't take any longer or cost any more money for the manufacturer to spin the silicon with a RISC core than with Elixent's core, and the RISC-based SoC would be easier to program. General-purpose RISC architectures are well understood and have lots of mature software-development tools.

In contrast, the D-Fabrix isn't programmable with an everyday C compiler or assembler. To write the configuration code, programmers must use the hardware-description languages Verilog or VHDL, or a special version of C adapted for hardware design (Celoxica's Handel-C), or a special data-visualization and algorithm-development tool (MathWorks' Matlab). Not that those languages are inferior to plain-vanilla C; they're just different.

If critical parts of the application code already exist in Verilog or VHDL—possibly because an earlier implementation of the product used ASICs—the transition will be easier. Otherwise, before software development can begin, programmers must work closely with the SoC designers to determine how to partition the application into tasks that can execute in parallel or sequentially; estimate how large the fabric should be; decide how to distribute the application among the array processors; and allocate the closely coupled memories between configuration code and application data. Some of this analysis is unique to array architectures like the D-Fabrix, so it requires uncommon expertise. It also requires a degree of cooperation between the hardware and software engineers that will be new to some development teams.

To put it bluntly, expect a steeper learning curve for both the hardware and software flows of a project when adopting an unconventional architecture like the D-Fabrix. The potential payoff is higher performance and lower power consumption than would be possible with a DSP or an SoC built around a general-purpose RISC core. This is almost certainly achievable in data-intensive applications, which lend themselves to the nearly unlimited parallelism of a scalable array processor. Indeed, the D-Fabrix is capable of more parallelism than most applications can expose. This is not true of the general-purpose RISC cores available for SoC integration. Almost all licensable RISC cores are simple uniscalar-pipeline designs, although several of them have SIMD, VLIW, or DSP extensions that can exploit some data parallelism.

At the other end of the spectrum, a custom ASIC should have little trouble outperforming a D-Fabrix or RISC-based SoC if flexibility isn't an issue. Well-designed ASICs can exploit parallelism, too, and their hard-wired functions don't need reprogramming and aren't burdened with instruction fetching. An ASIC should easily be able to outrun a D-Fabrix SoC, whose complex architecture severely limits its maximum clock frequency.

In other words, it's the old programmability-vs.-performance debate that stretches as far back as the 1940s. The D-Fabrix architecture's highly parallel local-execution model, with an array of independently programmed processing units, offers an interesting middle ground.

Price & Availability

The D-Fabrix hard macro is available now for 0.13- or 0.18-micron CMOS processes at popular foundries, including Chartered, TSMC, and UMC. Elixent hasn't disclosed licensing fees. For more information, see *www.elixent.com*.