Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

# AMBRIC'S NEW PARALLEL PROCESSOR

*Globally Asynchronous Architecture Eases Parallel Programming*

*By Tom R. Halfhill {10/10/06-01}*

There are two general approaches to conceiving a new microprocessor architecture. One is to start by designing an elegant architecture, then finish by building the software-development tools. The other is to start with an elegant programming model, then create a microprocessor architecture capable of efficiently executing it.

Ambric, an Oregon-based fabless semiconductor company founded in 2003, favors the latter approach. Starting with a new microprocessor architecture and building the development tools as an afterthought is tolerable if the architecture is conventional. But if the new architecture is an exotic one intended for massive parallelism, the tools require more forethought. As overwhelmed programmers are discovering, it's easier to slap down multiple processor cores on a chip than it is to write efficient parallel-processing code for the damn things.

At this week's **Fall Microprocessor Forum** in San Jose, California, Ambric Fellow and senior hardware architect Mike Butts introduced his company's Am2045 massively parallel processor and architecture. This 117-million-transistor chip, fabricated in a modest 0.13-micron CMOS process, crams 360 proprietary 32-bit RISC processors and 585KB of SRAM onto a single compact die. Maximum theoretical performance exceeds one trillion operations per second (TOPS) at 333MHz. The Am2045 is designed to replace high-end embedded processors, DSPs, and FPGAs in applications that require fast general-purpose integer and digital-signal processing. Examples include H.264 digital-video compression/decompression and data processing for communication infrastructures.

More important than the chip is the programming model. Application programmers will write most of their code in Java, but the Am2045 isn't a Java chip. There's no Java virtual machine, bytecode interpreter, or just-in-time (JIT) compiler. There isn't even any bytecode. Instead, Ambric has adopted Java solely for its familiarity and object orientation. Ambric's special software-development tools convert Java source code into proprietary machine language, then automatically map the compiled objects onto the massively parallel array of RISC processors.

Ambric organizes the RISC processors into independent clusters, each cluster having its own ALUs, registers, and local memories. The processors and clusters communicate with each other by passing messages through special channels and registers. By duplicating the clusters in a cookie-cutter fashion, Ambric can design a chip with virtually any number of processors, within the limits of the fabrication technology. In addition, each processor cluster runs at its own variable clock speed, matching performance to the workload.

Ambric claims it has achieved its goal of making a massively parallel processor that's relatively easy to program. Of course, the truth will be evident when the Am2045 gets into the hands of more developers. In any case, Ambric has designed a fascinating architecture that rides three waves: multicore microprocessor design, intense interest in parallel processing, and object-oriented programming.

## Cache Is Good, Logic Is Better

A fundamental philosophy behind Ambric's design is to take advantage of Moore's law in ways other than dumping more cache on the chip. It's not that Ambric doesn't like cache—it's just that logic is the real workhorse of a microprocessor. Cache is the fallback position if the designers can't find anything more useful on which to spend their transistor budget. It's no longer unusual for processors to devote more die area to memory than to logic. Although cache is a valid way to improve performance and overcome the widening gap between processor performance and memory latency, some architects view it as a detour from the divine path of Moore's law. (See *MPR 12/13/04-02*, "Viewpoint: The Mythology of Moore's Law.")

CPU architects recently hit the walls of clock frequency and heat dissipation, but the march of fabrication technology continues to expand transistor budgets, often beyond the ability of architects and design tools to effectively apply the transistors to logic. Multicore design is the industry's reaction to that dilemma. By replicating processor cores, architects can design larger chips with relatively little additional effort. However, writing multithreaded code that efficiently utilizes a multicore processor is another matter. Recently, there has been an explosion of new interest in parallel processing and multithreaded programming. Small companies and university projects that once labored in obscurity are now competing for their 15 minutes of fame.

Over the years, most programmers have grown comfortable with the relatively extravagant expenditure of two critical computing resources: main memory and mass storage. In all but the smallest embedded systems, both resources are available in relative abundance today. In contrast, the most critical computing resource—processor clock cycles—is more scarce. Most systems have only one main processor. Now, multicore processors are making clock cycles as abundant as RAM and storage capacity have become. But programmers are still conservation minded, struggling to write multithreaded code that distributes complex workloads across multiple processor cores while wasting as few clock cycles as possible. What if processors were a plentiful and fungible resource, like RAM and disk space? That day is nigh, but programming models and tools haven't kept up.

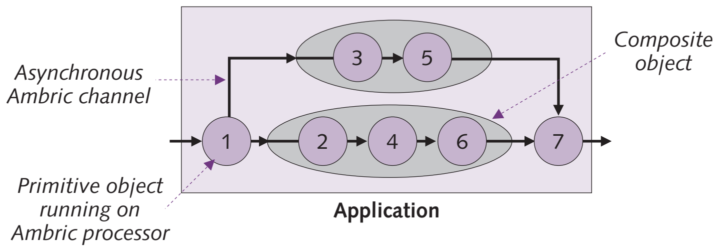
Asynchronous Ambric channel
Composite object
3
5
1
2
4
6
7
Primitive object running on Ambric processor
Application

**Figure 1.** This conceptual diagram shows how software objects (essentially, subroutines or groups of related subroutines) run on multiple processor cores in Ambric's massively parallel array. Simple or "primitive" objects may require only a single processor core. Complex or "composite" objects may run on multiple cores. Processor cores may run asynchronously in relation to other cores. Ambric's message-passing mechanism brings order to this organization.

Ambric's programming model assumes that a modern chip can integrate so many processor cores that a simple software object (a subroutine, more or less) can run exclusively on its very own processor. No other objects in the program need to contend for the same resources. More-complex objects may run on multiple processor cores. Although this programming model may waste some processing resources, it allows logically divided chunks of code to run with a high degree of autonomy and independence.

Locally, each small cluster of processors in Ambric's massively parallel array runs at its own clock speed, as dictated by its software workload. Globally, however, the processors don't march in time to a single clock signal. This organization of independent clock-frequency domains is called GALS (globally asynchronous, locally synchronous), and it's the key to understanding Ambric's architecture. Note, however, that each small cluster of processors in Ambric's chip is built in conventional synchronous logic, not in clockless asynchronous logic. That characteristic sets it apart from the SEAforth parallel-processing architecture that IntellaSys introduced at Spring Processor Forum. SEAforth is locally and globally asynchronous and has an equally innovative programming model based on the Forth language. (See *MPR 8/21/06-03*, "Embedded Arrays Venture Forth.")

## On-Chip Interconnects Regulate Processing

Figure 1 is a high-level conceptual diagram showing how Ambric maps its programming model onto multiple processor cores. For this simple example, we're showing only seven cores. Two of them (numbers 1 and 7) are running relatively simple software objects requiring only a single core. A more complex object, which Ambric calls a composite object, is running on two cores (numbers 3 and 5). An even more complex composite object is running on cores 2, 4, and 6. An on-chip network of channels connects the cores together.

One problem with a multicore architecture that allows individual cores to run at their own clock speed is synchronizing their processing. Obviously, one core can't begin working on data that depends on another core's results until those results are ready. This is the classic data-dependency problem that requires conventional out-of-order processors to devote so much logic overhead to pipeline control. But out-of-order processors exploit instruction parallelism discovered at run time, for the most part. Ambric's parallel processor exploits data parallelism discovered and mapped to the processor array at development time.

In Ambric's architecture, special data channels are wholly responsible for synchronizing the processor cores. There's no global flow control or explicit pipeline scheduling. Instead, the channels regulate the processors locally. A processor located downstream must wait until it receives results from a processor located upstream. The point-to-point channels are a word wide (32 bits, in this implementation) and are very short wires. At each end is a special register that's distinct from the general-purpose

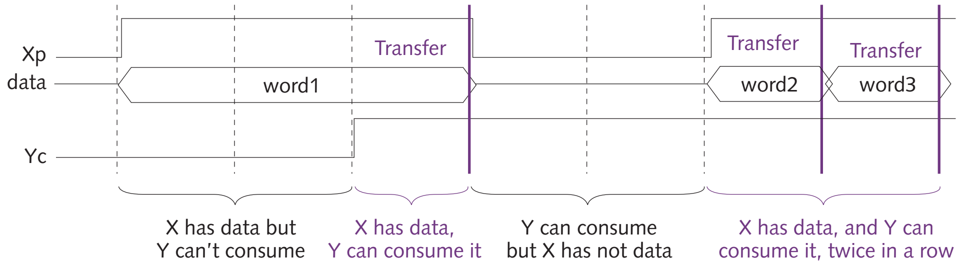
Xp
data
Yc
word1
Transfer
Transfer
word2
Transfer
word3
X has data but Y can't consume
X has data, Y can consume it
Y can consume but X has not data
X has data, and Y can consume it, twice in a row

**Figure 2.** Local channels that connect neighboring processor cores also synchronize the cores in Ambric's massively parallel architecture. There's no global flow control, so each core can run at a different clock speed, as regulated by the channels. In this diagram, two special channel registers (X and Y) are communicating over a unidirectional channel. Downstream signal Xp is high when X has data; upstream signal Yc is high when Y is ready to accept data. The channel transfers data only when one register is ready to provide it and the other register ready to receive it. Either register can stall the channel if necessary.

registers in the processor cores. To the processors, however, the channel registers look like normal registers—read/write operations require only one clock cycle. This arrangement allows very fast message passing between neighboring processors. Figure 2 illustrates the flow-control protocol of these channels.

From a global view, the parallel-processor array runs asynchronously. This allows the array to automatically vary the speeds of its processor clusters according to the tasks they execute. Individual processor clusters can run at widely different clock frequencies—from less than 1.0MHz to 333MHz, in Ambric's initial implementation. Locally, each processor cluster runs synchronously, coordinating with neighboring clusters through the chained-register channels. This GALS design avoids some potential pitfalls of clockless asynchronous architectures, such as race conditions and the difficulty of maintaining stability throughout the device.

Ambric says the architecture is an "asynchronous machine, but really it's clocked"—thus revealing that the company's name can be interpreted as a subtle acronym. Ambric also uses the "bric" portion of the acronym to describe the replicated clusters of processor cores and memory. (More on this later.)

## A Different Application of Java

First, let's delve a little deeper into the object-oriented programming model. Ambric bases its software-development tools on the open-source Eclipse framework, which is proving to be a godsend for startups that need to bring up a tool suite quickly. For convenience, Ambric's tools use a strict subset of Java as the source language for objects, but without a Java bytecode compiler. Instead, Ambric's tools statically compile the Java source code into the native machine language of the proprietary 32-bit RISC processors.

For maximum performance in critical loops and subroutines, programmers can write machine-language objects using Ambric's assembler. Whether programmers use Java or assembly language, they write the objects in conventional sequential code. The parallelism lies in multiple objects running simultaneously on the parallel-processor array using the on-chip network of channels.

To link the objects together in parallel structures for the channels, programmers can use a graphical interface or a textual language called aStruct. Both are unique to Ambric's tools. Figure 3 shows an example of the graphical interface. It presents a graphical representation of the program's structural model, which can be a good way to visualize a parallel process. However, Ambric is aware that programmers have largely rejected many previous attempts to promote graphical programming tools, so this interface is strictly optional. It's most useful for programmers new to Ambric's architecture. After learning the ropes, they will probably switch to the aStruct language.

When using aStruct, programmers can express parallelism by creating multiple instantiations (objects) of the same class (object template). Keep in mind that the objects themselves are written in single-threaded Java code—Ambric doesn't use Java's standard Thread class to express parallelism. Instead, programmers express parallelism by using aStruct to instantiate multiple objects, then bind those

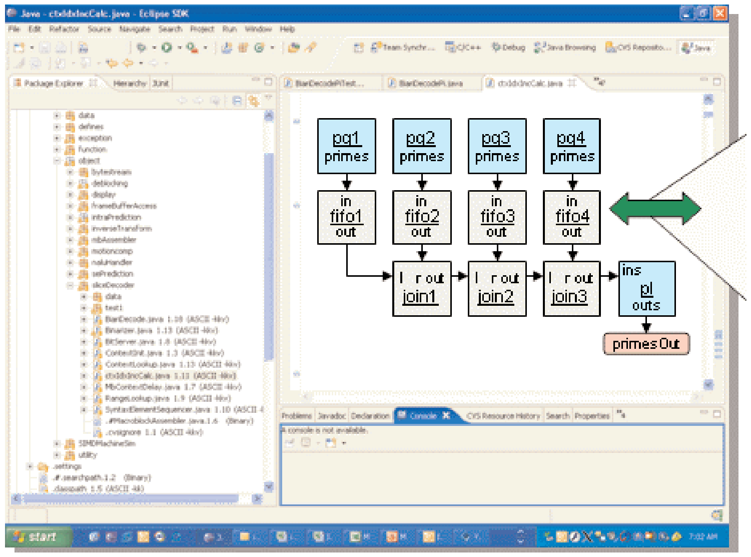

**Figure 3.** Ambric's proprietary software-development tools are based on the Eclipse integrated development environment (IDE). The structural editor shown here allows programmers to use an optional graphical interface that represents code objects as blocks, with arrows indicating the channels. Programmers write code for the objects in the conventional fashion, using a subset of Java or a proprietary assembler, then can link the objects together for parallelism using this interface.

objects to the self-regulating channels. That binding is visible in the 12 channel assignments in Figure 4, which shows the textual aStruct code for the structure in Figure 3.

## Channels Automatically Regulate the Flow

The simple example program in Figure 4 generates an ordered list of prime numbers. Although it's not the best way to find primes, it's a good challenge for parallel processing, because the time required to evaluate each number varies widely and unpredictably. Notice that the aStruct code in Figure 4 creates four objects of the PrimeGen class, which a programmer wrote in Java. A PrimeGen object tests the candidate integers to determine which are prime numbers; true primes eventually join other primes in an output stream. All together, the program can evaluate four candidates and return four results in parallel.

More important, the program orders the list of primes by using channels to synchronize the parallel operations. Synchronization is necessary because the number of iterations required to test each candidate varies. An integer requiring only a few tests to determine if it's prime could bypass another integer that requires more tests. If the channels didn't keep the parallel operations in sync, the list of primes would be out of order. Figure 5 shows the source code for the PrimeGen class.

FIFO buffers temporarily hold the primes until they are ready to take their proper place in the joined output stream. The buffers can't overflow, because if they're full, their input channels automatically generate "back pressure" to stall their associated PrimeGen objects from sending more results. Likewise, the output channels will stall the FIFO buffers from dispatching prime numbers into the final output stream until the order is correct. Essentially, the buffers provide load balancing. The program could work without buffers, but the PrimeGen objects would stall more often, because they couldn't begin testing another candidate integer until the previous candidate was ready to join the output stream in order.

By automatically stalling or releasing the registers at each end, the channels regulate every step of the flow. Programmers get the benefit of parallel processing by writing fairly conventional single-threaded Java code, then binding multiple instances of objects to input and output channels in parallel aStruct code.

Ambric's subset of Java doesn't stray far from Sun's Java standard. In fact, programmers can compile their Java source code using Sun's standard JAVAC compiler on a PC. The resulting Java bytecode runs on Ambric's functional simulator, which itself is a Java program running on a PC. The simulator allows programmers to debug system behavior before recompiling their code for Ambric's proprietary architecture.

The alternative to Ambric's approach would be to invent a completely new language wholly unfamiliar to programmers. That's basically what IntellaSys did for the SEAforth parallel architecture mentioned above. Chuck Moore, the brains behind SEAforth, reinvented the Forth language he first introduced in the 1970s, creating VentureForth. Either approach works, but today's programmers will probably find Java and aStruct easier to learn than VentureForth.

```
binding PrimeMaker Impl
    implements PrimeMaker {
  PrimeGen pg1 = {min = 3, increment =
    4, max = IPrimeMaker.max};
  PrimeGen pg2 = {min = 5, increment =
    4, max = IPrimeMaker.max};
  PrimeGen pg3 = {min = 7, increment =
    4, max = IPrimeMaker.max};
  PrimeGen pg4 = {min = 9, increment =
    4, max = IPrimeMaker.max};
  Fifo fifo1 = {max_size = fifoSize};
  Fifo fifo2 = {max_size = fifoSize};
  Fifo fifo3 = {max_size = fifoSize};
  Fifo fifo4 = {max_size = fifoSize};
  AltWordJoin join1;
  AltWordJoin join2;
  AltWordJoin join3;
  PrimeList pl;
  channel
    c0 = {pg1.primes, f1.in},
    c1 = {pg2.primes, f2.in},
    c2 = {pg3.primes, f3.in},
    c3 = {pg4.primes, f4.in},
    c4 = {f1.out , j1.1},
    c5 = {f2.out , j1.r},
    c6 = {j1.out , j2.1},
    c7 = {f3.out , j2.r},
    c8 = {j2.out , j3.l},
    c9 = {f4.out , j3.r},
    c10 = {j3.out , pl.ins},
    c11 = {pl.outs , primesOut};
}
```

**Figure 4.** This is a block of aStruct code, Ambric's textual source code for creating parallel structures of objects. This sample code is part of a program that generates prime numbers. It instantiates four user-written Java objects (PrimeGen and PrimeList), four FIFO buffers, and three channel-join objects. Everything is connected by a structure of 12 input and output channels. Figure 3 shows a graphical representation of the same structure.

```
public void PrimeGen (Output Stream<Integer> primes) {
    for (int candidate = min; candidate <= max;
                          candidate += 2*increment) {
        int factor;
        for (factor = 3; factor <= max; factor +=2) {
            if (candidate % factor == 0) break;
        }
        if (candidate == factor)  {// is prime
            primes.write(candidate) ; // write out
        }
        else primes.write(0);
    }
}
```

**Figure 5.** This Java source code defines a class named PrimeGen. Objects instantiated from this class can test candidate integers to determine which are true prime numbers. If a candidate is prime, the object sends out the number on a channel. If the candidate isn't prime, the object sends out a zero. Other objects downstream in Ambric's processor array collect the output of four PrimeGen objects to produce the final output stream, an ordered list of prime numbers.

## Using Brics for Building Blocks

Starting from this structural programming model, Ambric designed the microprocessor array architecture. The massively parallel array consists of numerous replicated clusters called brics. All the brics on a chip are functionally identical, though some are slapped down as mirror-image blocks by flipping the layout. Ambric designs a chip by a simple step-and-repeat process of laying out an array of these brics, then surrounding the array with standard I/O interfaces.

Components inside a bric may vary, depending on the implementation. All the brics in the Am2045 contain 32-bit streaming RISC processor cores with integer ALUs. Future implementations could have different processor cores, more cores per bric, more memory per core, wider datapaths, FPUs, and other variations. Some variations might break code compatibility, but that's not a major drawback, because they would aim for very different applications.

One improvement on Ambric's roadmap is logic optimization. To get the first chip out the door quickly, Ambric synthesized all the logic in the Am2045's brics, using standard cells. For the next-generation chip, Ambric plans to tweak some critical paths in the ALUs and memory interfaces, in addition to a process shrink to 90nm. Hand-optimizing a little logic would go a long way, because Ambric's step-and-repeat layout replicates the brics en masse.

Ambric designed two 32-bit RISC processors for the Am2045's brics. The simpler core is called the Streaming RISC (SR) processor, and the beefier core is called the Streaming RISC with DSP extensions (SRD) processor. SR processors are intended to run smaller code objects, particularly for control purposes or to prepare data for additional processing by the more powerful SRD processors. This division of labor—along with the streaming RISC architecture—allows the SRD processors to sustain higher throughput.

Figure 6 shows a block diagram of the simpler SR core. It has a single ALU capable of executing one 32-bit operation or two 16-bit operations per clock cycle. For high code density, instructions are only 16 bits long. These instructions follow the traditional RISC canon; most accept two 32-bit input operands and generate a 32-bit result. There are eight general-purpose 32-bit registers and 256 bytes of local memory for caching instructions or data.

Figure 7 is a block diagram of an SRD processor. It's quite a bit more powerful than an SR core. It has 32-bit-long instructions and three 32-bit ALUs—two in series, one in parallel—allowing two-way, or even three-way, instruction-level parallelism. Each of the two series ALUs can execute one 32-bit operation, two 16-bit operations, or four 8-bit operations per clock cycle. The third ALU is pipelined for rapidly executing multiply-accumulate (MAC) and sum of absolute differences (SAD) instructions. This ALU can execute one 32- × 8-bit operation or two 16- × 8-bit operations per cycle. The SRD core has 18 general-purpose 32-bit registers and a 64-bit accumulator, plus 1KB of local memory for caching instructions or data.

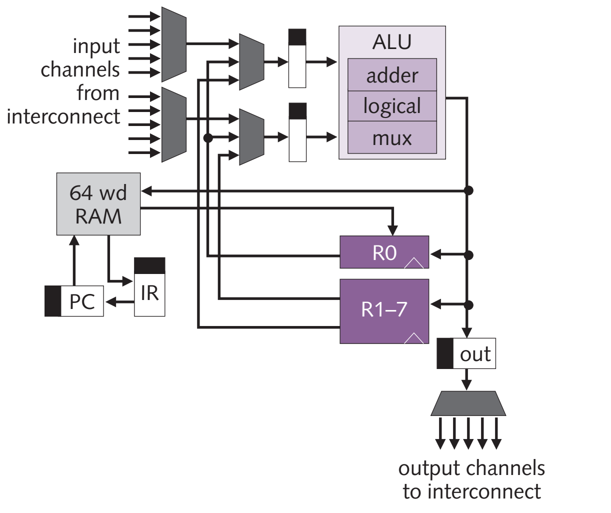

**Figure 6.** Streaming RISC (SR) processor block diagram. This is the simpler of the two proprietary 32-bit RISC cores that Ambric designed for the Am2045 and future chips in the family. It executes 16-bit-long instructions and has a small set of eight 32-bit registers. Note that these registers are separate from the registers at each end of the input and output channels connecting the cores together. Nevertheless, the processor can access the channel registers as easily as it accesses the general-purpose registers. Either type of register can hold input operands for instructions and receive results.

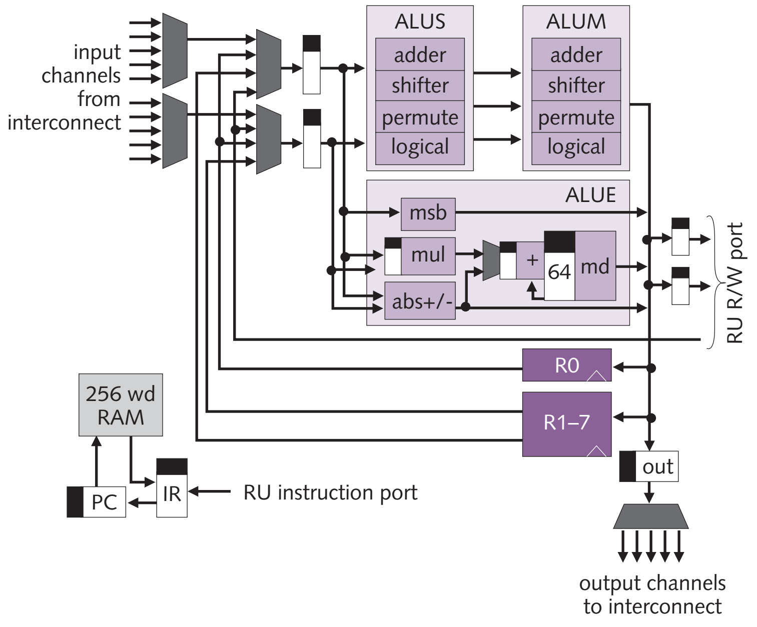

**Figure 7.** Streaming RISC with DSP (SRD) processor block diagram. Unlike the simpler SR processor, this core is built for heavy lifting. In addition to having longer instructions and more ALUs and registers, it has special read/write channels that allow it to quickly fetch instructions and data from elsewhere on the chip. An extended instruction set supports digital-signal processing.

## Brics Are Clusters of Processor Cores

To make a bric, Ambric starts with a cluster of SR and SRD processor cores, then adds more local memory and all the interconnects and control structures necessary for rapid communications. Next, Ambric flips the layout of that cluster to make a mirror-image of it. Joining those two clusters together makes a single bric. The final step is to assemble numerous brics into an array and wrap the whole thing in external I/O interfaces.

Figure 8 is a block diagram of the basic cluster (half a bric). It has four processor cores—two SR cores and two SRD cores—linked by a dense network of interconnect channels. This four-processor cluster connects to four local memories, each 1KB in size. Note that these memories supplement the more tightly coupled memories associated with each processor (256 bytes in each SR core, 1KB in each SRD core). The four processors in the cluster can share the four 1KB memory banks in various ways, as the software requires. The memory banks can store instructions or data and can serve as FIFO buffers.

Figure 9 shows a complete bric surrounded by parts of other brics. Flipping the layout for each half bric is important, because it arranges the clusters of processor cores and local memories very close together. Rapid communications between neighboring brics allow the chip to use larger virtual clusters of processors and memories, if necessary. If a complex software object is too large to run on a single bric, the processors of one bric can share the memory of neighboring brics. This arrangement allows a processor to access up to 16KB of local memory—that is, 16 of the 1KB local memory banks in four adjoining brics. (Again, this memory is in addition to the tightly coupled memory in each processor core.)

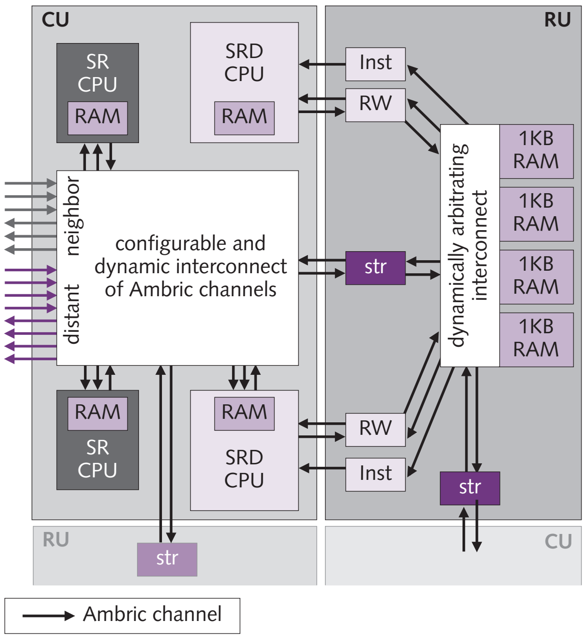

**Figure 8.** This block diagram shows a basic cluster of four processor cores, four local memories, and their associated interconnects and control structures. This cluster is half a bric. By flipping the layout and joining the two halves together, Ambric makes a whole bric. The final chip has a large array of brics. The array can be almost any size, within the limits of the fabrication technology.

Ambric's first chip, the Am2045, has a 5×9 array of brics. That's 45 brics, each with eight processor cores (four SR cores and four SRD cores) and 8KB of local memory, for a total of 360 processors and 360KB of SRAM. Add the tightly coupled memories in the processors, and total on-chip memory is 585KB. To finish the design quickly, Ambric licensed industry-standard I/Os from other vendors. The Am2045 has two DDR2-400 DRAM controllers, a four-lane PCI Express controller, 128 general-purpose I/O ports (100MHz), a serial flash-memory interface, an eight-bit host-processor interface, and a JTAG debug port. Figure 10 shows a physical layout of the Am2045.

What remains to be seen is how efficiently Ambric's proprietary software-development tools map the application software onto the processor array. Ambric's Java and aStruct languages and tools look easy to learn, and the microprocessor architecture is relatively comprehensible for a massively parallel device. But between the source code and the hardware is the critical back end of Ambric's development tools. First, the compiler must convert each object's high-level source code into efficient machine code. Then, it must distribute the executable code objects across the array of brics in a manner that efficiently utilizes the 360 processor cores, 360 tightly coupled memories, and 360 local memory banks. Managing off-chip resources (such as external DRAM for large data streams) is another problem to solve.

Whether Ambric's brand-new tools can accomplish all those feats is the critical question. Mastering the interaction between software and hardware is usually the biggest challenge for any extreme microprocessor architecture. This question won't be fully answered until developers start writing real-world programs for the Am2045.

## Ambric's Benchmarks Promise Performance

Until then, we do have some benchmark results from Ambric. In theory, the Am2045's maximum performance is 1.08 TOPS. That figure assumes that all 360 processors on the Am2045 are running full blast at 333MHz. (The eight processors in each bric can execute a maximum 24 billion operations per second at that frequency.) Table 1 compares the performance of some actual code running on an Am2045 with a simulated larger version of the chip, with assembly code running on a Texas Instruments C641*x*-family DSP, and with a Xilinx Virtex-4 LX100/LX200 FPGA. For this test, Ambric optimized the Am2045's compiled Java code with some hand-tweaked assembly language.

Comparing exotic microprocessor architectures is always difficult. They don't have much in common, and industry-standard benchmark suites don't adequately measure their performance. This is particularly true for massively parallel architectures, because benchmark programs are usually small, single-threaded kernels with little exploitable data parallelism. However, *MPR* has covered several massively parallel processors with similarities to Ambric's architecture.

One intriguing competitor is MathStar, a young fabless semiconductor company that already claims a dozen design wins. (See *MPR 7/24/06-02*, "MathStar Challenges FPGAs.") MathStar calls its chip a field-programmable object array (FPOA). It's similar to an FPGA, but developers work at a higher level of abstraction. Instead of programming gate arrays, developers use a massively parallel array of preconfigured function units called Silicon Objects. Most Silicon Objects execute 16-bit integer instructions or MACs in one clock cycle. MathStar's first FPOA has 400 Silicon Objects surrounded by banks of SRAM and external I/O interfaces.

MathStar's chip runs at 1.0GHz, so the maximum theoretical throughput is 400 billion operations per second. That's less than half the peak throughput of the Am2045 at three times the clock frequency. Another disadvantage for MathStar is that developers must program the FPOA using a hardware-design language (HDL), which requires developers to explicitly schedule the tasks. In contrast, Ambric provides a high-level software language whose objects run on a timing-independent, globally asynchronous array. However, it's possible that MathStar's closer-to-the-metal development tools will do a better job of mapping algorithms to a massively parallel array than Ambric's tools will. Both companies have shown examples of efficient digital-video processing. (Ambric's example uses 89% of the chip's compute capacity.)

Ambric will face additional digital-video competition from Connex Technology, another fabless-semiconductor startup. (See *MPR 1/9/06-01*, "Massively Parallel Digital Video.") The first commercial implementation of Connex's fascinating architecture has 1,024 small processor cores that can operate on an equal number of data words simultaneously. Unlike Ambric, Connex has extended an existing high-level language with special constructs for expressing parallelism. Programmers use a proprietary version of ANSI C called Connex Programming Language (CPL). Connex seems to be focusing more narrowly on digital video than Ambric is, which may give Connex an advantage in its target market (consumer digital TV). The learning curve appears similar for both companies' architectures and tools.

## More Parallel-Processor Competitors

Don't forget the Brits, who invented some primitive but prescient parallel-processing machines to crack German ciphers during World War II. Elixent, based in Bristol, England, is a

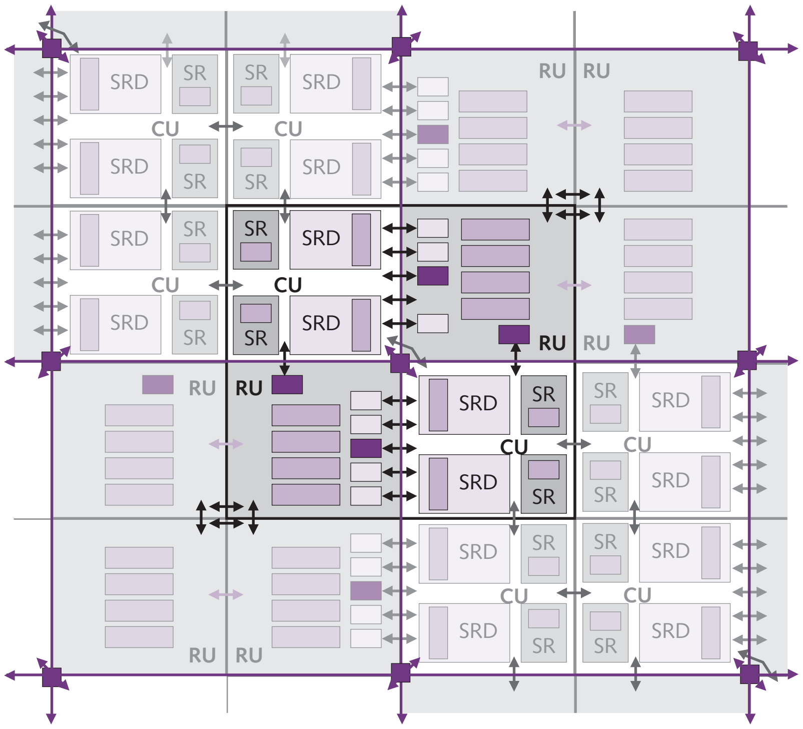

**Figure 9.** This block diagram shows one complete bric in the center, surrounded by parts of eight adjacent brics. Each bric consists of two processor-memory clusters, flipped to create two halves that are mirror images of each other. In this diagram, halves of adjacent brics are visible in the north, south, east, and west directions. In each of the four corners of the diagram, only one-fourth of the diagonally adjacent brics is visible. Fast interconnects allow the processor cores in one bric to access memory banks in adjacent brics.

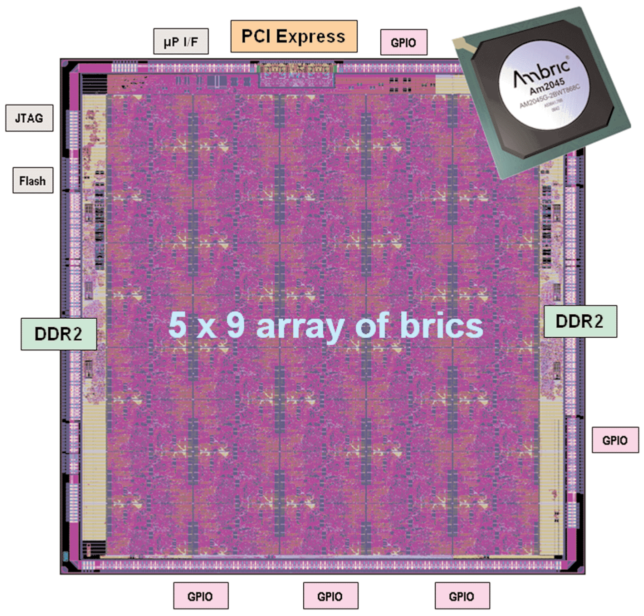

**Figure 10.** Am2045 chip layout. Although Ambric synthesized the brics in standard-cell logic, it's still possible to distinguish them in the sea of gates. The local memory banks and the edges of the replicated brics are clearly visible.

## Price & Availability

Ambric plans to have engineering samples of the Am2045 in January 2007, with general availability following in 2Q07. Development boards with one or two chips will be available in January. Ambric will offer three additional chips based on the same die. By turning off sections of the die that have manufacturing defects, Ambric can obtain higher yields while still providing usable (though less powerful) chips. The AM2045, with 360 processors, will cost $450 in 1,000-unit volumes. Other chips in the family are the Am2035 (288 processors) for $345; the Am2024 (192 processors) for $225; and the Am2012 (96 processors) for $99. For more information about Ambric, visit *www.ambric.com*.

five-year-old spinoff from Hewlett-Packard Labs. (See *MPR 6/27/05-02*, "Elixent Improves D-Fabrix.") D-Fabrix is an array of four-bit ALUs, multiplexers, registers, local memories, and switchboxes that work together in a massively parallel on-chip fabric. Proprietary tools allow developers to configure the fabric for their applications.

Elixent's business model differs significantly from Ambric's. Whereas Ambric sells its chips as standard parts, Elixent licenses D-Fabrix as a hard macro for SoC integration. The size of the fabric depends on the implementation. The first D-Fabrix chip is Toshiba's ET-1, which has 1,152 ALUs. Another early D-Fabrix adopter is Matsushita Electric. Elixent's customers have the flexibility to design a parallel-processing array exactly the right size for their applications, but Ambric's chips are available off the shelf. Another consideration is that D-Fabrix developers must use Verilog, VHDL, a special version of C adapted for hardware design (Celoxica's Handel-C), or a data-visualization and algorithm-development tool (MathWorks' Matlab).

PicoChip Design is another U.K. company with a massively parallel architecture. PicoChip's first device was the PC101, unveiled at Embedded Processor Forum 2003. The PC101's PicoArray has 430 processors capable of executing 19.2 billion 16-bit MACs per second at only 160MHz. (See *MPR 7/28/03-02*, "PicoChip Preaches Parallelism.") Six months later, PicoChip announced the PC102, which has 344 processors but more MAC units. It can execute 41.6 billion MACs per second at 160MHz—about 70% of the performance of Ambric's Am2045, albeit at half the clock rate. (See *MPR 10/14/03-03*, "PicoChip Makes a Big MAC.")

Although the PC101 and PC102 are communications chips for cellular telephony and wireless networks, they're obviously suitable for other signal-processing applications. Programmers can write software using an ANSI C compiler and assembler, but they must explicitly specify signal flows among the processors using structural VHDL. (The VHDL has nothing to do with logic synthesis, because PicoChip's devices are off-the-shelf parts.) In contrast, Ambric's customers don't need to explicitly set and balance signal flows among the processors, because the special channels automatically govern those communications.

The Dutch are weighing in, too. Silicon Hive, a Netherlands-based subsidiary of Royal Philips Electronics, has an exotic architecture based on ultralong instruction words (ULIW). (See *MPR 6/20/05-01*, "Busy Bees at Silicon Hive.") It's not a massively parallel architecture, but it's designed to exploit parallelism in the critical loops of target applications. A single instruction word can stretch as long as 918 bits, which should intrigue developers whose algorithms and data offer opportunities for extensive vector processing.

Like Ambric, Silicon Hive wants to replace high-end DSPs and ASICs in data-intensive applications. The proprietary HiveCC compiler allows programmers to write signal-processing code in C, without diving into assembly language. However, Silicon Hive acknowledges that programmers must use special intrinsic functions and pragmas to express more instruction-level parallelism than the compiler can find automatically.

### Rough Road for Extreme Processors

In addition to all the architecturally similar (or similarly exotic) competitors, Ambric is going into battle against conventional programmable solutions—high-end DSPs, FPGAs, ASSPs, multicore SoCs, configurable processor cores, and even some general-purpose microprocessors. We pity the project manager who tries to evaluate all these alternatives in the detail they deserve. Merely comparing the massively parallel architectures and their proprietary development tools is enough to boggle the mind. It's amazing that just a few years ago, high-end DSPs like TI's 'C64 family were considered extreme architectures. Today their VLIW architectures seem almost mundane in comparison.

| | Ambric AM2045 (45 BRICs) | Ambric AMxx70 (70 BRICs) | Texas Instruments C641x DSP | Xilinx Virtex-4 LX100 / LX200 |
|---|---|---|---|---|
| IC Process | 130nm | 90nm | 90nm | 90nm |
| Core Freq | 333MHz | 450MHz* | 1.0GHz | 500MHz |
| Published DSP Benchmarks | 10x–25x | 20x–50x* | 1x | n/a |
| MACs (16bx16b=32b) | 60 billion/sec | 125 billion/sec* | 4 billion/sec | 48 billion/sec |
| SADs | 240 billion/sec | 485 billion/sec* | 8 billion/sec | n/a |

**Table 1.** Despite having the disadvantage of an older fabrication process and slower clock frequency, the Am2045 outperformed a high-end TI DSP and Xilinx FPGA when running loops of multiply-accumulate (MAC) and sum of absolute differences (SAD) instructions. The 70-bric Ambric chip in the second column is a next-generation simulation, extrapolated from the performance of the 45-bric Am2045 chip. Ambric says that executable code for its processors is about one-third the size of TI's DSP code. (*Ambric's estimate. n/a: data not available.)

For a project manager, the easy way out is to ignore the new extreme architectures in favor of conventional solutions. In the old days, nobody ever got fired for buying IBM. Today, the safe bets are probably DSPs or ASSPs from major vendors like Analog Devices, Broadcom, or TI. But taking the safe route may overlook a more-flexible, higher-performance, lower-cost, lower-power architecture—albeit one that requires more time to understand. A competitor that spends that additional time might design a better product. Such are the hazards of architectural abundance.

Ambric is taking the right approach by starting with a relatively straightforward programming model and applying it to a flexible, powerful architecture. To succeed, companies like Ambric need as much expertise in tool development as they have in hardware development. If Ambric's tools work as well as the company promises, they will advance the state of the art for programming massively parallel arrays and give Ambric a fighting chance in the competition for design wins. ◇