Ambric

Process Network in Silicon:
A High-Productivity, Scalable Platform for High-Performance Embedded Computing

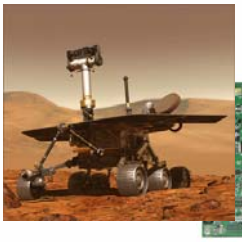
Mike Butts

mike@ambric.com / mbutts@ieee.org

Chess Seminar, UC Berkeley, 11/25/08

Focused on Embedded Systems

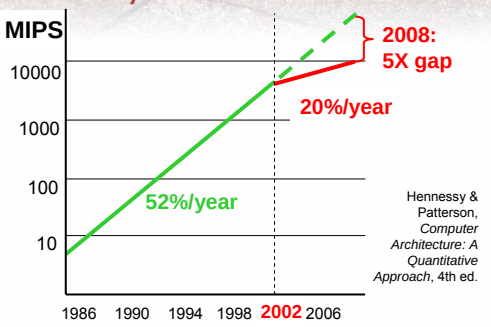
- Not **General-Purpose Systems**
  - Must support huge existing base of apps, which use very large shared memory
  - They now face severe scaling issues with **SMP** (symmetric multiprocessing)
- Not **GPU Supercomputers**
  - **SIMD** architecture for scientific computing
  - Depends on data parallelism
  - 170 watts!



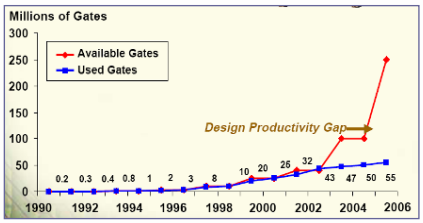
- **Embedded Systems**
  - Max performance on one job, strict limits on cost & power, hard real-time determinism
  - HW/SW developed from scratch by specialist engineers
  - Able to adopt new programming models

Scaling Limits: CPU/DSP, ASIC/FPGA

- Single CPU & DSP performance has fallen off Moore's Law
  - All the architectural features that turn Moore's Law area into speed have been used up.
  - Now it's just device speed.
- **CPU/DSP does not scale**
- ASIC project now up to $30M
  - NRE, Fab / Design, Validation
- HW Design Productivity Gap
  - Stuck at RTL
  - 21%/yr productivity vs 58%/yr Moore's Law
- ASICs limited now, FPGAs soon
- **ASIC/FPGA does not scale**



Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.



Gary Smith, *The Crisis of Complexity*, DAC 2003

Parallel Processing is the Only Choice

Parallel Platforms for Embedded Computing

- Program processors in software, far more productive than hardware design
- Massive parallelism is available
  - A basic pipelined 32-bit integer CPU takes less than 50,000 transistors
  - Medium-sized chip has over 100 million transistors available.
- But many parallel processors have been difficult to program.
- **The trick is to**
  1) Find the right programming model first,
  2) Arrange and connect the CPUs and memories to suit the model,
  3) To provide an efficient, scalable platform that's reasonable to program.
- Embedded computing is free to adopt a new platform
  - General-purpose platforms are bound by huge compatibility constraints
  - Embedded systems are specialized and implementation-specific

Choosing a Parallel Platform That Lasts

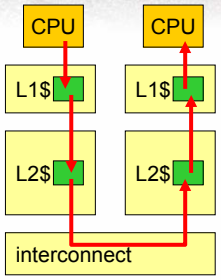
- How to choose a durable parallel platform for embedded computing?
  - Don't want adopt a new platform only to have to change again soon.
- Effective parallel computing depends on common-sense qualities:
  - **Suitability:** How well-suited is its architecture for the full range of high-performance embedded computing applications?
  - **Efficiency:** How much of the processors' potential performance can be achieved? How energy efficient and cost efficient is the resulting solution?
  - **Development Effort:** How much work to achieve a reliable result?
- Inter-processor communication and synchronization are key:
  - **Communication:** How easily can processors pass data and control from stage to stage, correctly and without interfering with each other?
  - **Synchronization:** How do processors coordinate with one another, to maintain the correct workflow?
  - **Scalability:** Will the hardware architecture and development effort scale up to a massively parallel system of hundreds or thousands of processors?

Symmetric Multiprocessing (SMP)

- Multiple processors share similar access to a common memory space: *Symmetric MultiProcessor*
- Incremental path from the old serial programming model
  - Each processor sees the same memory space it saw before.
  - Existing applications run unmodified (unaccelerated as well of course)
  - Old applications with millions of lines of code can run without modification.
- SMP programming model has task-level and thread-level parallelism.
  - Task-level is like multi-tasking operating system behavior on serial platforms.
- To use more parallelism the tasks must become parallel: *Multithreading*
  - Programmers write source code which forks off separate threads of execution
  - **Programmers must explicitly manage communication and synchronization**

SMP Communication

- In SMP *communication is a second-class function*.
  - Just a side-effect of shared memory.
- Data is copied five times through four memories and an interconnect.
  - The destination CPU must wait through a two-level cache miss to satisfy its read request.
- Poor cache reuse if the data only gets used once.
  - Pushes out other data, causing other cache misses.
- Communication thru shared memory is expensive in power compared with communicating directly.
- The way SMPs do inter-processor communication through shared memory is complex and expensive.

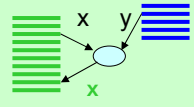


The Troubles with Threads

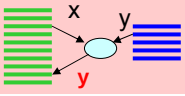
- SMP's multithreaded programming model is deeply flawed: *Multithreaded programs behave unpredictably.*
- Single-threaded (serial) program always goes through the same sequence of intermediate states, i.e. the values of its data structures, every time.
  - Testing a serial program for reliable behavior is reasonably practical.
- Multiple threads communicate with one another through shared variables:
  - Synchronization: partly one thread, partly the other
- Result depends on behavior of all threads.
  - Depends on dynamic behavior: indeterminate results.
  Untestable.

"If we expect concurrent programming to become mainstream, and if we demand reliability and predictability from programs, we must discard threads as a programming model." -- Prof. Edward Lee

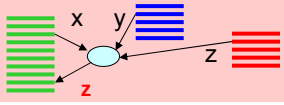
Intended behavior



Synchronization failure



Another thread may interfere



The Troubles with Threads

Ambric

- Avoiding synchronization errors is left entirely up to the programmer:
  - SMP's programming model and languages have no intrinsic protections.
  - Correct synchronization and "thread safety" is a DIY job for the programmers.
- This is hard enough for a single programmer, harder still when many programmers' modules are combined.
  - *"Programs that use threads can be extremely difficult for programmers to understand. If a program is incomprehensible, then no amount of process improvement will make it reliable." – Prof. Edward Lee*
- Reliability and predictability are especially essential to embedded systems.
  - Multithreaded desktop programs crash all the time.
  - Real-time embedded systems encounter much more variation of input data and timing than desktop and server systems ever see.
  - Automotive, medical, avionic systems are too vital to permit such failures.
- When parallelism doubles, their cross-product set of possibilities squares.
  - Massively parallel SMP: thread bug explosion
- **Multithreading is a scaling limit to SMPs.**

SMP Summary

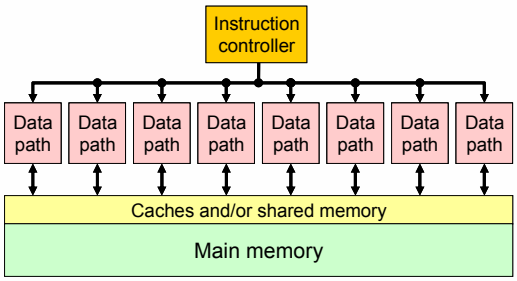
Ambric

- Easy use of general-purpose 2-to-4-way SMPs is misleading.
  - Big difference between small multicore SMP implementations, and massively parallel SMP's expensive interconnect, cache coherency
- SMPs are non-deterministic, and get worse as they get larger.
  - Debugging massively parallel multithreaded applications promises to be difficult.

Suitability: Limited. Intended for multicore general-purpose computing.
Efficiency: Fair, depending on caching, communication and synchronization.
Development effort: Poor: DIY synchronization, multithreaded debugging.
Communication: Poor: Complex, slow and wasteful.
Synchronization: Poor: DIY thread synchronization is difficult, dangerous.
Scalability: Poor: Interconnect architecture, communication through caches, and multithreaded synchronization problems indicate poor hardware and/or software scalability beyond the 2-to-8-way multicore level.

- **Massively parallel SMP platforms are unlikely to be well-suited to the development, reliability, cost, power needs of embedded systems.**

Single Instruction Multiple Data (SIMD)

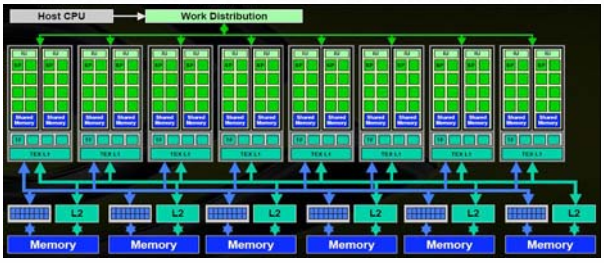


- Tens to hundreds or more datapaths, all under the control of a single instruction stream
  - Often a general-purpose host CPU executes the main application, with data transfer and calls to the SIMD processor for the compute-intensive kernels.
- SIMD has dominated high-performance computing (HPC) since the Cray-1.
  - Massively data-parallel, feed-forward and floating-point-intensive
  - Computational geoscience, chemistry and biology, structural analysis, medical image processing.

SIMD in embedded systems?

- SIMD's long pipelines are a disadvantage
  - When there are feedback loops in the dataflow (x[i] depends on x[i-n])
  - When data items are only a few words, or irregularly structured
  - When testing and branching (other than loops)
- Testing and branching on individual data values is expensive
  - All the non-tested datapaths are idle.
- SIMD is not as well suited to high-performance embedded applications
  - Which are often function-parallel
  - Which may have feedback paths and data-dependent behavior
  - Increasingly found in video codecs, software-defined radio, networking and elsewhere.
- Real-time H.264 High Profile encoding of broadcast-quality HD video
  - Massive parallelism is required
  - Feedback loops in the core algorithms
  - Many different subsystem algorithms, parameters, coefficients, etc. are used dynamically, in parallel (*function-parallel*), according to the video being encoded.

SIMD/SMP Example: GPU



NVIDIA: "GPU Parallel Computing Architecture and CUDA Programming Model"

- GPUs naturally have a SIMD internal architecture
  - Graphics has massive data parallelism, extremely regular and predictable.
- Recent GPUs have been extended and applied to HPC.
- Example: NVIDIA's CUDA architecture
  - SIMD processors for data-parallel floating-point processing.
  - SMP of these SIMD processors, with multi-level-cached shared memory.
  - GeForce 8800 GPUs: SMP with eight processors, each 16-wide SIMD
  - SMP multithreading communication, synchronization risks apply

SIMD Summary

- SIMD architectures were developed for the massively data-parallel feed-forward applications found in scientific computing and graphics.

Suitability: Limited. Intended for scientific computing (HPC).

Efficiency: Good to Poor: Good for data-parallel feed-forward computation. Otherwise it gets Poor quickly.

Development effort: Good to Poor:
Good for suitable applications, since there is a single instruction stream. Gets poor when forcing complexity, data-dependency into SIMD model.

Communication and Synchronization: Good by definition, everything's always on the same step.

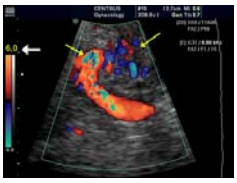
Scalability: Poor without lots of data parallelism available in the application. A few embedded applications have vector lengths in the hundreds to thousands, most don't.

- **Massively parallel SIMD platforms are unlikely to be well-suited to the most high-performance embedded system applications.**

Massively Parallel Processor Array (MPPA)

Ambric

- Developed specifically for high-performance embedded systems.
  - Video codecs, software-defined radio, radar, ultrasound, machine vision, image recognition, network processing.....
  - Continuous GB/s data in real time, often hard real-time.
  - Performance needed is growing exponentially.
- Function-parallel, data-parallel, feed-forward, feedback, data-dependent
- TeraOPS, low cost, power efficiency, and deterministic reliable behavior.

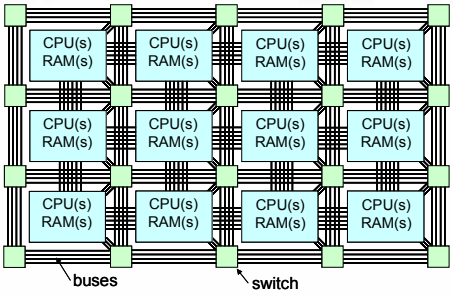


- MPPA platform objectives:
  1) Optimize performance, performance per watt
  2) Reasonable and reliable application development
  3) Moore's Law-scalable hardware architecture and development effort

MPPA Architecture

Ambric

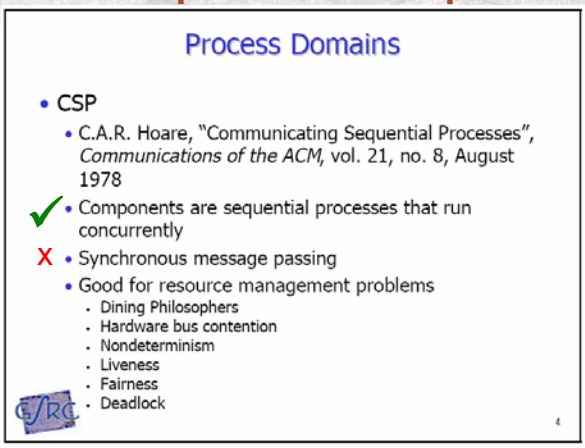
- Massively parallel array of CPUs and memories
- 2D-mesh configurable interconnect of word-wide buses
- MIMD architecture
  - Distributed memory
  - Strict encapsulation
  - Point-to-point communication



- Complex applications are decomposed into a hierarchy of subsystems and their component function objects, which run in parallel, each on their own processor.
- Likewise, large on-chip data objects are broken up and distributed into local memories with parallel access.
- Objects communicate over a parallel structure of dedicated channels.
- Programming model, communications and synchronization are all simple, which is good for development, debugging and reliability.

Model of Computation: not quite CSP

Ambric

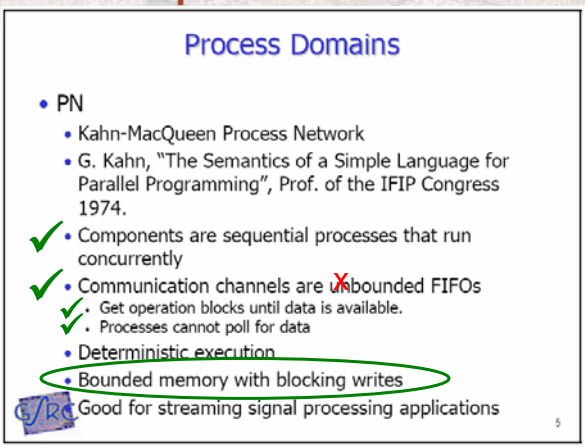


slides from "A Brief Tutorial on Models of Computation" The MESCAL Team, UC Berkeley, Fall 2001

- Ambric MoC was inspired by CSP, but is not quite CSP.
  - Message passing is buffered, not strictly synchronous.

Model of Computation: Process Network

Ambric

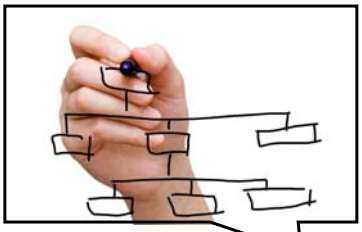


slides from "A Brief Tutorial on Models of Computation" The MESCAL Team, UC Berkeley, Fall 2001

- Ambric MoC is a Process Network with bounded FIFOs.
  - FIFO-like primitive register, streaming RAMs for bigger FIFOs.
  - Channels carry data and control, and strictly preserve sequence.

Choose the Right Programming Model *First*

Everyone draws block diagrams, great!



Everyone wants to write software, great!

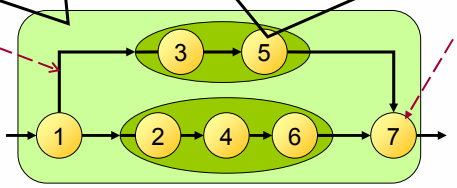
```
for (int r = 1; r <= numRefPictures; r ++)
{
    // generate TMS workload requests for current macroblock
    outCtl.writeTag(outCtl.intOfTag("TagCurrentBlock"));
    outCtl.writeDEO(((mbSizeX + 3) >> 2) * mbSizeY);
    GenerateWorkloadCB(outTms, addrPtr[0] + j, picLinePitch[0]);

    // zero motion vector
    outCtl.writeTag(outCtl.intOfTag("TagMotionVector"));
    outCtl.writeDEO(0);

    int xRadius = searchRFAxRadius[s];
```

Structure of ***self-synchronizing*** *Ambric channels*

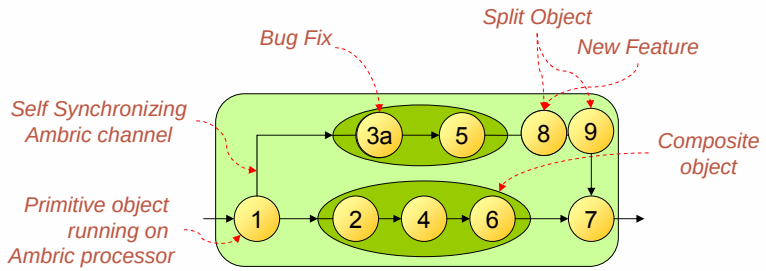
Software ***objects*** *run on CPUs*



Ambric's Structural Object Programming Model

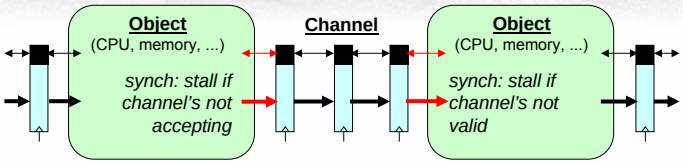
Structural Object Programming Model

- Objects are software programs running concurrently on an asynchronous array of Ambric processors and memories
- Objects exchange data and control through a structure of self-synchronizing asynchronous Ambric channels
- Objects are mixed and matched hierarchically to create new objects, snapped together through a simple common interface



Application

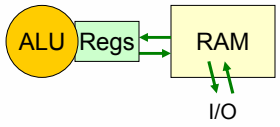
Process Network Realized in Silicon



- Objects exchange data and control thru a structure of Ambric channels
  - Each stage has forward and backward flow control, and buffering
- Standard interface between objects
  - Encapsulation, reuse
- Self-synchronizing on each transfer
  - Asynchronous system
- Channels can be any length or speed: no scheduling, no timing closure
  - Easier on tools, easier to program, easier to debug, reliable

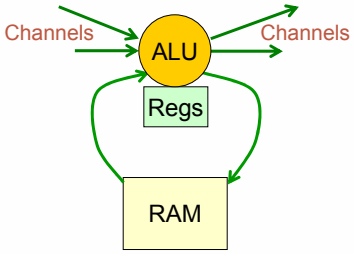
Traditional vs. Ambric Processors

- **Traditional processor architecture**
  - Primary: register-memory hierarchy
  - Secondary: communication



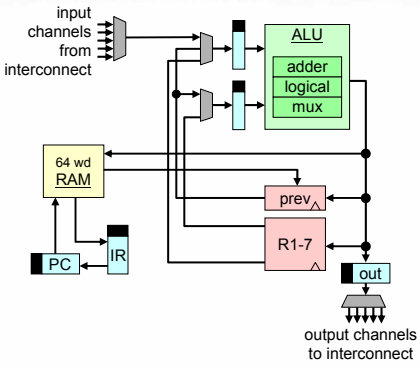
- **Ambric processor architecture**
  - Primary: communicate through channels
  - All data goes through channels
    - Memory
    - Registers
    - Inter-processor streams
    - Instruction streams

    to reduce local storage
  - Channels synchronize all events



Ambric SR Processor

- Simple 32-bit Streaming RISC
- Mainly for fast small utility objects:
  - complex addressing, complex fork/join, pack/unpack, serialize/deserialize
- Ambric channels
  - 1 input, 1 output per instruction
  - Instruction fields select inputs, outputs just like selecting registers
- One ALU: 32b or dual 16b ops
- 8 general registers
- 16 bit instructions for code density
  - Zero-overhead looping
- 64 word local code/data RAM
  - 128 instructions
- Three-stage Ambric channel datapath

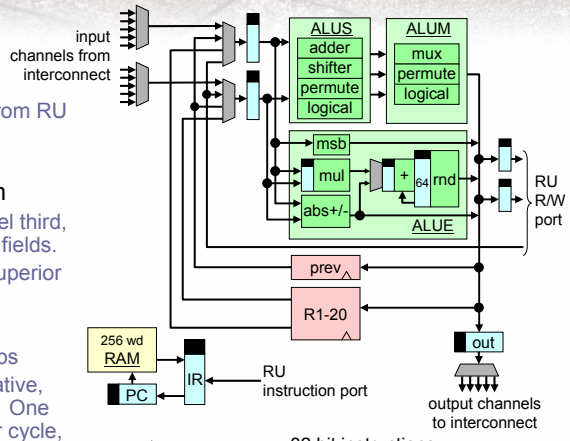


16 bit instructions

ALU op	src1	src2	dest	out
ALU op	src1	opcode	dest	out
mem r/w	src1	src2	address	
branch		condition	offset	
loopstart		count	offset	

Ambric SRD Processor

- Streaming RISC with DSP extensions
- 32 bit instructions
  - 256 in local RAM, more from RU
  - Zero-overhead looping
- Multiple ALUs capture instruction-level parallelism
  - 2 ALUs in series, a parallel third, with individual instruction fields.
  - For stream processing, superior code density to VLIW
- 3 ALUs
  - 32b, dual 16b, quad 8b ops
  - 3rd ALU alongside is iterative, pipelined, for MAC, SAD. One 32b*8b or two 16b*8b per cycle, 64-bit accumulator, rounding
- RU read-write channels
- 3-stage Ambric channel datapath

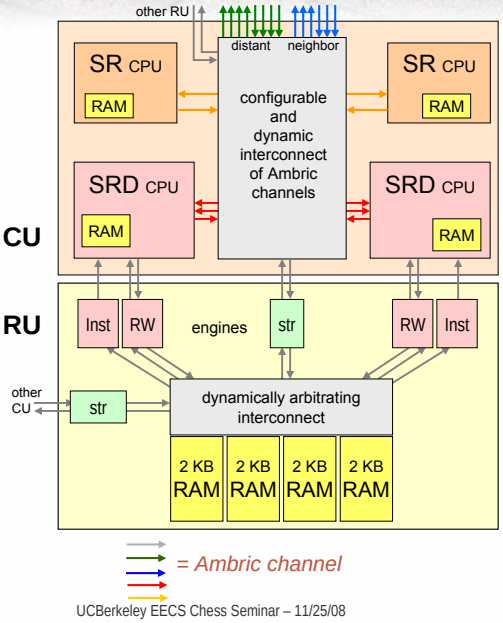


32 bit instructions

ALUS op	ALUM op	src1	src2	dest	out
ALUE op		src1	src2	dest	out
mem r/w	base	src1	src2	address	
branch		condition	offset		
loopstart		count	offset		

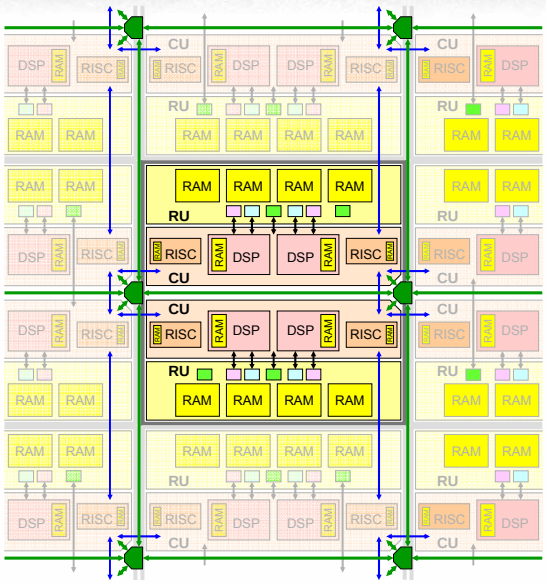
Compute Unit and RAM Unit

- Compute Unit (CU)
  - Two SRD 32-bit CPUs
  - Two SR 32-bit CPUs
  - Channel interconnect
- RAM Unit (RU)
  - Four 2KB RAM banks
  - RU engines turn RAM regions into channels
  - Engines dynamically connect to banks through channels with arbitration
- Local Channels connect CU and RU



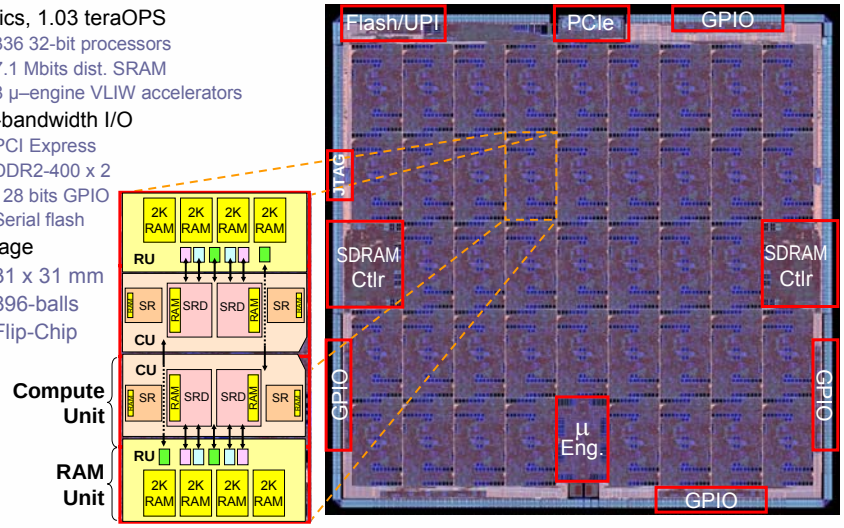
Brics with Configurable Channel Interconnect

- 2 CU-RU pairs form a Bric
  - Brics serve as the silicon building block
- Neighbor channels connect adjacent Brics
- Distant channels connect to other Brics via switches
  - No wires longer than a Bric



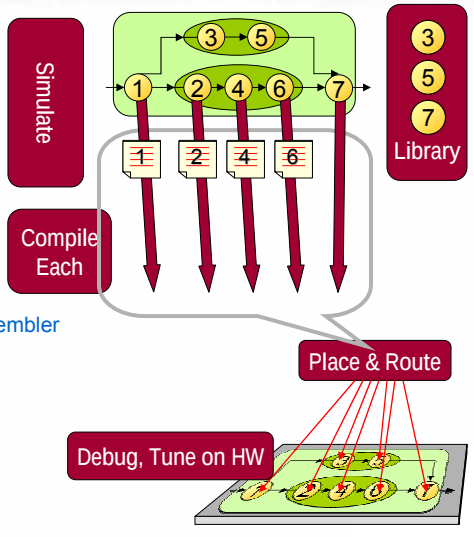
Am2045 Chip

- 130nm standard-cell ASIC
  - 180 million transistors
- 45 brics, 1.03 teraOPS
  - 336 32-bit processors
  - 7.1 Mbits dist. SRAM
  - 8 µ–engine VLIW accelerators
- High-bandwidth I/O
  - PCI Express
  - DDR2-400 x 2
  - 128 bits GPIO
  - Serial flash
- Package
  - 31 x 31 mm
  - 896-balls
  - Flip-Chip

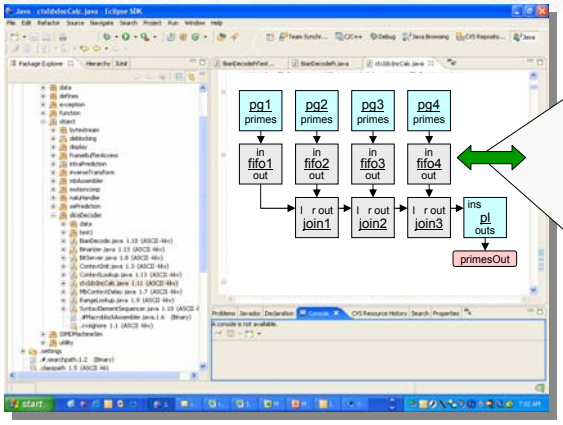


Ambric Development Tools: aDesigner

- Eclipse-based complete IDE
- Structure
  - Conceive your application as a structure of objects and the messages they exchange
  - Divide-and-conquer using hierarchy
- Reuse
  - Encapsulated library objects
- Code and Test
  - Write your new objects in Java or Assembler
  - Verify with functional simulation
- Realize on HW
  - Compile each object separately
  - Run mapper-router, configure chip
  - Debug, profile and tune performance



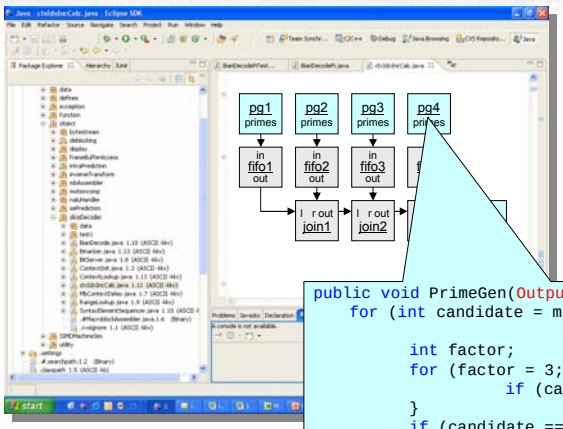
Structural Programming in aStruct




```
binding PrimeMakerImpl
     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.l},
    c5 = {f2.out , j1.r},
    c6 = {j1.out , j2.l},
    c7 = {f3.out , j2.r},
    c8 = {j2.out , j3.l},
    c9 = {f4.out , j3.r},
    c10 = {j3.out , pl.ins},
    c11 = {pl.outs , primesOut};
}
```

- Graphical or textual entry
  - menus or text (not shown) defines channel interfaces, object parameters
- Hierarchical, modular

Standard Language for Objects

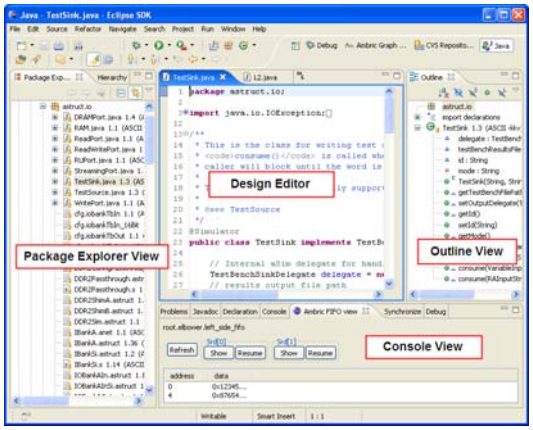


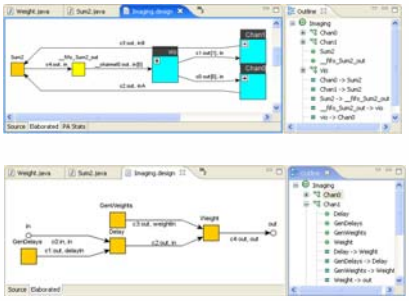
- Program primitive objects in Java
  - Strict subset of standard Java
    - static memory
  - Classes define the channels

```
public void PrimeGen(OutputStream<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);
    }
}
```

- Language-agnostic
  - C, etc. to follow

Design Entry with Graphical Viewing Capability

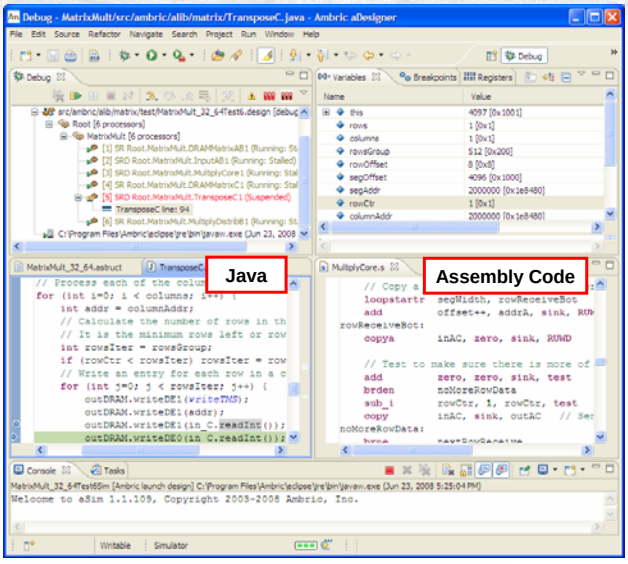




- **Elaborated hierarchical graphs**
  - Color coded objects: Leaf (orange); Memory (yellow); Composite (cyan)
  - Flexibility to expand composite object

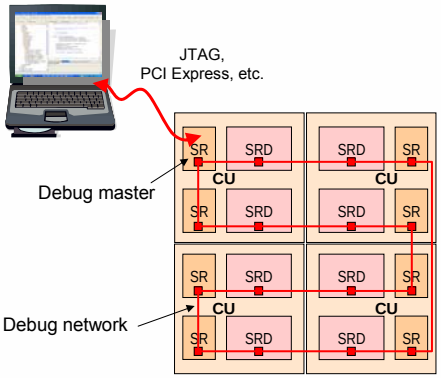
Functional Simulator

- **Enables top-down or bottom-up design development**
- **Ability to verify partial designs with Virtual I/Os**
- **Mix-and-match Java and assembly code in simulation**
- **Same stimulus can be used for JVM, ISS and on H/W**



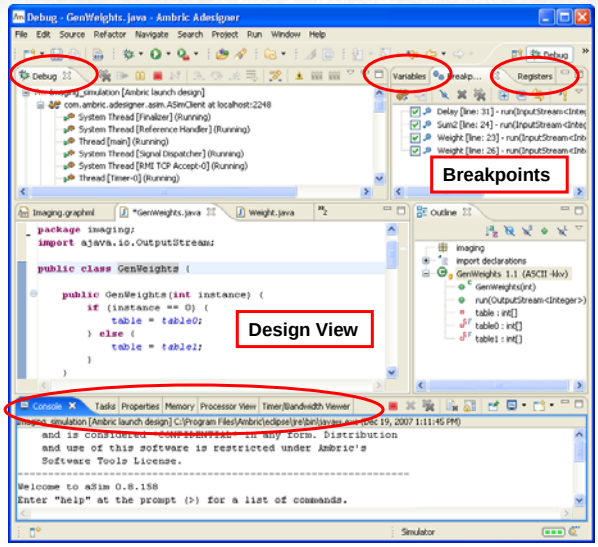
Direct Debugging of Hardware

- Debugger/Profiler in IDE
  - Integrated with source code
- Built-in Debug Network in silicon
  - Strictly separate network that can't deadlock
  - Non-intrusively observes processors for performance analysis
  - Controls processors for debugging



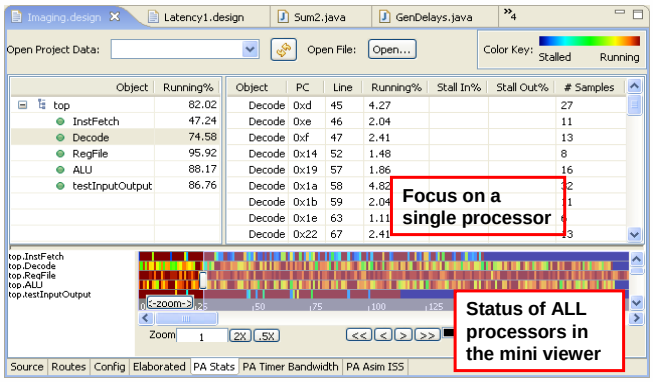
Interactive Parallel Debugging

- **Various windows to access debug information**
- **Symbolic references, single step and breakpoints**
- **Ability to debug across multiple processors concurrently**
- **Special features like:**
  - **Channel monitoring**
  - **Synchronization of stalling processors**



Performance Analysis in Hardware

- **View and tune processor performance for maximum utilization**
- **Non intrusive. No impact to design's runtime performance and behavior**



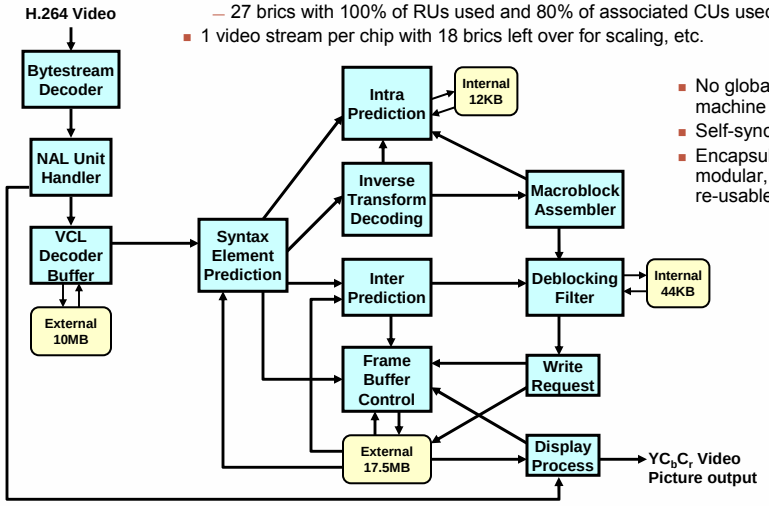
Many emerging application areas

All present massive compute demands growing without bounds

- Video processing
  - Digital video encoding/decoding/transcoding
    - Broadcast quality HD and/or multiple SD channels
  - Acceleration for desktop video editing and production
- Medical imaging
  - Ultrasound, X-ray, CAT, MRI, PET
- Software-defined radio
  - WiMAX, LTE base stations and terminal equipment
- Image processing and computer vision
  - Smart surveillance and security
  - Remote and autonomous vehicles, vehicle safety
  - Robotics
  - Object recognition for internet search engines, databases

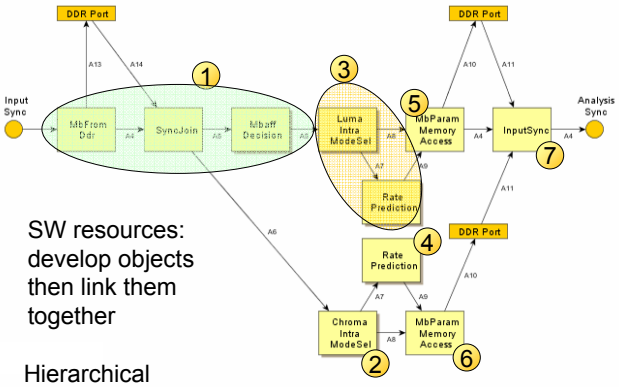
Example: HD H.264 Decoder

- HD-H.264, High Profile - Level 4
- Am2045 resources
  - 27 brics with 100% of RUs used and 80% of associated CUs used
- 1 video stream per chip with 18 brics left over for scaling, etc.



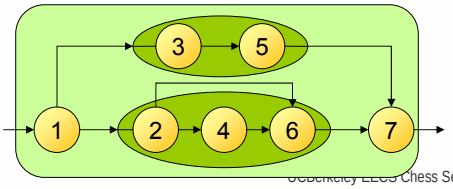
- No global state machine
- Self-synchronizing
- Encapsulated, modular, re-usable code

Application Example – AVC Intra



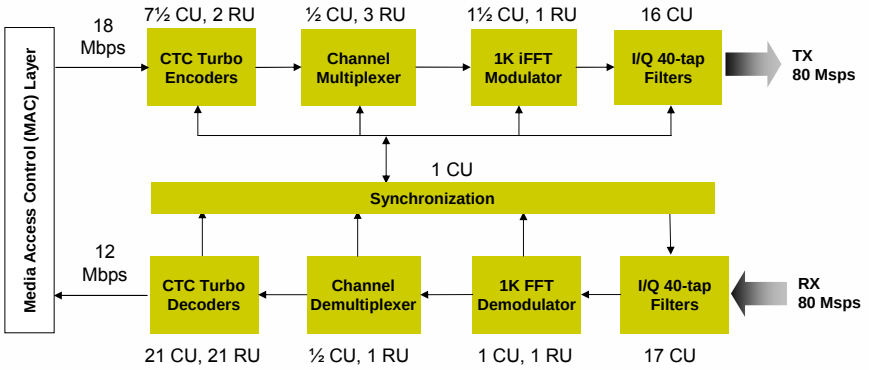
SW resources: develop objects then link them together

Hierarchical approach to development



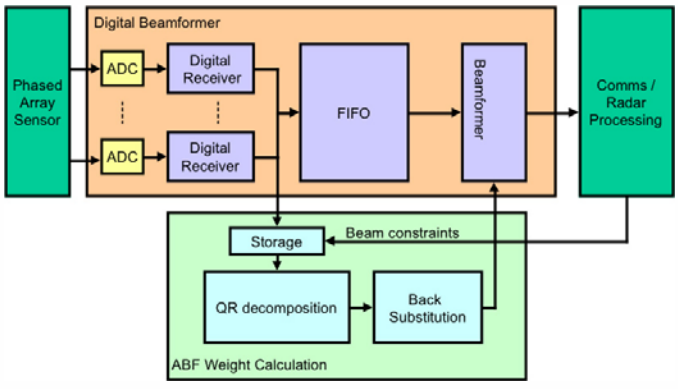
- Map SW resources to problem hierarchy
- Each team develops in predictable, stable environment
- Local changes stay local

Example: WiMAX Base Station Block Diagram



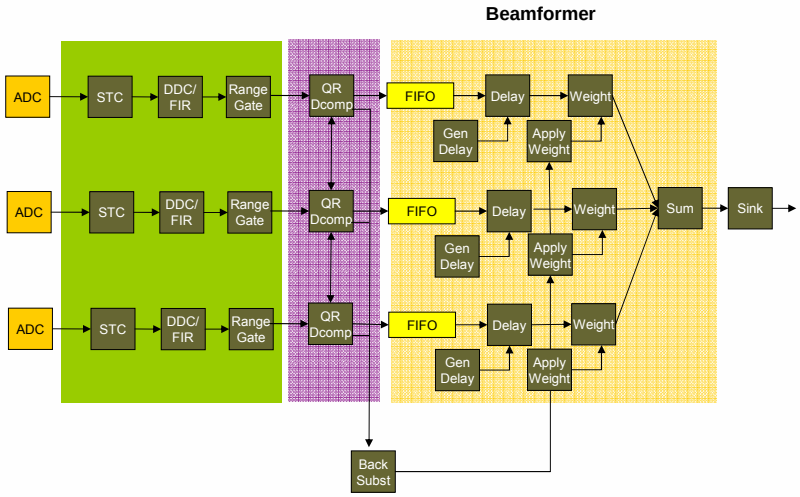
Total: 67 CU, 31 RU = 34 brics = 80% Am2045

Adaptive Beamforming



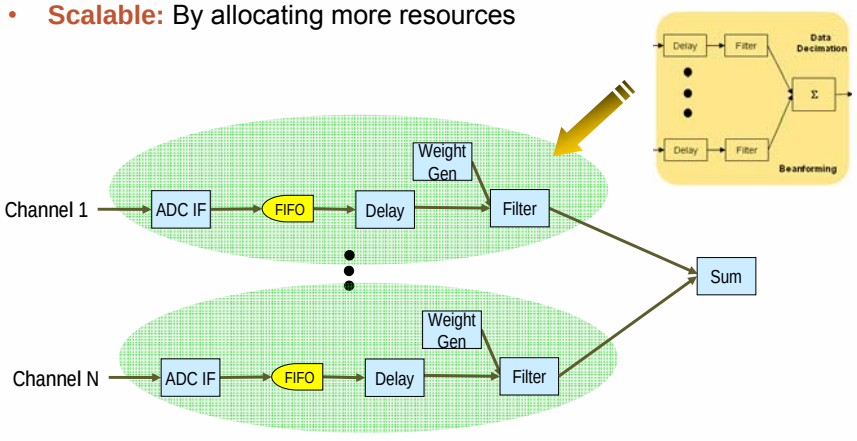
White Board Design

- Dissect the problem space into functional modules



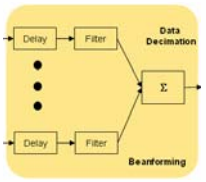
Beaming Forming Example

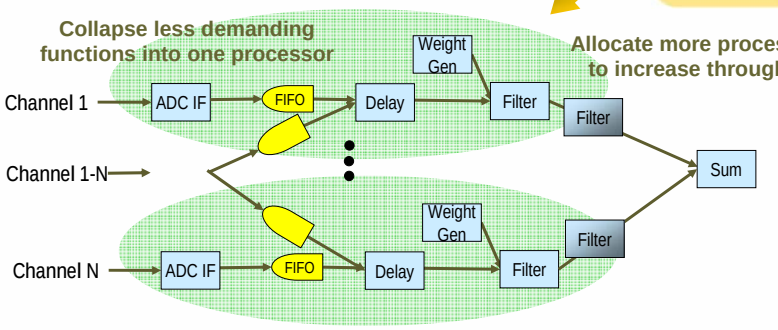
- **Programmable**: Map functional structure into objects
- **Predictable Performance**: Regulated by dataflow
- **Scalable:** By allocating more resources



Beaming Forming Example (cont.)

- **Scalable:** Adjust resource allocation according to performance requirements
- **Ease of Development:**
  - 32 Channels, 16 MHz sampling rate
  - Coded and debugged in 2 wks

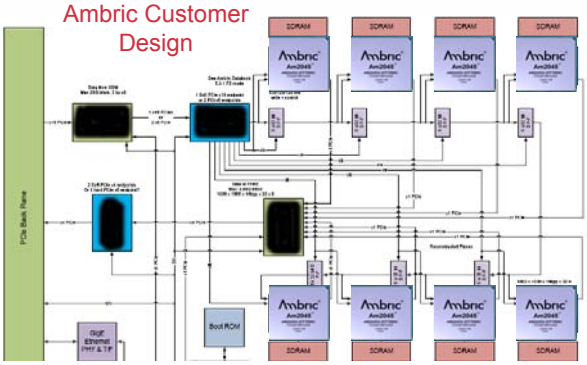




Realizing MPPA Solutions with Ambric

Worlds first and only silicon that was created for MPPA software tools that scales from 1 to 10,000+ processors

Ambric Customer Design



4 cards per system

8 AM2045/card

336 processors / AM2045 +

1 card with 4 AM2045

12K+ processors > 36 TeraOPS of performance

MPPA Summary

- MPPA hardware is dense, fast, efficient and scalable.
- Efficient, reliable applications, reasonable development effort, scalability.
  - Software must be rewritten into MPPA form

Suitability: Good. Developed specifically for embedded systems.

Efficiency: High. Data or functional parallel, feed-forward or feedback, regular or irregular control.

Development effort: Good, though apps must be written with MPPA in mind. Modularity and encapsulation help team development and code reuse. Whole classes of bugs, such as bad pointers, synchronization failures are impossible. Testing is practical and reliable.

Communication: Good: Direct comm'n is fast, reliable, deterministic, efficient.

Synchronization: Good: Built in. Fully deterministic.

Scalability: Good: Hardware is free from many-to-many interconnect and caching, and can easily use GALS clocking for very large arrays. Software development is free from SMP's multithreading and SIMD's vector-length scaling limits, and leverage from code reuse is effective.

What users are saying…

"Solving **real time high definition video processing** and digital cinema coding functions poses some unique programming challenges. Having an integrated tool suite that can simulate and execute the design in hardware **eases development of new products and features for high resolution and high frame-rate imaging** …"

Ari Presler, CEO of Silicon Imaging

"…designers are getting **implementation done in half the time**. Our engineers are even **having fun** using the tool!…"

Shawn Carnahan, CTO, Telestream

*…merous software development … the **quality of** …esigner tool suite** …orous tests on …ied development …essor development."*

…erest Consultants Inc.

"A general principle in **CERES' researc** … massive cooperation among a large numb … law, massive co-operation between proce… implement extremely high performance, yet progra… chip, as well as at the board, level present challenging research que… **with its hundreds of processing elements, is an attractive silicon platform fo…**

…l through the intelligent and …scaling following Moore's … makes it possible to …tions between the units at the **…ric's massively parallel chip, …systems** research."

Professor Svensson, Halmstad University, Sweden

***Over 40 active users* of aDesigner with more in the process of ramping up**

www.Ambric.com

- "Synchronization through Communication in a Massively Parallel Processor Array", Mike Butts, IEEE Micro, Sept/Oct 2007.
- "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing", Mike Butts, Anthony Mark Jones, Paul Wasson, IEEE FCCM, April 2007.
- "Multicore and Massively Parallel Platforms and Moore's Law Scalability", Mike Butts, Embedded Systems Conference, April 2008.