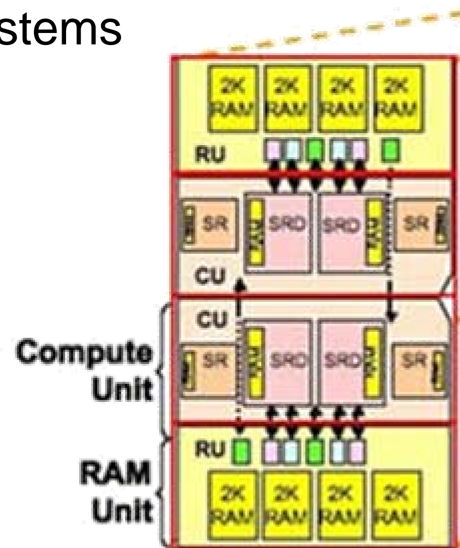
The Ambric Processor Array - a first look

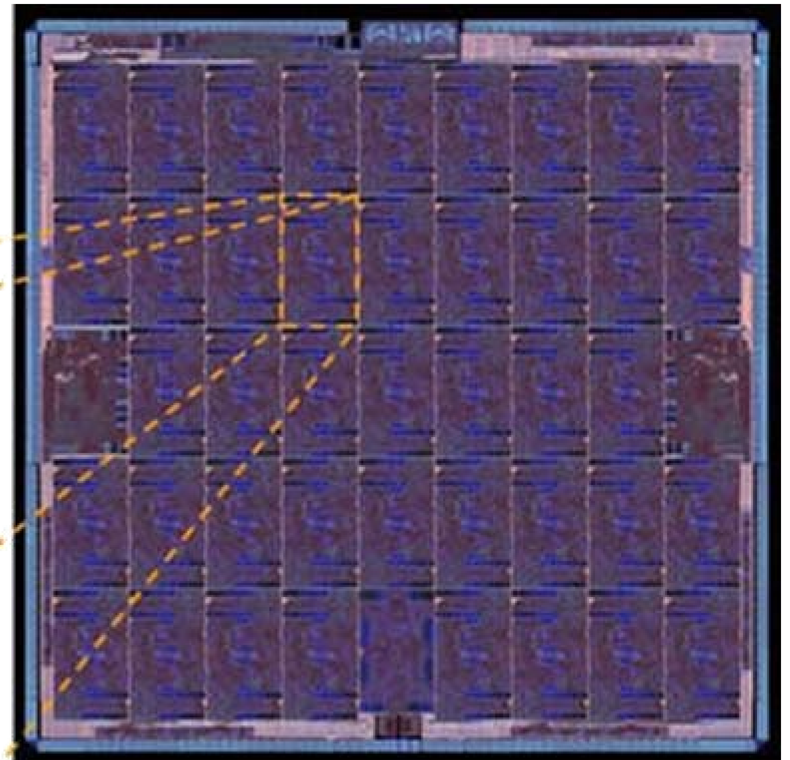
360 processors
@333 Mhz

Peak performance > 1 TOPS

117 million transistors

For embedded systems





Target Applications

- 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.

- Ambric claims it has achieved its goal of making a massively parallel processor that's relatively easy to program.

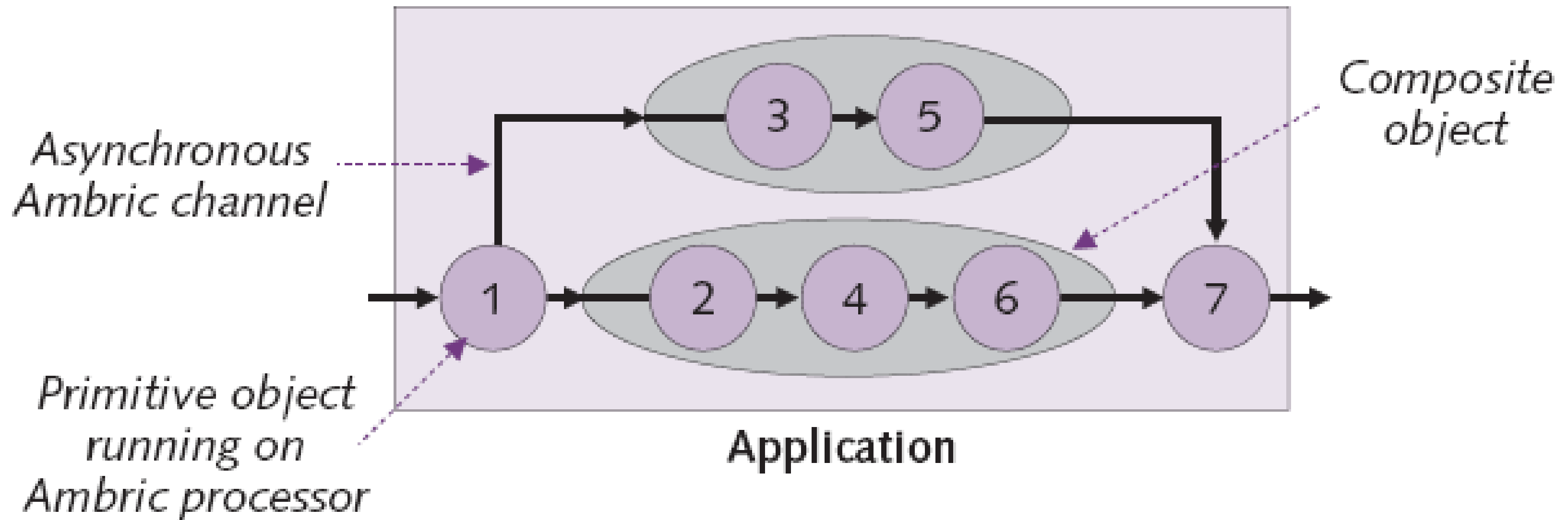
Philosophy and Principle

- 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.

- 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).**

Programming Model

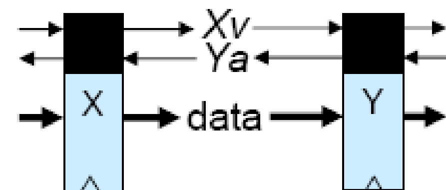


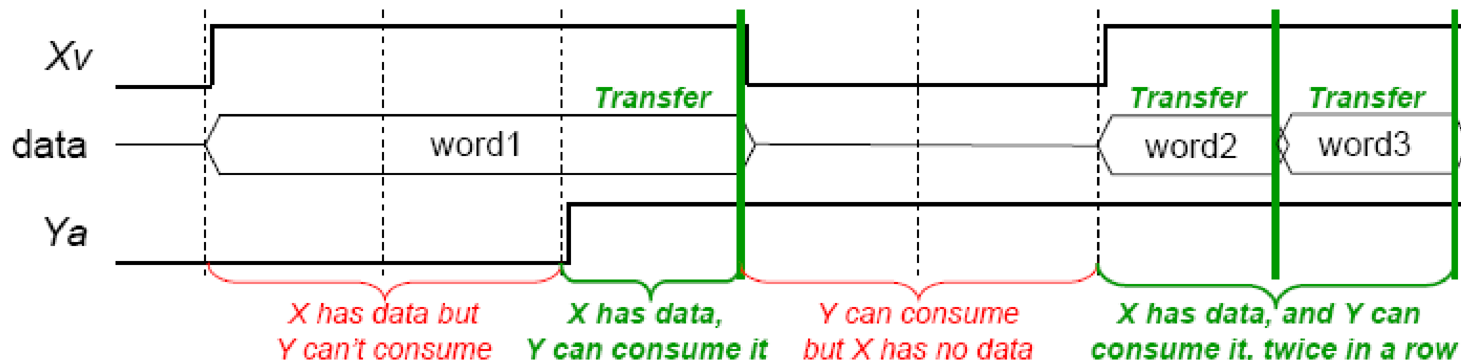
EXAMPLE
Two cores (numbers 1 and 7) are running relatively
simple software objects requiring only a single core. A more
complex object (a composite object) is running on two cores
(numbers 3 and 5). An even more complex composite object
is running on cores 2, 4, and 6. An on-chip network of channels
connects the cores together.

Synchronization

- **Data channels** are wholly responsible for synchronizing the processor cores.
- A processor located downstream must wait until it receives results from a processor located upstream.
- To the processors 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.

- Ambric register protocol:
  - output asserts *valid* when it has data
  - input asserts *accept* when it can consume





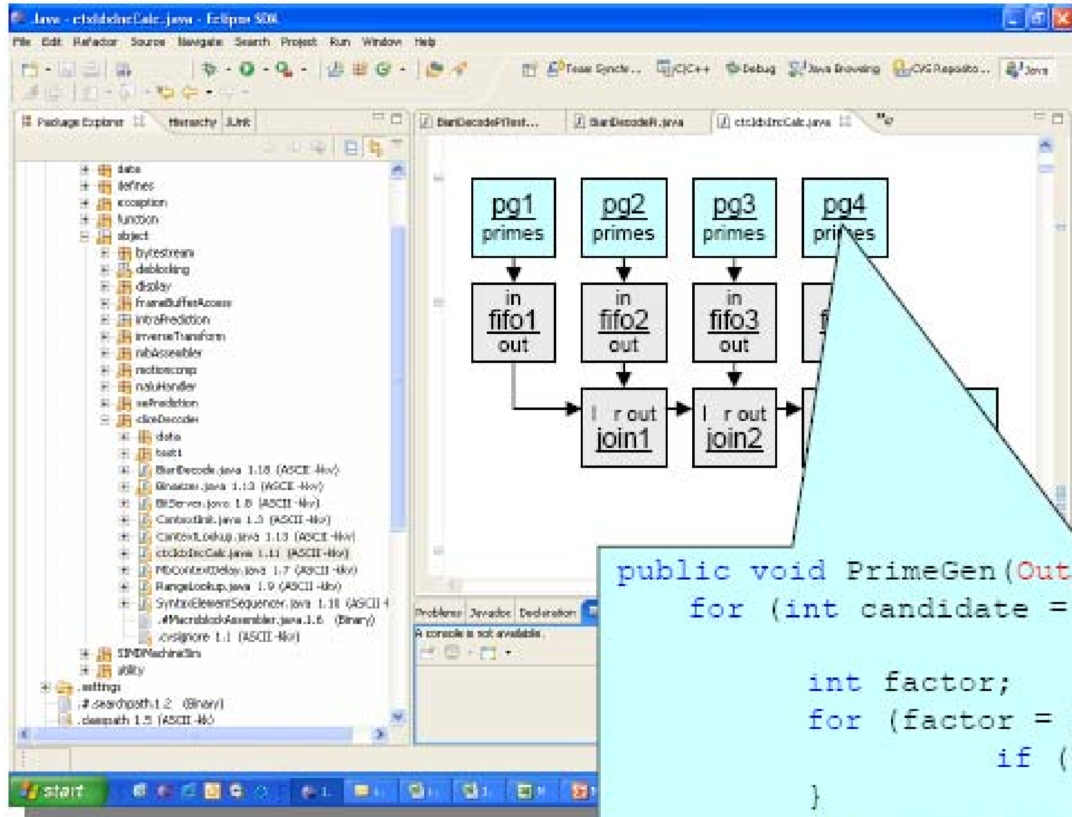
GALS

- 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.

Programming

- Source language for objects: **aJava**, a strict subset of Java.
- Ambric's tools statically compile the Java source code into the native machine language of the proprietary 32-bit RISC processors.
- To link the objects together in parallel structures via channels:
  - a **graphical interface** or
  - a textual language called **aStruct**.

Object Programming in standard language

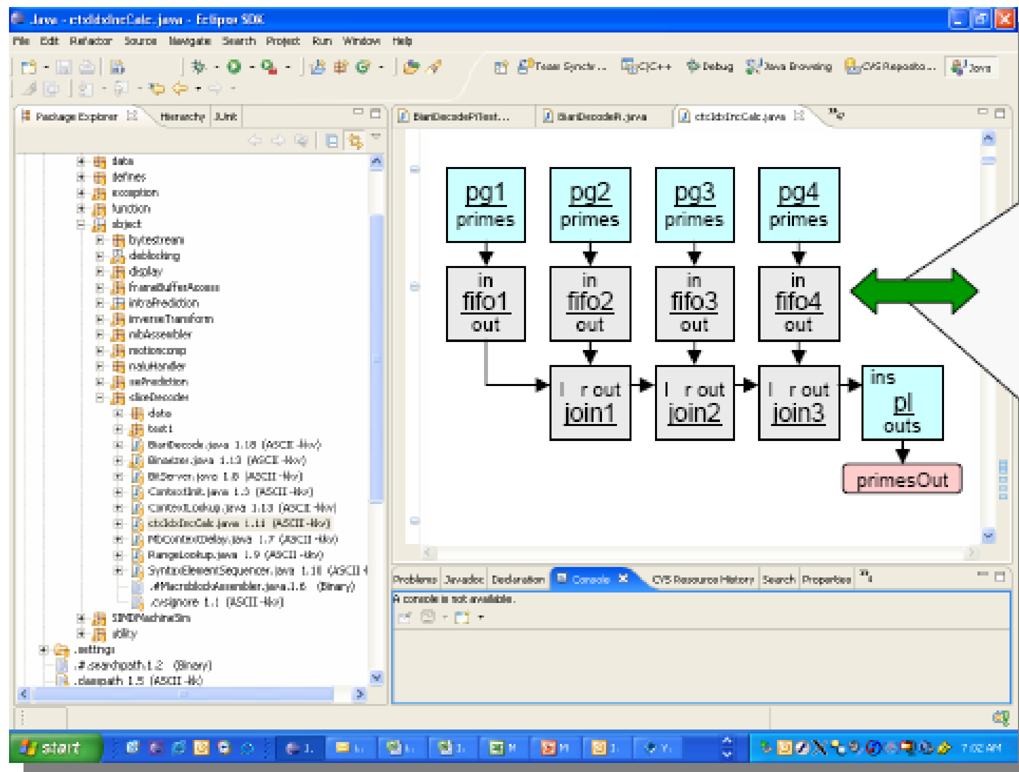


- **Program primitive objects in Java**
  - Strict subset of standard Java
    - static memory
  - Classes define the channels

```
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);
    }
}
```

- **Ambric is language-agnostic**
  - C, etc. to follow

Structural Programming in aStruct

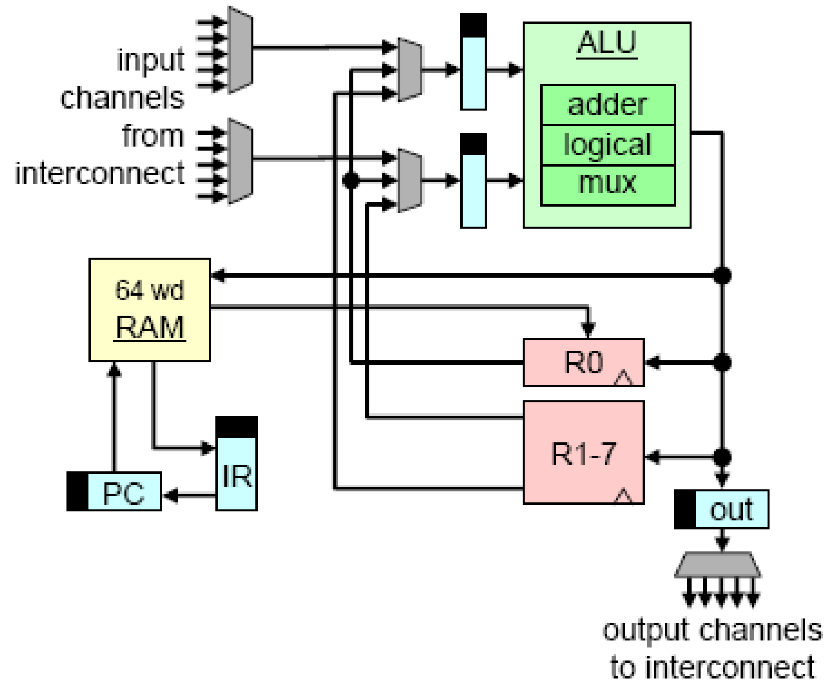



```
binding PrimeMakerImpl
      implements PrimeMaker {
  PrimeGen pg1 = {min = 3, increment =
     4, max = IPrimeMaker.max};
  PrimeGen pg2 = {min = 5, increment =
     4, max = IPrimeMaker.max};
  PrimeGen pg3 = {min = 7, increment =
     4, max = IPrimeMaker.max};
  PrimeGen pg4 = {min = 9, increment =
     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

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
  - 2 inputs, 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
  - ELIO
- 64 word local code/data RAM
  - 128 instructions
- Three-stage Ambric channel datapath

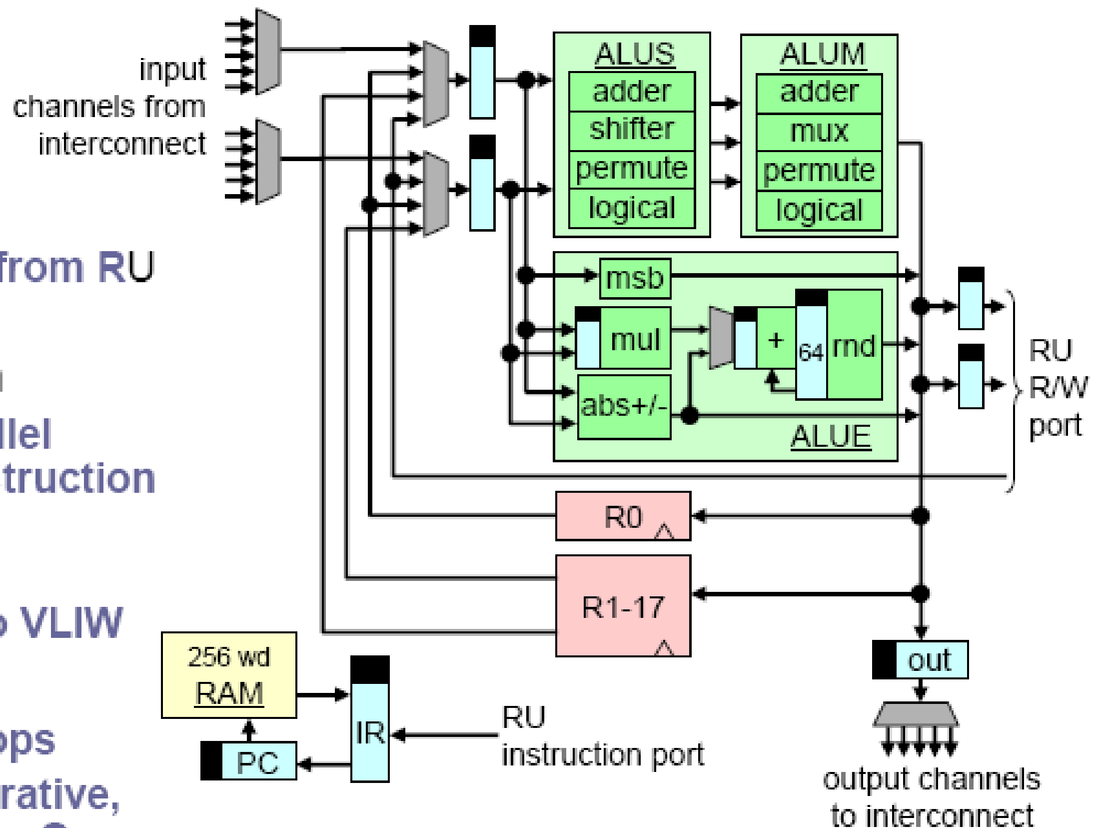


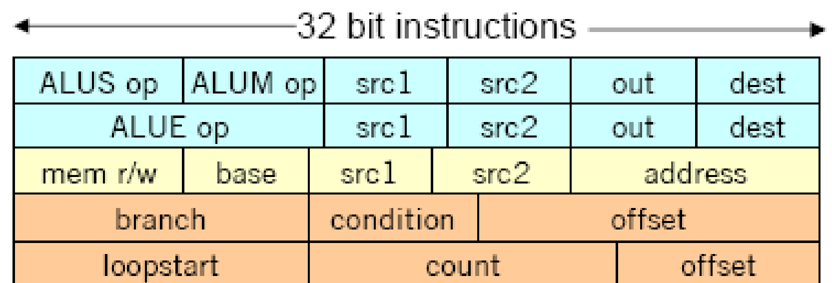
16 bit instructions

<table>
<tr><td>ALU op</td><td>src1</td><td>src2</td><td>out</td><td>dest</td></tr>
<tr><td>ALU op</td><td>src1</td><td>opcode</td><td>out</td><td>dest</td></tr>
<tr><td>mem r/w</td><td>src1</td><td>src2</td><td colspan="2">address</td></tr>
<tr><td colspan="2">branch</td><td>condition</td><td colspan="2">offset</td></tr>
<tr><td colspan="2">loopstart</td><td>count</td><td colspan="2">offset</td></tr>
</table>

Ambric SRD Processor

- Streaming RISC with DSP extensions
- 32 bit instructions
  - Multi-op ELIO
  - 256 in local RAM, more from RU
- 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	out	dest
ALUE op		src1	src2	out	dest
mem r/w	base	src1	src2	address	
branch		condition	offset		
loopstart		count		offset	

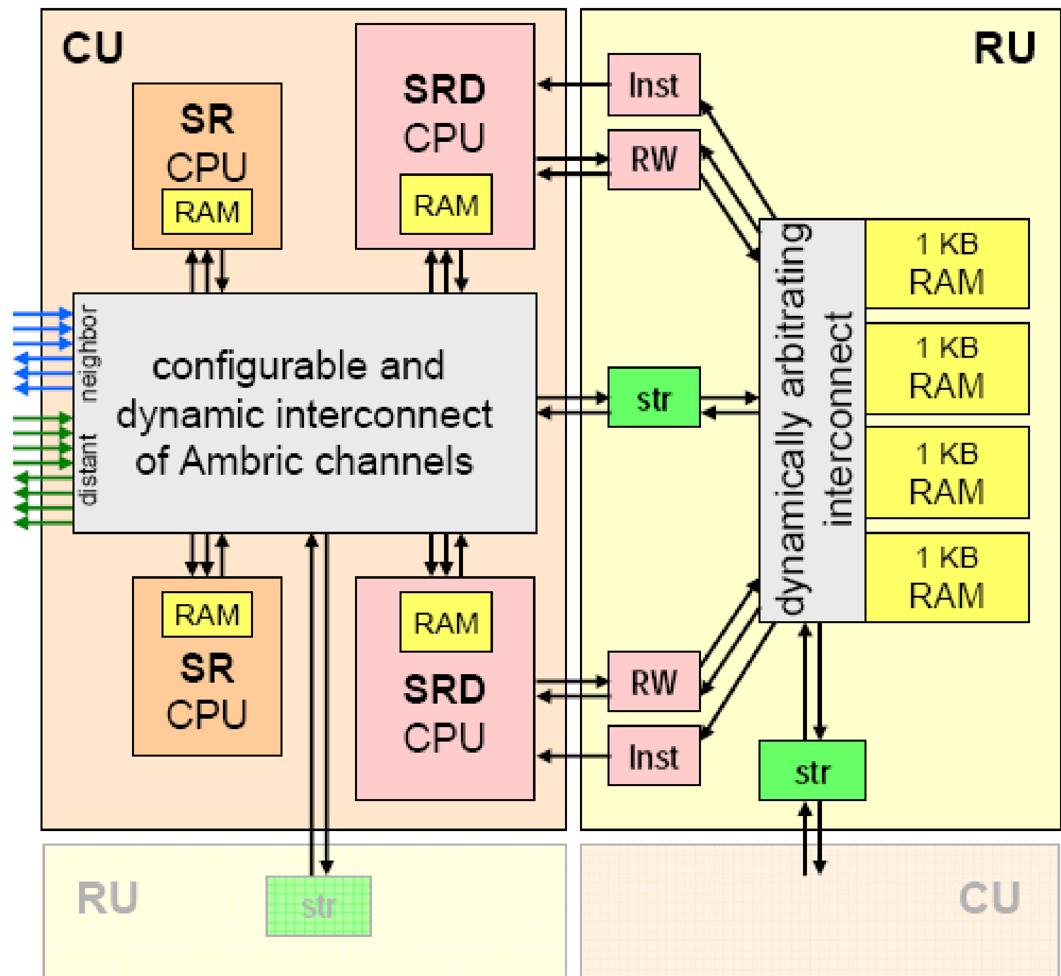
Ambric Compute Unit, RAM Unit

Compute Unit (CU)

- Two SRD 32-bit CPUs
- Two SR 32-bit CPUs
- Channel interconnect
  - **CPU-CPU is dynamic under instruction control**
  - **CU-Neighbor, CU-Distant are statically configured**

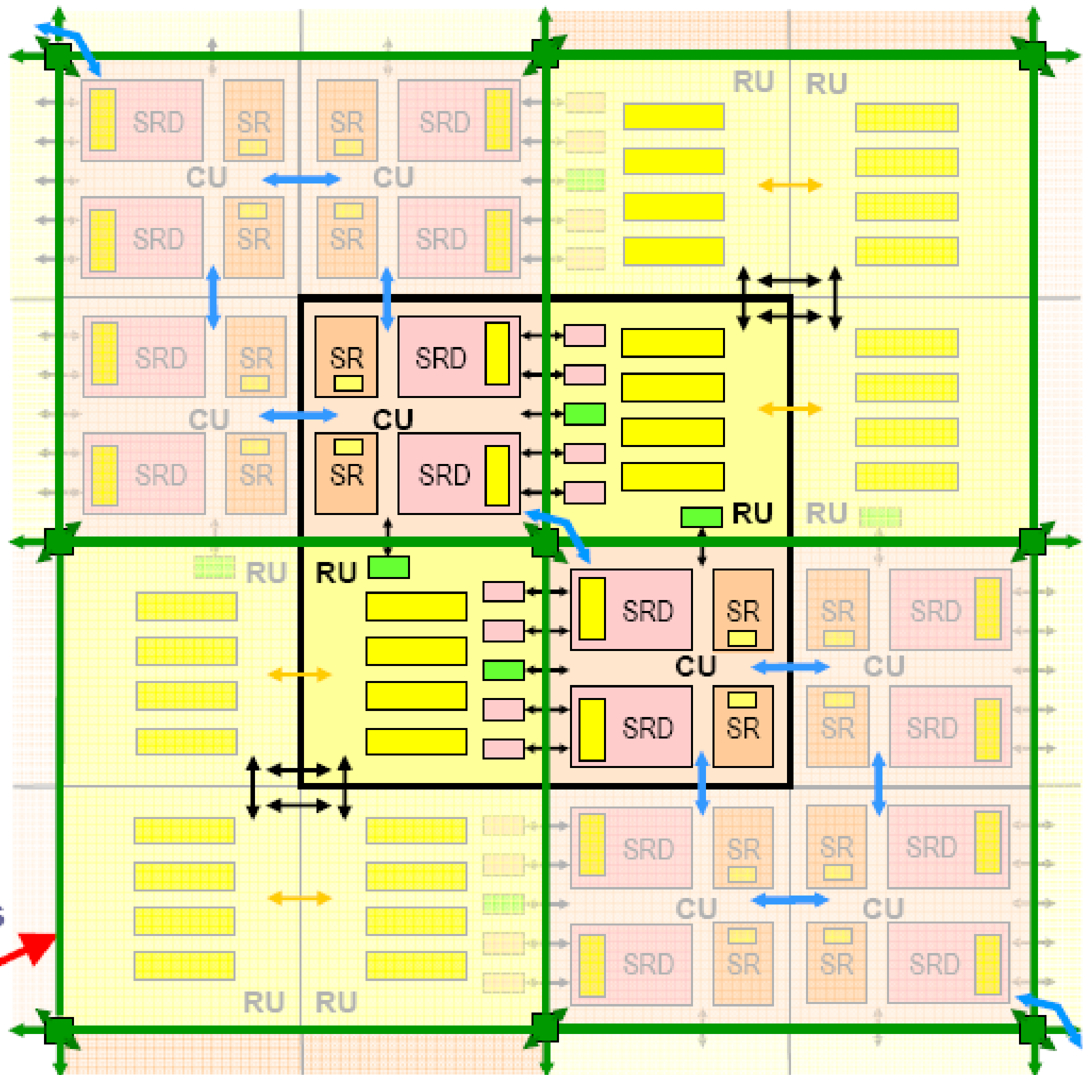
RAM Unit (RU)

- Four 1KB RAM banks
- RU engines turn RAM regions into channels
  - FIFO and random access
- Engines dynamically connect to banks through channels with arbitration

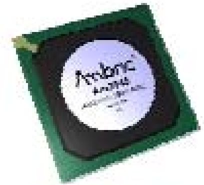


Ambric Brics and Interconnect

- The bric is the physical building-block
  - Two CU-RU pairs
  - 8 CPUs
  - 13KB RAM
- Brics connect by abutment to form a core array
  - CU quads, RU quads
- Neighbor CU channels
- Distant CU channels:
  - bric-length hops
  - configurable switches
- No wires longer than a bric



Am2045 Chip



- 130nm standard cell ASIC
  - 117 million transistors
- 45 brics, 1 teraOPS
  - 360 32-bit processors
  - 4.6 Mbits dist. SRAM
- High-bandwidth I/O
  - PCI Express
  - DDR2-400 x 2
  - 128 bits GPIO, 100 MHz
  - Serial flash, JTAG, µP
- BGA-868 package
  - Thermally enhanced

µP I/F

PCI Express

GPIO

JTAG

Flash

5 x 9 array of brics

DDR2

GPIO

GPIO

GPIO

GPIO

Ambric

 Copyright © 2003-2006 Ambric, Inc.

Simple Example: Prime Numbers

- Issue an ordered list of prime numbers
- For each odd number *p*, divide by each odd *i*, $3 \leq i \leq p$, until $p \bmod i = 0$. If $p = i$, *p* is prime. If not, it's not.
- **A dumb way to find primes, but...**
  A challenge for parallel execution:
  - Very irregular execution time ***(test 47 = 22 steps, test 49 = 3 steps)***
  - Must synchronize to keep prime list in order
  - Must buffer to keep busy
- This will show how an easily programmed asynchronous system of Ambric objects and channels self-synchronizes and stays busy

Ordinary sequential C code

```
main() {
  const long N=100000;
  long primes[N], number, i, number_of_primes=0;
  printf("Determining primes from 1-%d \n", N);
  primes[ number_of_primes++ ] = 2; // the even prime

  for( number = 3; number <= N; number += 2 ) {
    long factor = 3;
    while( number % factor ) factor += 2;
    if( factor == number )
      primes[number_of_primes++] = number;
  }
  printf ("Found %d primes\n", number_of_primes);
}
```