2007 International Symposium on Field-Programmable Custom Computing Machines

A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing

Michael Butts, Anthony Mark Jones, Paul Wasson
Ambric, Inc., Beaverton, Oregon
mike@ambric.com

Abstract

A new platform for reconfigurable computing has an object-based programming model, with architecture, silicon and tools designed to faithfully realize this model. The platform is aimed at application developers using software languages and methodologies. Its objectives are massive performance, long-term scalability, and easy development.

In our structural object programming model, objects are strictly encapsulated software programs running concurrently on an asynchronous array of processors and memories. They exchange data and control through a structure of self-synchronizing asynchronous channels. Objects are combined hierarchically to create new objects, connected through the common channel interface.

The first chip is a 130nm ASIC with 360 32-bit processors, 360 1KB RAM banks with access engines, and a configurable word-wide channel interconnect. Applications written in Java and block diagrams compile in one minute. Sub-millisecond runtime reconfiguration is inherent.

1. Motivation

Systems of all sizes, from pocket to desktop to rack-mount, demand increasingly high performance, with increasingly complex and flexible function, at acceptable hardware and development cost, within strict power constraints. Limitations of individual processors, DSPs and ASICs are well known to reconfigurable computing researchers.

FPGAs have been the dominant platform for reconfigurable computing since the first FCCM. A persistent and fundamental problem with computing on FPGAs is the need to design hardware to program the application. The reconfigurable computing developer must be an application expert, a software engineer, and a hardware engineer. The intersection of those three sets of people is very small. Training software engineers to be hardware engineers, writing in hardware description languages (i.e. VHDL, Verilog, Handel-C), mindful of registers, clock domains, timing closure, area and speed costs, has been too heavy a burden to be accepted.

Even in hardware engineering, RTL productivity has scaled at a much slower rate than Moore's Law, such that development cost is now the limiting factor, not available gates [19]. RTL's difficulty and poor scalability make it unacceptable for reconfigurable computing [3].

Instead, ever since that first FCCM, extensive research has been conducted to find ways to automatically translate C software into hardware designs [9]. In spite of first-class work, and a number of commercial offerings past and present, these tools still require hardware-specific pragmas or other input from a hardware engineer [8]. VHDL is still used in many FCCM application papers in recent years.

The heart of this problem is the programming model. Software languages such as C, C++ and Java are successful because they directly embody the von Neumann model of doing a sequence of operations, one at a time, on data in one memory space. The whole point of hardware, programmable or otherwise, is to use many operations operating in parallel on data in many parallel memories. Translating from one domain into the other remains a professional engineering job. Automating it, with high enough quality results to be effective, amounts to artificial intelligence.

Rather than attempting to translate software into hardware, we can now afford to run hundreds of instances of software in a reconfigurable fabric on one chip. A 32-bit processor/memory/interconnect core takes the same silicon area in today's 130nm process as a 4-LUT/flip-flop/interconnect tile took in the 2-micron process used for the first generation FPGAs (0.5 mm^2) [2][21]. Just as LUTs were the best building blocks for the 1980's FPGA, processors are the best building blocks today. As RISC pioneer Prof. David Patterson said, "Processors are the transistors of tomorrow." [18]

Most reconfigurable computing research to date has chosen a hardware platform first, the FPGA, and adapted some programming model to it second. To offer a better platform, we chose a programming model for reconfigurable computing first, and developed hardware architectures and chip implementations to suit that model.

2. Related Work

U. C. Berkeley's SCORE project [4], and others, have demonstrated the value of stream computing, with a clear model of computation, a graph of compute nodes with streaming data flow, and memory accessed through streams. MIT's RAW [20] is an array of processors, but with a more conventional programming model that includes cached and shared memory space, and depends on advanced compiler technology to derive parallelism from conventional code.

Current projects, such as TMD at U. Toronto [17] and BEE2 at U. C. Berkeley [5], have developed multiprocessors in FPGAs. It's a shame to waste silicon by a factor of fifty [22] to program CPUs into FPGAs, but programmable interconnect makes a distributed programming model possible.

3. Structural Object Programming Model

An array of 32-bit RISC processors and memories (Figure 1) is programmed with conventional sequential code. A programmed processor or memory is called a primitive object. Objects run independently at their own speeds. They are strictly encapsulated, execute with no side effects on one other, and have no implicitly shared memory.

Objects intercommunicate through channels in hardware. Each channel is word-wide, unidirectional, point-to-point, strictly ordered, and acts like a FIFO-buffer. Channels carry both data and control tokens, in simple or structured messages. Channel hardware synchronizes its objects at each end, dynamically as needed at run time, not compile time.

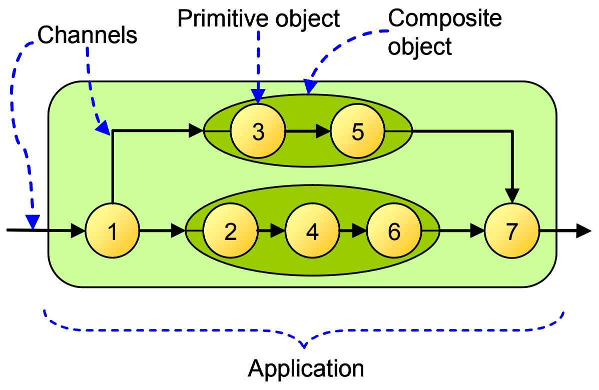


Figure 1: Structural object programming model

Inter-processor communication and synchronization are combined in these channels. The transfer of a word on a channel is also a synchronization event. Processors, memories and channel hardware handle their own synchronization transparently, dynamically and locally, relieving the developer and the tools from this difficult task.

Channels provide a common hardware-level interface for all objects. This makes it simple for objects to be assembled into higher-level composite objects. Since objects are encapsulated and only interact through channels, composite objects work the same way as primitive objects. Design reuse is practical and robust in this system.

Programming is a combination of familiar techniques. The application developer expresses object-level parallelism with block diagrams, like the ordinary "whiteboard" architecture diagrams normally used to architect an application. First a hierarchical structure of composite and primitive objects is defined, connected by channels that carry structured data and control messages. Then ordinary sequential software is written to implement the primitive objects that aren't already available in a library.

3.1. Model of Computation

This programming model has some similarities to, and was somewhat inspired by, Communicating Sequential Processes (CSP) [10]. Like CSP, it has processes running in parallel, all communication is by message passing through FIFO channels, and there is no shared memory. But CSP channels are rigidly synchronous. The sender stalls until the receiver is ready and vice versa. Statically implemented in a hardware fabric, it would have a heavy performance cost.

Our model of computation is closest to a Kahn Process Network [7][12], which has buffered channels. In its pure form the buffering is infinite, so writes into channels never stall. Real hardware does not permit that, so this model has bounded buffers, in which writes stall when buffers are full.

4. Hardware Architecture

The hardware architecture objectives are:

- Realize the Structural Object Programming Model as faithfully and effectively as possible.
- Maximize silicon area efficiency, since the model enables practically unlimited numbers of processors, memories and channels. Find the essential elements and keep them generic and economical.
- Take maximum advantage of the model's scalability, in communications, clocking and power efficiency.

4.1. Registers

At the heart of the hardware that embodies this programming model is the Ambric register. The architecture and implementation are based on its capabilities. Ambric registers (Figure 2) replace ordinary edge-triggered registers everywhere in the system. (In the rest of this paper, the word register refers to an Ambric register.)

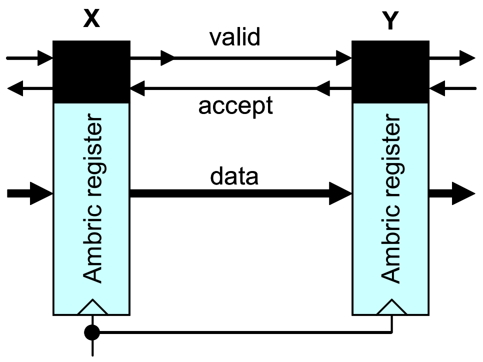


Figure 2: Ambric registers

It's still a clocked register with data in and data out. Alongside data are two control signals, *valid* and *accept*, that implement a hardware protocol for local forward and backpressure. This protocol makes systems using Ambric registers self-synchronizing and asynchronous.

Each Ambric register is preemptive, in that its operation is self-initiated. When a register can accept an input, it asserts its *accept* signal upstream; when it has output available, it asserts *valid* downstream. When they both see *valid* and *accept* are both true, they independently know the transfer has occurred, without negotiation or acknowledgement.

This protocol is simple and glueless, using only two signals and no intermediate logic. Unlike FIFOs, these dynamics are self-contained, so they become hidden and abstract.

Scalability calls for all signals to be local, one stage long. When the output of a register chain stalls, the stall can only propagate back one stage per cycle. To handle this, all Ambric registers can hold two words, one for the stalled output, and one caught when necessary from the previous stage.

4.2. Channels

A chain of registers like Figure 2 is called an Ambric channel (or just a channel). Channels are the fully encapsulated, fully scalable technology for passing control and data between objects, called for by the programming model.

Local synchronization at each stage lets channels dynamically accommodate varying delays and workloads on their own. Changing the length of a channel does not affect functionality, only latency changes.

4.3. Primitive Hardware Objects

The general form of a primitive hardware object is combinational logic, state machines and/or RAMs, interconnected by Ambric registers, with only channel interfaces to the outside, as illustrated by Figure 3. Logic blocks use *valid* and *accept* signals to receive inputs and issue outputs, synchronizing internally, and with the outside world, through these transfer events. Specific examples include processor objects and memory objects.

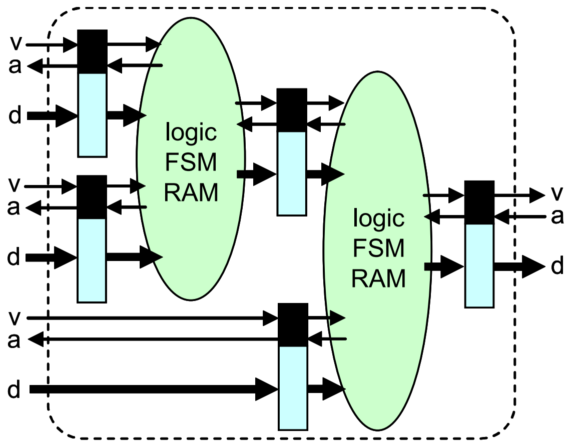


Figure 3: Primitive hardware object with channel interfaces

As called for by the programming model, primitive hardware objects are strictly encapsulated through channel interfaces, execute with no side effects on one other, and have no implicitly shared memory. They are programmed and connected through channels to form composite objects and ultimately complete applications.

Clock-crossing registers, that work with different clocks on input and output, are made possible by the register protocol. Objects connected through clock-crossing registers can run at their own speeds. Their circuit-level implementation is kept synchronous, avoiding metastability, by deriving both clocks from a common fastest clock reference.

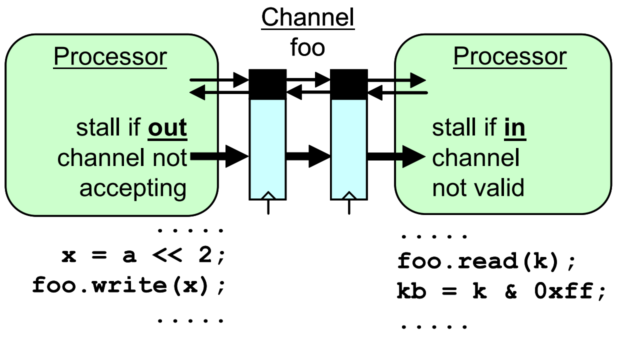
Objects can dynamically self-adjust their clock rates over a wide range entirely in software. Applications save power by running each processor at the lowest rate possible.

4.4. TMS packets

A standard packet format called The Message System (TMS) is used by hardware and object software for read and write memory requests and responses, commands, and other purposes. Memory access engines, external memory interfaces and I/O devices are hardwired for TMS packets. Among other uses, TMS packets can randomly access a memory object at any distance through channels.

4.5. Processors on channels

Ambric processors are instances of primitive hardware objects. They are interconnected through channels, as shown in Figure 4. When programmed with object code, they are synchronized with one another by channel transfer events.


Channel
foo
Processor
stall if out channel not accepting
Processor
stall if in channel not valid

```
        .....
   x = a << 2;
 foo.write(x);
        .....
```

```
.....
foo.read(k);
kb = k & 0xff;
.....
```

Figure 4: Processors joined by an Ambric channel

When an object tries to output a word to a channel that cannot consume it, the processor stalls until it can. Likewise, when an object needs a word of input from a channel that does not have one available, it stalls until one is. The small amount of buffering in an Ambric register lets both processors keep running, so they only stall according to the control and data dynamics of the application.

4.6. Processor Architecture

Traditional von Neumann architecture processors read and write variables in a memory space, implemented in a register-memory hierarchy. Efficient handling of streaming data through channels is not inherent in this architecture.

The objective of Ambric processor architecture (Figure 5) is processing data and control from channels. Memories are encapsulated objects like any others, which are read and written using channels. Instruction streams arrive through channels. Thus channel communication is a first-class feature of the architecture.

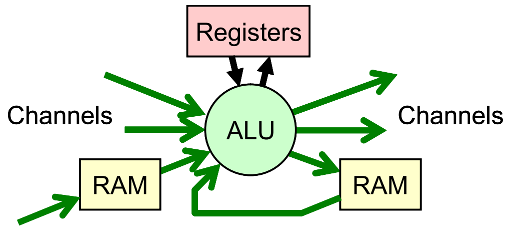


Figure 5: Processor architecture

This makes for very lightweight 32-bit streaming RISC CPUs. Channels are accessed the same way as general registers. Every datapath is a self-synchronizing Ambric channel. Data and control tokens keep moving, so RAMs get used more for buffering, rather than as a static global workspace.

Channels are first-class members of the instruction set architecture. Each source and destination instruction field chooses from general registers and multiple channels.

With hundreds of processors on a chip, code density and simple implementation again become essential. Instruction sets must be bit efficient, have simple and fast implementations, and take advantage of instruction-level parallelism.

Processor datapaths are channels, whose natural self-synchronization makes pipeline control easy. Their "general registers" are not Ambric registers. They are conventional registers, essentially memory locations, since they may be read many times, and overwritten anytime.

ALU instructions can execute one or more operations, do a loop iteration, input from a channel and output to a channel in a single cycle. A one instruction loop has performance equivalent to a configurable hardware block, as in an FPGA. But unlike FPGAs, the full range of performance versus complexity between hardware and software representations is available, with software programming.

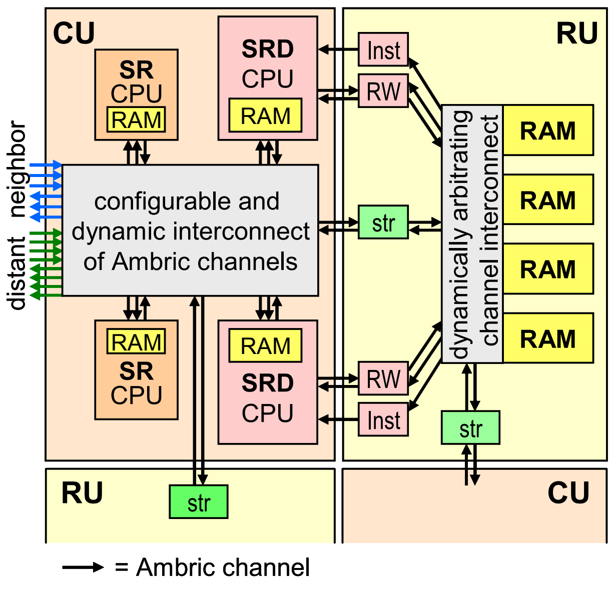


Figure 6: Compute Unit - RAM Unit Pair

4.7. Channel Execution

Processors can execute instructions from their local memories in the conventional way. They can be also switched into channel execution mode, which dispenses with the program counter and local memory, instead consuming an instruction stream from any input channel.

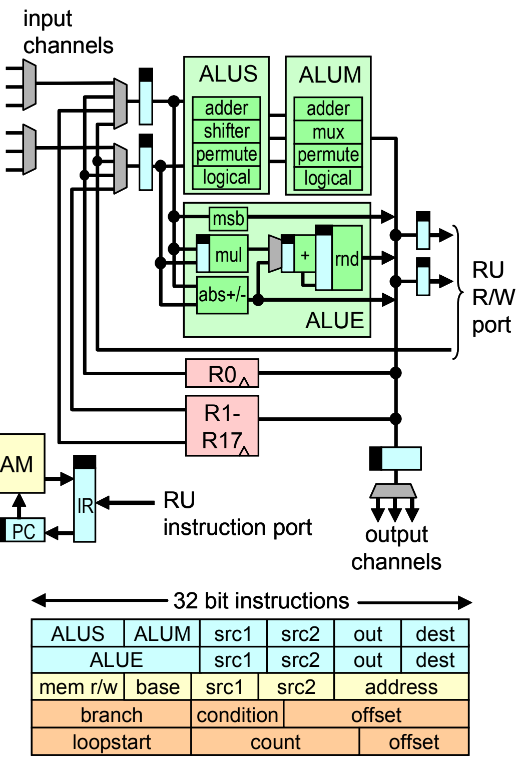
When each instruction completes, it accepts the next one from the channel. Conditional forward branches accept and skip instructions, and loops are constrained to one instruction. Multiplexer instructions can select between two inputs, without disrupting the instruction stream as branches do.

Channel execution is the purely streaming convergence of code and data. Many execution patterns are possible:

- A tree of channels feeding the same instruction stream to many processors is a simple and memory-efficient way to do SIMD execution in this MIMD architecture.
- A number of instruction streams may be fed to an object. As it executes one, the others stall unaccepted, until the code changes channels, analogous to a jump.
- An instruction stream can load a multi-instruction loop body into its local memory, switch to memory execution to execute the loop, and return to channel execution on loop exit, discarding the code in memory.
- A channel-executing client object can send requests to its instruction source object, according to runtime conditions. The instruction source object implements conditional branching, subroutines, looping, caching, or any other execution patterns for its client, according to how it is written. For example, a caching object responds to client requests for different code blocks, delivering code it caches on-chip or reads from off-chip.

4.8. Compute Unit

A cluster of four Ambric processors, two SRs and two SRDs, is a Compute Unit (CU), shown in Figure 6, paired with a RAM Unit (RU) with configurable access engines.



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

Figure 7: SRD processor

SRD is a 32-bit streaming RISC processor with DSP extensions for math-intensive processing and larger, more complex objects (Figure 7). It has three ALUs, two in series and a parallel third, with individual instruction fields. This captures instruction-level parallelism, with superior code density to VLIW. Its ALUs include full-word, dual half-word and quad byte logical, integer and fixed-point operations, including multiply and barrel shift.

It has 256 words of local memory for its 32-bit instructions, and can directly execute more up to 1K more from the RAM Unit next to it, through its dedicated RU instruction engine. SRD has streaming and random access to that RU's memory through its dedicated RU read/write engine.

SR is a simple 32-bit streaming RISC CPU, essentially a subset of the SRD, used mainly for small and/or fast tasks, such as forking and joining channels, generating complex address streams, and other utilities. It has a single integer ALU. 64 words of local memory hold up to 128 16-bit instructions and data. SR has no dedicated RU connections.

The CU interconnect of Ambric channels (Figure 8) joins two SRDs and two SRs with one another, and with CU input and output channels, which connect directly with neighbor CUs, and with the distant channel network. These interconnects establish channels between processors and memories according to the application's structure.

Each SR and SRD processor has an input crossbar to feed its two input channels, which is dynamically controlled by instruction fields. Each input crossbar can receive every CU input, and the processor outputs. Hardware forks (not shown) are available to split any CU input channel between two processors. Input crossbars are partially populated.

A statically-configured output crossbar connects processor outputs and CU inputs to CU outputs. The output crossbar is fully populated for processor outputs, and partially populated for CU inputs.

The CU interconnect accepts 12 input channels and drives 12 output channels: 2 neighbor channels each way with each of three CUs, 4 each way with a distant channel switch, and 2 each way with RU streaming engines in different RUs, giving direct access to two sets of memory banks.

4.9. RAM Unit

The RAM Units are the main on-chip memory facilities. Each RU has four independent single-port RAM banks (Figure 6), each with 256 words. It has six configurable engines that turn RAM regions into objects that stream addresses and data over channels: two for SRD data by random access or FIFOs, two for SRD instructions by random access or FIFOs, and two for channel-connected FIFOs, or random access over channels using TMS packets.

RAM banks are accessed by the engines on demand, through a dynamic, arbitrating channel interconnect. To get one word per cycle FIFO bandwidth from single-port RAMs, the engines can be configured to use two banks, striping even addresses to one and odds to the other. When the reader and writer collide on the first bank, one wins arbitration, say the writer, and the reader stalls a cycle. Next cycle the writer uses the second bank, while the reader gets the first. Channel self-synchronization has naturally organized their accesses into full bandwidth for each. Engines can interconnect through RU neighbor channels to form multiple RU FIFOs.

4.10. Brics and Interconnect

A bric is the physical layout building-block that is replicated to make a core. Each bric (Figure 9) has two CUs and two RUs, totaling 8 CPUs and 13 KBytes of SRAM.

The chip interconnect is a configurable three-level hierarchy of channels. At the base of the hierarchy are the local channels inside each CU, shown in Figure 8. At the next level, neighbor channels directly connect CUs with nearby CUs and RUs with nearby RUs shown in Figure 9, left.

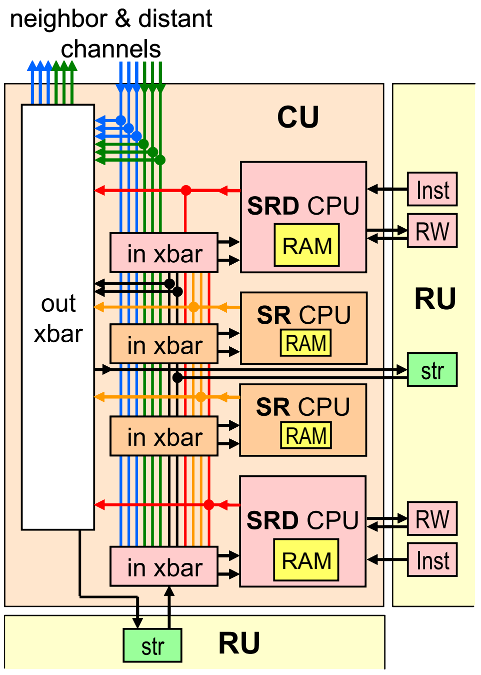


Figure 8: Compute Unit interconnect

At the top level, the network of distant channels for long connections is a 2-D circuit-switched interconnect, shown in Figure 9, right. Each hop of this interconnect is one bric long, carrying four channels in each direction, connecting through a statically reconfigurable registered switch in each bric. The switch also connects with four Compute Unit channels each way, to reach processors and memory objects. These bric-long channels are the longest signals in the core, except for the low-power clock tree and the reset.

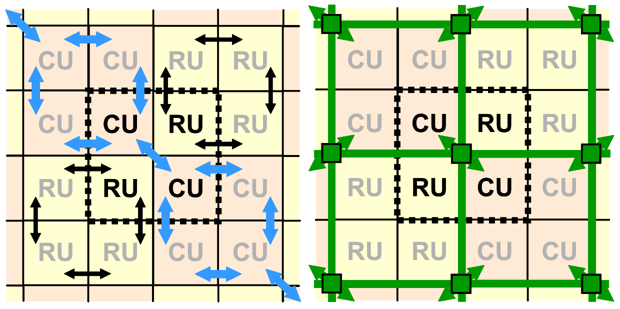


Figure 9: Core region centered on one bric, with neighbor channels (left) and the distant channel network (right)

The distant channel network always runs at the maximum clock rate, connecting to CUs through clock-crossing registers. Since distant channels cover a wide range of endpoints, but don't all need the highest bandwidth, they are virtual channels, multiplexed on a common physical channel.

Each switch has a physical Ambric register for every virtual channel. Between switches, each virtual channel has its own *valid* and *accept* wires, and the data wires are shared. When a virtual channel's *valid* and *accept* are both true, the data wires are switched to its physical registers at each end and the transfer completes. If more than one virtual channel can transfer they are fairly arbitrated. Virtual channels are multiplexed dynamically on demand at runtime. No compile-time analysis or user involvement is needed.

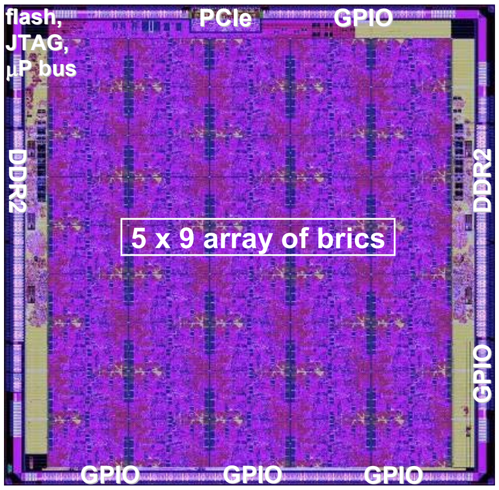


Figure 10: Am2045 chip

5. Chip Implementation

Am2045 is a standard-cell 130nm ASIC, with 117 million transistors, packaged in an 868-pin BGA. Its core has 45 brics in a five-by-nine array, containing 360 32-bit processors and 4.6 Mbits of distributed SRAM, with a 333 MHz clock. In contrast to FPGAs, its configurable interconnect takes less than 10% of the core area, leaving the rest for processors and memories. Pre-production silicon is functional.

The core bric array is surrounded by I/O interfaces (Figure 10) connected to neighbor and distant channels:

- Two DDR2-400 SDRAM interfaces, 32 bits each.
- 4-lane PCI Express, endpoint or root. Available at power up, supporting initial chip configuration from the host.
- 128 bits of parallel ports for chip-to-chip channels.
- Serial flash, microprocessor bus, JTAG interfaces.

6. Software

Since the hardware architecture directly implements the programming model, application development, compilation and debug tools are straightforward. They are much simpler, more reliable and easier to use than RTL synthesis and simulation, FPGA placement, routing, and timing analysis, or parallelizing and scheduling multiprocessor compilers.

All development and debug tools run in the Eclipse integrated development environment (IDE) [6], using its powerful facilities for source code display and analysis, live syntax checking, and background compilation during editing.

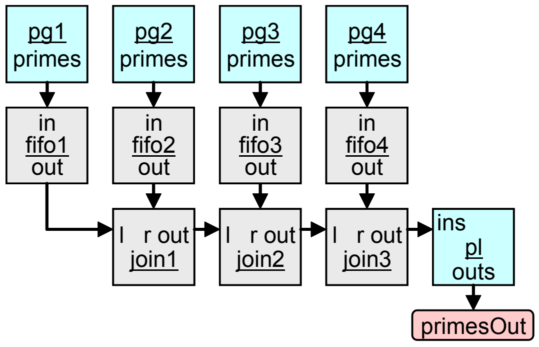


Figure 11: Graphical entry of composite object structure

6.1. Design entry

The developer can use a structural editor to graphically define objects, channels and how they connect (Figure 11). Graphical design is easy to learn and use, and provides direct visualization of the parallel structure. Object hierarchy can be explored in the graphical environment.

A textual coordination language [13] developed by Ambric, aStruct, is equivalent to the graphical form. Many developers prefer to use text once they're familiar with the tools. Figure 12 shows the composite object of Figure 11 in aStruct form, excerpted for space. aStruct also has constructs to support instance-specific structure generation.

```
binding PrimeMakerImpl
     implements PrimeMaker {
  PrimeGen pg1 = {min = 3, increment =
     4, max = IPrimeMaker.max};
  PrimeGen pg2 = {min = 5, increment =
     4, max = IPrimeMaker.max};
         ........
  Fifo fifo1 = {max_size = fifoSize};
         ........
  AltWordJoin join1;
         ........
  PrimeList pl;
  channel
    c0 = {pg1.primes, f1.in},
    c1 = {pg2.primes, f2.in},
         ........
    c11 = {pl.outs , primesOut};
}
```

Figure 12: Composite object structure in aStruct

The `Fifo` and `AltWordJoin` objects are from a library of common standard objects. Any primitive or composite object may be checked into a library for reuse. Since objects are strictly encapsulated, and interface through standard self-synchronizing channels, they can be thoroughly validated, and easily reused with confidence. Object reuse gives application development effort long-term scalability.

Primitive objects are written in an integer subset of standard Java, or SR/SRD assembly code. Java source may include assembly language intrinsics, and is compiled directly to the processor, without a JVM, dynamic memory, etc. Classes are defined to read and write channels. Figure 13 is the source code for the `PrimeGen` object, with its channel output in bold. Java is the first language implemented because it is naturally object-based, and it is well-supported by Eclipse. Other languages such as C will follow.

```
public void PrimeGen
          (OutputStream<Integer> primes) {
  for (int cand = min; cand <= max;
           cand += 2*inc) {
    int fac;
    for (fac = 3; fac <= max; fac += 2) {
        if (cand % fac == 0) break;
    }
    if (cand == fac) {// is prime
        primes.write(cand); // output
    }
    else primes.write(0);
  }
}
```

Figure 13: Primitive object written in Java

6.2. Compilation Tool Chain

The tool chain illustrated in Figure 14 compiles an application into a configuration stream for the chip. Eclipse compiles object structure and sources automatically in the background during development, so it's immediately ready for simulation or placement and routing.

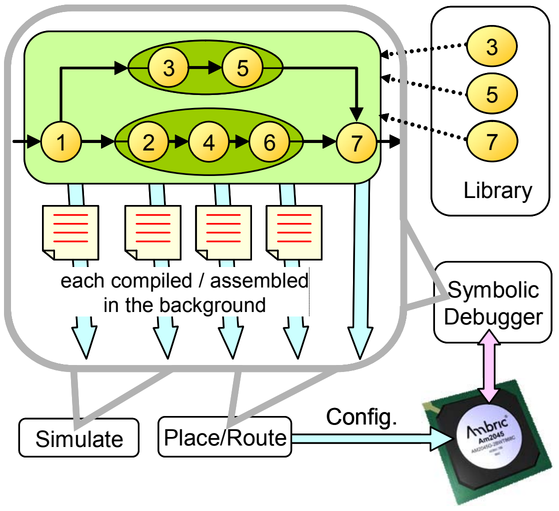


Figure 14: Tool chain

A behavioral simulator, aSim, can execute and debug applications on a workstation without the chip. Java is compiled (by the standard Java compiler) to the workstation, and SR/SRD instruction set simulators execute assembly code. aSim models parallel object execution and channel behavior.

A simulated annealing placer [1] maps primitive objects to resources on the target chip, and a PathFinder router [15] assigns the structures' channels to specific hardware channels in the interconnect. Channel self-synchronization means no interconnect scheduling is needed. Then it generates a configuration stream file.

Since objects are compiled, not synthesized, timing closure is a non-existent issue, and the number of objects and channels to be placed and routed is in the hundreds, not the tens of thousands like FPGAs, chip compilation is very fast. Most applications are compiled in less than one minute. The most densely packed cases still compile in less than five minutes. As with software development, the design/debug/edit/rerun cycle is nearly interactive.

6.3. Debugging and Performance Analysis

Many errors that can occur in hardware design or software design are not possible in this programming model. Hardware timing failures and state-machine/datapath synchronization errors are absent. Software objects that are encapsulated cannot damage each others' shared memory. Synchronization errors are less common with self-synchronizing channels than with spinlocks and semaphores. But any code has bugs, and any parallel system with distributed control and feedback could potentially deadlock.

Since the hardware directly implements the programming model, the programmer's source-code-level view is close to what is actually running, not obscured through parallelizing compilers or logic synthesis. Its internal state matches source code directly, and is visible and controllable.

A number of debugging techniques are available. Simulation in aSim exposes many errors quickly, in a fast, interactive environment with complete visibility.

Observation and analysis instrumentation objects may be written, and run on processors, memories and channels unused by the application. As with FPGAs, a family of chip sizes will be available, so a larger chip may be used for development, and a smaller chip deployed in products.

Channel traffic is easily tapped and traced, by inserting a duplicating fork object. Unlike adding debug code to an ordinary system, adding debug objects that run independently in parallel has a negligible effect on execution.

An integrated source level debugger uses the chip's debug network, a separate dedicated channel. An SR processor acts as debug network master on the debugger's behalf. Each CU and RU has a debug slave controller on the network, which can halt, step and restart objects running on processors, and observe or trap on channel events.

Since objects mutually self-synchronize through channels, the debugger can stop and inspect an object in a running application without crashing it. It just stalls on the object's channels, until it's started again.

7. Performance Metrics

At full speed, all 360 SR and SRD processors together are capable of 1.08 trillion operations per second. This is based on 8-bit and 16-bit sums of absolute differences, which are the kernel operations in video motion estimation. Interconnect bisection bandwidth is 425 gigabits per second.

A DSP benchmark was written to implement published TI benchmarks for FIR and IIR filters, dot product, maximum value and saturation, listed in Table 1. The quantity of kernels running on the chip at once is listed as well as the throughput of each. Each DSP kernel runs on a single SRD. A parallel form of the FIR filter is also listed, which is bigger and faster. The 130nm ASIC Am2045 at 333 MHz delivers 10 to 25 times greater throughput than the 90nm custom TI C641x DSP at 1 GHz. Its code is 1/3 the size and much less complex, without VLIW setup and teardown. Developing the five kernels for Am2045 took one engineer 1.5 days.

Kernel	Qty.	Speed	Notes
32-tap FIR	180	5.2 Msps	16 bit data
	5	247 Msps	
9-tap IIR	180	24 Msps	16 bit data
Dot Product	180	222 Msps	16 bits in, 32 bit sum
Max Value	180	381 Msps	n=100, 16 bits
Saturators	180	667 Msps	signed 16 to unsigned 8
Viterbi ACS	360	667 Msps	16 bit data
1K point FFT	90	9.8 Msps	complex 16-bit
AES	60	201 Mbps	feedback modes
	8	1.18 Gbps	non-feedback

Table 1: Am2045 kernel performance at 333 MHz

Additional kernels listed are the add-compare-select operation in Viterbi decoding, a 1024-point complex FFT, and AES encryption in feedback and non-feedback modes.

8. Application Examples

8.1. Motion Estimation

Motion Estimation is the most compute-intensive part of video compression. For every frame in real time, each 16-by-16 pixel macroblock is predicted by comparing with macroblocks in other frames, to find the best match and its motion vector [16]. The application does real-time motion estimation across two reference frames of broadcast-quality 720p HD video in real time.

The motion estimator for a single reference frame is shown in Figure 15. Two of these fit in one Am2045 chip. It does full exhaustive search, which is much more computationally intensive than commercial motion estimators.

RAMIF buffers incoming frames and candidate motion vectors in off-chip SDRAM and issues 16-by-16-pixel macroblocks. The search is done in parallel by a chain of identical Motion Estimation objects (*MEunit)*. Each handles part of the comparison region. A pair of channels link the chain, one sends the workload and the other collects results. Each *MEunit* is a composite object, consisting of 4 *Calc* objects that each process one macroblock at a time, and a block to collect and choose the best results. Each *Calc* has 1 memory and 4 processor primitives.

Both real-time motion estimators fit in 89% of the brics on the chip, and only need to run at 300 MHz. Its performance on Am2045 is 0.46 teraOPS, a large fraction of maximum available performance given its 160-way parallelism.

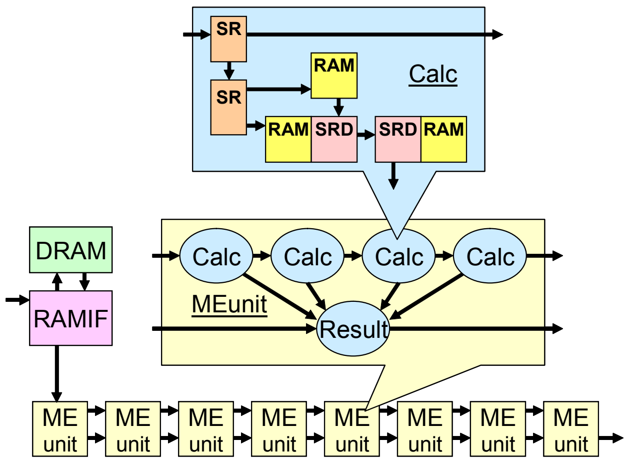


Figure 15: Motion estimation application

8.2. H.264 Deblocking Filter

All video compression leaves artifacts on pixel-block edges that must be filtered out, so decoders have deblocking filters. This has become complex in the H.264 standard [14]. The adaptive deblocking filter is in a closed decoding loop, so it must be bit-accurate according to the standard.

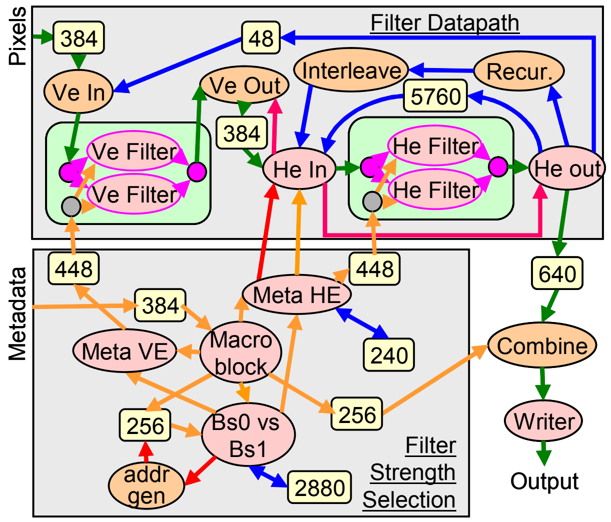


Figure 16: H.264 Deblocking Filter

Macroblocks, blocks, and pixels take different paths through different functions taking different lengths of time, under control of metadata that accompanies the pixel data. It's very hard to parallelize effectively in other architectures.

The structural implementation of a broadcast-quality HD H.264 deblocking filter is shown in Figure 16. It derives directly from the video architect's block diagram. It meets a complex standard that is irregular and dynamically data-dependent, with feedback-dependent behavior. Yet it is very parallel, making good use of composite and primitive objects self-synchronizing with one another through channels.

This is actually the structural program for the chip, all flattened down to primitive objects running on SRs, SRDs and FIFO memories. The architect's block diagram already captures the parallelism, so it is used as the parallel program. The top half is the filter datapath composite object, with pixels in and filtered pixels out; the bottom half processes metadata that adaptively controls the filter datapath.

This filter uses 7 CUs and 12 RUs, 11% of the Am2045 chip. For 1080i60 HD video, it processes 63M pixels/sec.

Conventional architectures, computing on a common memory, must read and write edge pixels horizontally and again vertically, requiring 188 MB/s of off-chip memory bandwidth. This implementation streams edge pixels from stage to stage through self-synchronizing channels and FIFO buffers, needing no off-chip memory bandwidth.

9. Configuration

Configuration is just the first program that runs on the chip after reset. Configuring the configurator is inherently recursive, based on building a chain of SR processors.

SRs, SRDs and RU memories and engines are configured by TMS packets sent through normal channels. Statically configured interconnects (the CU output crossbars and distant network switches) are directly configured by SRs, through bit-serial channels in each row of brics.

All SRs and SRDs come out of reset in channel execution mode. The first SRs in each row of brics receive a configuration stream, from off-chip through PCI Express, serial flash ROM, JTAG or the microprocessor bus. They execute the first part of the stream to configure interconnect for a chain of SRs, one per CU, through the entire core. Then the SR chain processes the rest of the stream recursively to configure the application.

A small four-SR chain in Figure 17 illustrates its operation. The configuration stream comes in through a channel fork (marked F) that routes packets alternately to the first SR's instruction input InX and data input In0.

It has a recursive structure, shown in Figure 18. The whole stream consists of three packets: Split code (S1), Data (D1), and Configuration code (C1). The Data packet contains the contents of the next level of Split, Data and Configuration packets, and so on. Each SR's Split code splits its Data packet into Split code, Data and Configuration code packets for the next SR in the chain.

At the end of the chain, the last SR runs its Configuration code (C4 in Figure 18), which contains the configuration payload for its CU and RU inline, encoded in load-immediate instructions. The payload starts with TMS packets, which the SR sends to an RU streaming engine (Figure 19) to configure SRDs and the RU. Packets load an SRD's local memory contents into RU memory, and feed a configuration program into the SRD through its RU instruction engine, which copies them in through its RU read/write engine. It finishes by stalling on a CU Lock bit for synchronization, after which it will execute from local memory, starting its object. The rest of the TMS packets load RU RAMs and configure its engines.

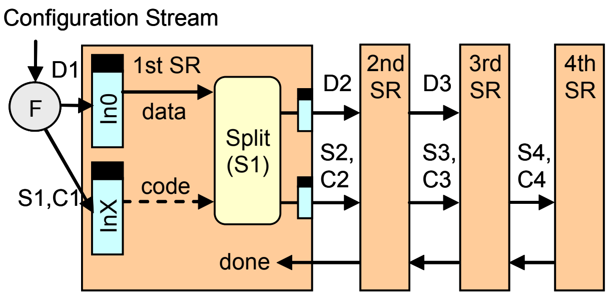


Figure 17: SR configuration chain

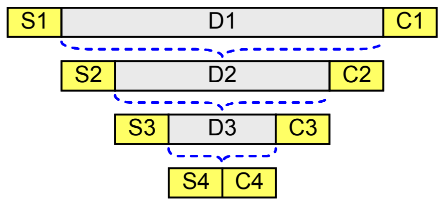


Figure 18: Recursively structured configuration stream

Then the chain SR feeds code and data from the payload into the other SR's local memory and starts it, and finally it does the same for itself. Before stalling on Lock, it sends a *done* token back up the chain. The next-to-last SR runs its configuration code and configures its CU and RU, passes the *done* token back and so on.

When *done* comes in, the first SR runs the rest of the stream, which configures static interconnects for the application's channels, and releases Locks, starting the application.

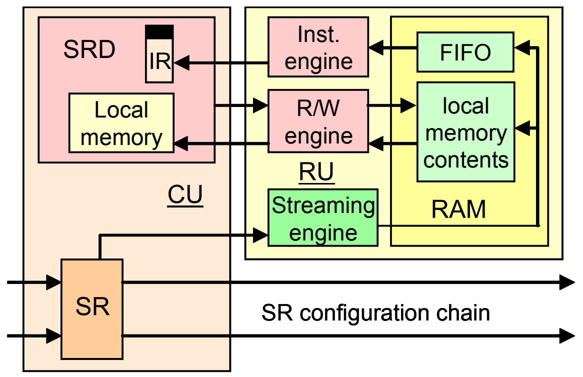


Figure 19: RU and SRD configuration from SR chain

A near-maximum application for Am2045's 45 brics, whose SR and SRD local memories are all full of object code and data, with little or no initial data in RU memories, has a 68Kword payload. 337K 16-bit load-immediate instructions (169K words) encode the payload inline. At 333 MHz the SR configuration chain can deliver this in 1 millisecond. Static interconnect for the chain and then the application, just 210 bits per bric, only takes 0.08 millisecond to configure. Most applications will not fill all processor local memories, so they will be smaller and faster.

A configuration chain can have a decompression object at its head. For example, a gzip-like decompressor (LZ77 and Huffman), which runs in one CU-RU pair, has delivered 3X compression of SR binary instructions. The example application loads from an 8Mbit 33 MHz flash chip in 165 msec., but compressed into a 2 Mbit chip it takes 55 msec.

9.1. Runtime reconfiguration

Since initial configuration is itself a configured application, reconfiguring parts of an application at runtime is straightforward. Given the bandwidth and compute power available, it can be very fast. Since objects are independent and encapsulated, reconfiguration can happen while other parts of an application continue to run normally.

A reconfigurable composite object (RCO) is a set of member composite objects, which all connect with and use the same set of input and output channels in a consistent way, and use a common region of CUs and RUs in the core.

An RCO also includes a persistent configurator object, which loads configuration streams from off-chip SDRAM. It constructs an SR configuration chain through the CUs in the RCO region, then deploys the stream down the chain. Chain SRs run stream-delivery programs to configure their CUs and RUs. Streams are delivered at full rate, using a packet-copy instruction that transfers one word per cycle.

Channels must be flushed and frozen during reconfiguration. CU channel registers have Flush controls (Figure 20) that empty them, and Holds that keep them empty. Flush, Hold and Lock are set before configuring a CU (except on channels used by configuration), and released to restart.

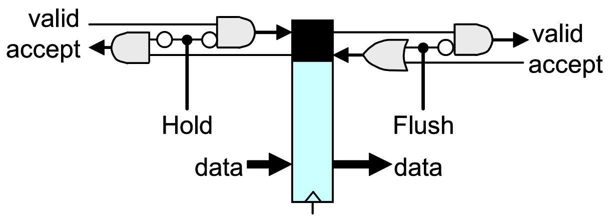


Figure 20: Ambric register with Flush and Hold controls

When an RCO shuts down before reconfiguration, its input and output channels stall. Hold and Flush keep those channels stalled during reconfiguration. Objects outside the RCO just stall on those channels, and continue normally when the newly reconfigured RCO starts running.

DDR2 SDRAM's 400Mword/sec. bandwidth meets the configuration chain's full 333 MHz rate. For example, a four-bric member object's CU local memories and RU configuration, 6K words, can be loaded in 18 microseconds. Four brics of static interconnect configuration, one for the chain and one for the member object, takes another 7, totalling 25 microseconds to load a new RCO member object.

10. Conclusions

We defined a programming model for reconfigurable computing, which is massively parallel and scalable, area and power efficient, and reasonable to use by software developers. We devised a hardware architecture, development languages and tools, and silicon implementation that faithfully embody the programming model. The result is a new reconfigurable computing platform which offers great potential for research and applications.

References

[1] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer, 1999.

[2] S. Brown, R. Francis, J. Rose, Z. Vranesic. *Field-Programmable Gate Arrays*, pp. 97-8. Kluwer, 1992.

[3] M. Butts, A. DeHon, S.C. Goldstein. Molecular Electronics: Devices, Systems and Tools for Gigagate, Gigabit Chips. *Proc. ICCAD 2002*, pp. 443-440.

[4] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). *Proc. FPL 2000*, pp. 605--614.

[5] C. Chang, J. Wawrzynek, R.W. Brodersen. BEE2: a high-end reconfigurable computing system. *IEEE Design & Test of Computers*, March-April 2005, pp. 114- 125

[6] Eclipse Foundation. Eclipse - an open development platform. http://www.eclipse.org.

[7] Mudit Goel. Ptolemy II Process Networks. http://ptolemy.berkeley.edu/ptolemyII/ptIIlatest/ptII/ptolemy/domains/pn/doc/main.htm. Jan. 2007.

[8] Richard Goering. C-to-RTL compiler touts full-chip design. *EE Times*, January 8, 2007.

[9] Maya Gokhale, Ron Minnich. FPGA Computing in Data Parallel C. *Proc. FCCM 1993*, pp. 94-101.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM* 21 (8), pp. 666–677, 1978.

[11] A. M. Jones, M. Butts. TeraOPS Hardware: A New Massively-Parallel MIMD Computing Fabric IC. *IEEE Hot Chips Symposium*, August 2006.

[12] G. Kahn. The semantics of a simple language for parallel programming. *Info. Proc.*, pp. 471-5, Aug. 1974.

[13] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33-42, May 2006.

[14] P. List, et. al. Adaptive Deblocking Filter. *IEEE Trans. Circ. Syst. Video Tech.*, pp. 614-9, July 2003.

[15] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. *Proc. FPGA 1995*, pp. 111-117.

[16] J. Ostermann, et. al. Video coding with H.264/AVC. *IEEE Circuits and Systems Magazine*, Vol. 4, No. 1., pp. 7-28.

[17] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, P. Chow. A Scalable FPGA-based Multiprocessor. *Proc. FCCM 2006*, pp. 111-120.

[18] David Patterson, et. al. RAMP: Research Accelerator for Multiple Processors. *IEEE Hot Chips Symposium*, 2006.

[19] G. Smith. The Crisis of Complexity. *Dataquest*, 2003.

[20] E. Waingold, et. al. Baring it all to Software: Raw Machines. *IEEE Computer*, Sept. 1997, pp. 86-93.

[21] Xilinx, Inc. Celebrating 20 Years of Innovation. *Xcell Journal,* Issue 48, 2004.

[22] P. Zuchowski, et. al. A Hybrid ASIC and FPGA Architecture., *Proc. ICCAD 2002*, pp. 187-194