# NanoMesh: An Asynchronous Kilo-Core System-on-Chip

Jonathan Tse
Computer Systems Laboratory
Cornell University
Ithaca, NY, U.S.A.
jon@csl.cornell.edu

Andrew Lines
Switch and Router Divison
Intel Corporation
Calabasas, CA, U.S.A.
andrew.lines@intel.com

***Abstract*—Innovative asynchronous circuits are central to the Ethernet switch chips from Intel's Switch and Router Division (formerly Fulcrum Microsystems). These circuits are complex, and it can be hard to gauge their benefits since there are few direct comparisons. For this paper, we apply the technology and tool flow developed for these commercial products to a familiar benchmark: a network of general purpose processors on a chip. The processor is a single-issue 32-bit integer RISC core, a from-scratch implementation mostly compatible with the MIPS R3000. The network uses a 16-port 32-bit fully connected Nexus crossbar. We achieve greater scalability by linking these crossbars in a 2D mesh with clusters of 8 cores and 4 cardinal and 4 diagonal links per tile. Each core has 64KB of local memory and can access the memory of any other core in the mesh. Our design makes heavy use of the Proteus synthesis, place & route flow, as well as existing custom cells. It required only a few man-months of effort to develop a complete gate-level design and physical floorplan which can run simple C programs such as Dhrystone. A few more man-months will produce a test chip, expected in 2013.**

***Index Terms*—asynchronous; many-core; mesh network; System-on-Chip; SoC; Network-on-Chip; NoC; MIPS; MIPS R3000; RISC;**

## I. Introduction

The increased transistor density available to designers through Moore's law scaling has sparked an industry-wide trend toward System-on-Chip (SoC) integration of a few dozen general purpose cores, or hundreds of specialized compute engines. Designing large chips with a single global clock has become difficult, especially in heterogeneous systems. The traditional solution to the clock problem has been Globally Asynchronous, Locally Synchronous (GALS) architectures, which simplify clock-domain crossings, timing closure, and system level design [1]. However, moving to a fully asynchronous systems can further simplify the problem of clocking by eliminating it altogether, reducing design complexity and adding easy modularity to the design [2].

A number of asynchronous interconnect designs have been proposed to address these timing issues as well as simplify the task of connecting heterogenous IP: Nexus [3], CHAIN [4], FAUST [5], QNoC [6], and DSPIN [7], among others. The simplest of these designs is Nexus, which provides low latency connectivity in the form of a high-radix crossbar. However, a single crossbar is insufficient to support kilo-core and larger systems, as crossbar area scales with $n^2$. For systems with IP instances numbering in the hundreds or thousands, a mesh of crossbars is more appropriate.

In this paper, we introduce NanoMesh: a general purpose, flexible SoC framework consisting of a Nexus crossbar-based mesh network paired with an asynchronous MIPS R3000-like CPU core as the building block of a many-core system. The circuits obey the Quasi-Delay Insensitive (QDI) [8] timing model and use integrated-pipelining templates [9]. This approach tolerates delay variability, consumes minimal power when idle, and has intrinsic flow control. Non-CPU IP blocks can be included in the mesh, including synchronous blocks wrapped in clock-domain conversion circuitry. The data and instruction memory of each asynchronous core is directly exposed via the low-latency crossbar-based mesh network, allowing any IP block to write programs or data to any core. This enables NanoMesh to dynamically reconfigure itself to changing workloads. In this way, the NanoMesh SoC framework provides scalable, customizable computational power and flexibility.

There are a number of existing multiprocessor SoC designs [10], such as the Intel Single-Chip Cloud Computer (SCC) [11], the Tilera TILE series [12], and SpiNNaker [13]. All three designs use relatively different core architectures than proposed by this work (an IA-32 core, a custom VLIW core, and an ARM968 core, respectively), but a general comparison of system architecture for design space context is valuable. The SCC and Tilera architectures are the most similar to NanoMesh in that they offer a sea of general-purpose compute cores connected by a 2D rectangular mesh. In contrast, SpiNNaker is a multi-chip architecture scalable to 65536 18-core chips specialized for use in modeling neural-networks.

We devote Sections II, III, IV, and V to a description of the NanoMesh architecture and programming model, making comparisons to other architectures where appropriate. NanoMesh is still in active development and a more complete analysis and characterization is forthcoming. Sections VI and VII discuss implementation details and what characterization data we have available. A test chip is expected in late 2013.

## II. NanoMesh Architecture

The NanoMesh architecture is logically and physically partitioned into tiles connected by two separate full-duplex

mesh networks. Each tile contains 8 IP instances and a pair of 16-port mesh routers—each built around a 16x16 Nexus crossbar. For the sake of discussion, in this paper we assume each IP instance in a tile is an asynchronous CPU core, but the NanoMesh architecture can easily support arbitrary IP instances. A maximal NanoMesh configuration, 16 tiles by 16 tiles, supports 2048 IP instances, plus 188 additional IP instances or I/O interfaces via the routing links at the mesh network periphery. The relative simplicity of our Network-on-Chip (NoC) and cores allow us to target more ambitious IP instance counts than the current SCC or Tilera designs.

The pair of mesh networks in the NanoMesh architecture is actually two separate physical mesh networks routed in parallel: Request and Response. One mesh services requests from cores to resources, and the other responses from resources back to cores. This avoids cyclic protocol deadlocks and provides additional bandwidth. This is a simple brute-force alternative to virtual channels [14], which add complexity and latency to routers, or credit-based algorithms [15], which scale poorly due to the need for local buffering proportional to the total system size.

In general, virtual channels and credit-based algorithms seek to maximize the utility of a single NoC by building several logical networks on top of a physical network, as in SCC [16]. However, since modern process technologies offer abundant planar wiring resources, building multiple physical networks is a simple, if unconventional, alternative [17]. Tilera also uses several specialized physical mesh networks [18] to simplify design, increase overall bisection bandwidth, and minimize the need for on-chip buffers. In contrast, SpiNNaker makes heavy use of multicast traffic and hierarchical routing topologies to mitigate the high costs of off-chip communication [19].

Keeping in the theme of simplicity, the NanoMesh routing protocol is dimension-order (avoiding deadlocks from routing) and heavily leverages the Nexus crossbar. Flow-control is implicitly handled by the handshaking in the QDI circuit implementation, as is buffering—each electrical buffer is also a WCHB-style pipeline buffer. It is worth noting that SpiNNaker's communication network is fully asynchronous and that they follow a GALS system design style, connecting synchronous computation islands with asynchronous interconnect. On-chip, SpiNNaker leverages an asynchronous crossbar to do inter-core communication and route traffic to and from the off-chip network, which is also asynchronous [20]. In this way they can leverage their synchronous ARM core IP while still obtaining the same benefits as NanoMesh with regards to the asynchronous interconnect. Our mesh of crossbars is a good fit for our relatively simple NoC, as we are not targeting an intra-chip network.

The Nexus crossbar [3] protocol can be seen in Figure 1a. It supports variable-length flits, terminated by a *true* value on the *Tail* channel. The output port is selected by sending the port number on the *To* channel in parallel with the first data word.

To support mesh routing, we modify the protocol to send a 32-bit header word followed by the data payload, again terminated by a *true* value on the *Tail* channel, as shown in Figure 1b, where the header word is denoted by `H`. Figure 2 shows the header word structure. A flit consisting of only the header word is possible, simply by sending the header word with a *true* value on the *Tail* channel. Such a flit is useful for control messages where the message payload is contained in the `Ctrl` byte of the header word.

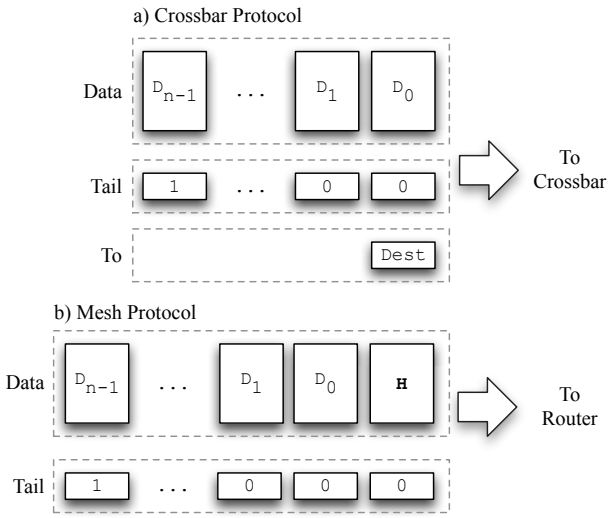


Fig. 1. Crossbar and Mesh Protocols. Header word denoted by `H`

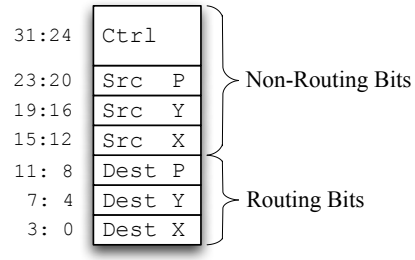


Fig. 2. NanoMesh Header, expanded from Figure 1b. The numbers to the left indicate bitfields.

As shown in Figure 3, eight ports of each mesh router are connected to/from the cores, leaving the other eight ports for mesh routing. Eight routing ports allows us to support diagonal routes in addition to cardinal routes, reducing cross-mesh latencies and increasing the bisection bandwidth.

Routing decisions are made with a simple $<, =, >$ compare between the router's local $X, Y$ mesh coordinates and the destination $X, Y$. Diagonal routes are chosen if both the $X$ and $Y$ coordinates differ, and cardinal routes if only one coordinate differs. Once the $X, Y$ coordinate is reached, the final crossbar port is chosen directly by $P$, which is 4-bit to enable access to ports 8-15 for routers at the mesh boundary—used for I/O or other IP instances. Note that NanoMesh does not have any explicit fault tolerance or error correction hardware. SpiNNaker, due to its size and complexity, implements emergency routes in hardware for fault tolerance and congestion avoidance, and

has the added benefit that neural networks are inherently fault-tolerant [19].

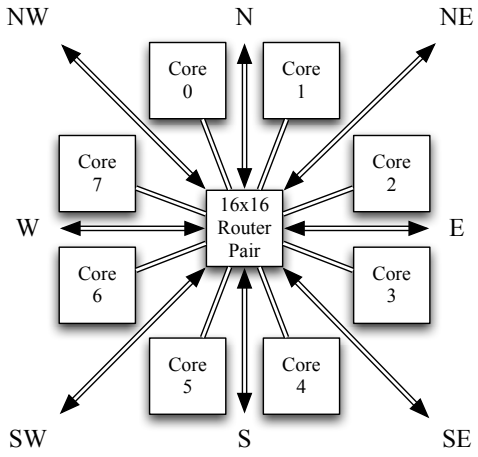


Fig. 3. A NanoMesh tile, which contains two 16x16 routers (built around 16x16 Nexus crossbars) and 8 cores. Both cardinal (N,S,E,W) and diagonal (NE,SE,SW,NW) routes are supported. Note that each double-line edge represents links from each mesh network (Request and Response), routed in parallel.

Clustering the cores in sets of eight naturally lends itself to memory spatial locality and divide-and-conquer style parallel programming. To support this, the programming model in NanoMesh is an un-cached shared-memory model, where the memory is distributed across all cores in the fabric (an approach sometimes called Processor-in-Memory [21]). I/O is also memory-mapped, as are interfaces with non-CPU cores. Every IP block has full read/write access to every core's local memory, and we add hardware and ISA support for block data transfers and message passing.

The NanoMesh core CPU is single-issue and in-order, with a 32-bit integer datapath. Its RISC instruction set ($n = 58$) is derived from the MIPS R3000 to leverage existing assembler and compiler tools (`gcc`). We have previous experience with this ISA, and can directly compare it to the MiniMIPS [22]. Each core has 64KB of local SRAM and supports several memory addressing modes. Different physical mappings are selected by the high-order bits of the address, enabling explicit software control over memory locality:

- **Local**: Contiguous access to the core's local 64KB of memory via direct high-bandwidth, low-latency connections.
- **Remote**: Contiguous access to 64KB of a remote core's memory via the mesh network.
- **Tile**: Striped access in 64B chunks to the combined memory of all cores in a tile. For 8 cores per tile, this provides 512KB of memory. This level of memory satisfies producer-consumer ordering and still has low access latency.
- **Global**: Striped access in 64B chunks to all memories across the NanoMesh. Access is first striped across tiles then within tiles. Assuming 512 cores, this provides 32MB of memory. Read-write ordering is maintained, but producer-consumer ordering is not.

Core remote memory accesses generate point-to-point traffic and read/write a single word per cycle. Our core has new instructions supporting block memory copies up to 64B. Instruction and data memory is completely shared. This enables quick software reconfigurability, and allows efficient use of scarce on-chip memory.

Currently, the request mesh carries only load-request and store-request flits. These start with the header word, then an address word, then in the case of the store, 1 to 16 words of data. The Response mesh carries only load-response flits. These start with the header word, then 1 to 16 words of data. The number of words to be loaded or stored is encoded in `Ctrl` field of the header word. The header word also contains the source and destination $(X, Y, P)$ tuples for routing.

These load- and store-flits are self-contained and point-to-point communication is guaranteed to be in order. Unlike Tilera, we do not support explicit data streams, although in practice one can simply send many load- or store-flits. This eliminates the need for hardware to de-multiplex flit arrival from multiple sources, as is the case in Tilera [18], because each flit represents a single load or store operation. Return traffic addressing is trivial, as we simply reverse the source/destination tuple fields in the return flit's header word.

Our mesh network itself is capable of supporting additional protocols (e.g. cache coherency) that may be needed in future enhancements. Likewise, the 16-word limit is imposed by the core, not the mesh.

As seen in Figure 4, each core is logically and physically partitioned into a memory (MEM) and a compute (CPU) subsystem, each with feature-complete top-level CSP [23]. To save on area, we can instantiate the memory subsystem by itself as an IP instance. This provides the same storage capacity without the area and energy cost of instantiating the CPU subsystem. This is possible because the mesh network interfaces are entirely encapsulated in the memory subsystem, as shown in Figure 4b. The CPU interface to the mesh is abstracted via memory mapping.

The CPU itself is composed of an instruction decode unit (DEC) and four functional units: a arithmetic/logic unit (AL), a multiply/divide (MD) unit, a branch/jump unit (BJ), and a load/store unit (LS). Table I lists the MIPS instructions that each unit implements, as well as the new instructions for communication and synchronization. We also added ISA support for power management, which SCC does by implementing software-controlled DVFS [16] to maximize power savings. In contrast, our QDI circuit implementation offers fine-grained, data-driven clock-gating-like behavior implicitly. As a result, the power consumption of a NanoMesh implementation will scale with system activity. Our addition of the `HALT` and `WAIT` instructions, described in more detail in Section III-A, expose direct control over activity to software, forcing cores to enter and remain in a low-power wait or halt state until a message or interrupt is received. For more aggressive power management, we can turn to power gating techniques for asynchronous circuits [24].

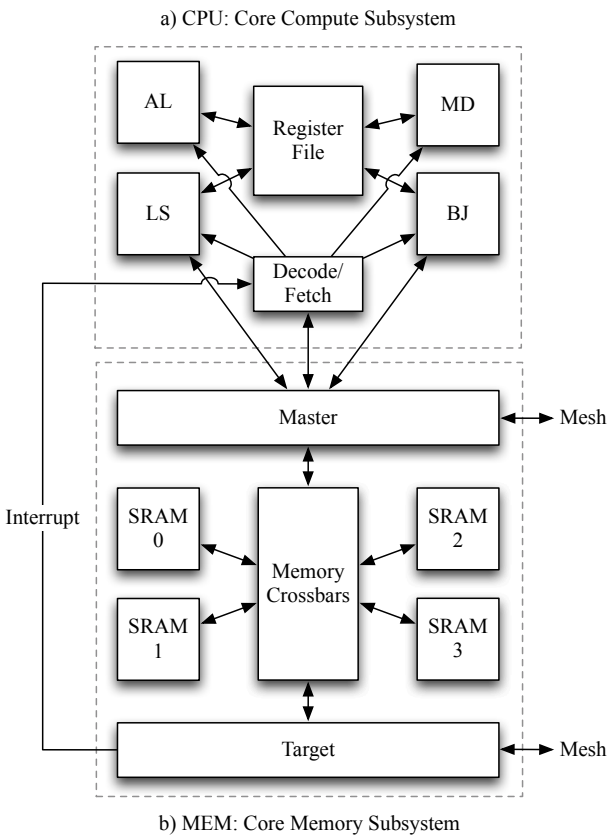


Fig. 4. NanoMesh Core, separated into the compute (CPU) subsystem and memory (MEM) subsystem.

We eliminated some instructions that are not used by `gcc`. Operating system support has been replaced with a simple exception and interrupt handler. Exceptions are currently only caused by unknown instructions. To run a real OS, we would need to add precise exceptions for memory protection, unlike SCC and Tilera, which already support running a full operating system.

Unlike the MiniMIPS, we kept byte and short memory access, as these are important for compatibility and benchmark performance. We also added new instructions and features to better support multi-core communication, as described in Section III. Currently these instructions are not automatically generated by `gcc`, so we manually insert them into code with `gcc`'s inline assembly calls.

## III. CPU: Compute Subsystem Architecture

As described in Section II, the CPU contains an instruction decode unit, a register file, and four function units connected by flow-controlled channels.

### A. DEC: Instruction Decode

The DEC extracts the function unit and register usage from each instruction, forwarding the appropriate information to the register file and selected function unit. It also probes for interrupts, and raises exceptions for unknown instructions (including `SYSCALL`). Interrupts and exceptions jump to the exception handler at address 0 and disable interrupts. Also, the program counter (PC), a bit to distinguish interrupts from exceptions, and a bit to indicate if the previous instruction was a taken branch are all saved to the `Cp0[0]` register[1].

TABLE I
Instructions by Unit

| Unit | R3000 | New |
|---|---|---|
| DEC | `NOP RFE SYSCALL` | `HALT WAIT` |
| AL | `ADDU ADDIU SUBU AND ANDI OR ORI NOR XOR XORI LUI SLL SLLV SRA SRL SRLV SLT SLTI SLTU SLTIU` | |
| MD | `MULT MULTU DIV DIVU MFHI MFLO MTHI MTLO` | |
| BJ | `BEQ BGEZ BGTZ BLEZ BLTZ BNE BGEZAL BLTZAL J JAL JALR JR MFC0 MTC0` | |
| LS | `LB LBU LH LHU LW SB SH SW` | `CF4 CF16 CT4 CT16` |
| Omitted | `ADD ADDI SUB LWL LWR SWL SWR` | |

DEC implements two new communication-related instructions. `HALT` stalls until a preemptive interrupt arrives from the mesh via the memory subsystem. `WAIT` is meant for cooperative interrupts or message passing. Remote cores can write to a memory-mapped address space in the local 64KB, which sets a "dirty" flag. `WAIT` stalls until the dirty flag is set, then clears it and continues sequential execution. A program executing a busy-waiting loop for semaphores should put a `WAIT` in the loop, which will eliminate wasted power. Program execution continues when a write to a sensitive memory address occurs, so that the program can re-evaluate the wait condition. This `WAIT`-based stalling can efficiently implement CSP-style message passing, including arbitration, as done in the Vortex CPU [25]. The DEC implements the `NOP` instruction directly to save power (instead of executing a vacuous shift operation, as in the R3000).

DEC is decomposed into PREDECODE and ISSUE stages. PREDECODE has the complex logic for distinguishing function units, register formats, and the few instructions implemented directly by ISSUE. ISSUE has a state loop to poll for interrupts as well as launch the initial PCs to fill the fetch loop on boot.

### B. REG: Register File and Bypass

The register file, shown in Figure 5, exposes dedicated operand/result ports to each of four function units for a total of 8 read and 4 write ports. Three stages of bypass logic optimize data-dependent latency, and reduce the parallelism to single-issue in the data store. The data store is implemented with two dual-ported 8T SRAMs. Writes are mirrored across both SRAM write ports, and each provides an independent read port.

The result of each function unit is passed through a 3-way Copy Filter (CF3) that conditionally (per output) produces a copy of its input. One of the outputs is routed to a 2-way conditional merge (M2), which feeds the CF3 in the next stage.

[1] `Cp0` is the MIPS Coprocessor 0 register bank

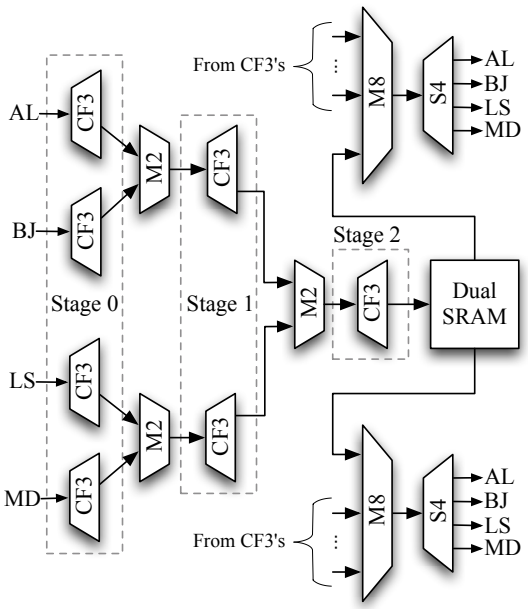


Fig. 5. Register File with three stage bypass logic. M2 and M8 are 2-way and 8-way merges. CF3 is a 3-way copy-filter, and S4 is a 4-way split. Each CF3 is connected to the next stage of bypass logic, shown with bold lines, as well as both M8's, although these connections are omitted for clarity.

The other two outputs of each CF3 are routed to 8-way merges (M8). The M8's also connect to the SRAM read ports. Each M8 is followed by a 4-way split (S4) that sends operands to each function unit.

This design allows direct bypass routing of the result of any instruction to the operand of the next instruction in only 3 domino stages of latency ($< 300$ps). It can also bypass results from 2 or 3 instructions previous, relaxing the latency goals of the SRAM and allowing function units to take more than one cycle's latency. However, the 2-way merging tree forces the 2nd previous result to be partially ordered before being bypassed, and the 3rd previous result and ultimate writeback is fully ordered. Writes to register 0 (a constant 0 in the R3000) are discarded in the first stage CF3, saving power as well as avoiding needless timing dependencies.

This design is both more elegant and absorbs more latency variation than the dual alternating write-back buses employed by the MiniMIPS for the same purpose. The bypass, however, is several times larger than the dual SRAM itself.

The control of the register file has to remember the function unit and destination register of the last 3 instructions, and compare them to the 2 operand registers of the current instruction to decide which bypasses (if any) to perform.

### C. BJ: Branch, Jump

The BJ unit is responsible for the R3000 branch and jump instructions, plus it holds the `Cp0` registers. The `Cp0[0]` register is used to store information for interrupts and exceptions. The `Cp0[1]` register is read-only and provides the 12-bit $X, Y, P$ location of this core in the mesh.

The BJ keeps a history of the last few PCs fetched, which include the exception PC, the base for relative branches, and the PC linked to the register file on subroutine calls. The R3000 ISA defines a single-cycle branch delay slot, which implies a two-cycle loop for fetching and decoding the next instruction. Our fetch loop latency is several cycles longer, due to the large shared SRAM and synthesis inefficiencies. Thus, we implement a straight-line prefetch mechanism, which fetches an additional N_PREFETCH tokens more than the expected 2 tokens in the loop. If a branch is taken, the N_PREFETCH instructions after the branch delay slot are canceled by the DEC, after which the correct sequence of instructions resumes. Our design has no branch prediction mechanism. Still, prefetching is enough to feed the pipeline at full frequency in the absence of taken branches. The 2-token latency loop implied by the ISA is between the BJ and DEC, which allows for easy timing closure.

For performance and power reasons, the BJ is decomposed into BJ_NEXT and BJ_CORE. BJ_NEXT is active on all instructions and is responsible for computing the sequential PC for non-branches, and maintaining the PC history and branch-taken information. On actual branch or jump instructions, the BJ_CORE is activated to communicate with the register file and compute the target address. BJ_CORE also implements the `Cp0` registers.

### D. AL: Arithmetic, Logic, Shift

In the original R3000 (and the MiniMIPS), there are signed ADD/SUB instructions that raise exceptions on overflow. We omit these instructions as C compilers never use them, and we don't implement data-dependent exceptions. Still, since all unknown instructions raise precise exceptions, it is possible to emulate them in software, if backward compatibility is absolutely required.

The arithmetic unit is synthesized from a single CSP cell. It is likely that better power or latency could be achieved by decomposing the longer latency shift instructions into a subcell and only conditionally using them.

### E. LS: Load, Store

As shown in Table I, we support byte, short, and word-sized loads & stores, with and without sign extension. Partial stores are currently implemented with a read-modify-write handshake with the memory system. This can be very slow for remote memory, so word stores are strongly preferred. Eventually we plan to push the read-modify-write into the memory system itself, and to the far end of a remote write. The `LWL`, `LWR`, `SWL`, and `SWR` unaligned word load/store instructions of the R3000 are unimplemented as modern compilers don't use them.

To enhance mesh communication, we add a family of block memory copy instructions: `CF4`, `CF16`, `CT4`, and `CT16`. These use one register operand plus a 16-bit immediate to compute an address in the local 64KB memory. The other register operand gives an address in a remote address space (a core can refer to its own memory, but it will still communicate

through the mesh). The CF are "copy-from" a remote to a local address, and the CT are "copy-to" a remote address from a local one. The "4" and "16" are the number of words copied. The addresses must be aligned to the block size so we don't have to split them into multiple flits to different destinations. The intention is that multi-word structures in remote memory can be copied into local memory, operated upon, and copied back. The CF instructions help enormously, because they allows up to 16 words to be read with only one round-trip latency, compared to 16 times longer if the words were read one at a time. We will add 2 and 8 word lengths, and will consider relaxing some of the alignment restrictions in the future.

### *F. MD: Multiply, Divide*

We implement the MIPS LO and HI multiply/divide registers, which allow the MD to iterate in the background while other instructions proceed. Only when the results are moved back to the main register file will it stall until the operations complete.

Despite an appealingly simple specification, compiling the $\star$ and / operators through the Proteus flow is not recommended. While functional, it would generate fully pipelined multiplier and divider structures, both of which would be active at once. By decomposing MD into 7 subcells, which communicate conditionally, we save on power and area.

First, the operands are converted into sign+value format instead of twos-complement. For multiplication, we perform 8-bit by 32-bit multiplication in one cycle, and shift and accumulate the results into the LO/HI registers. If one of the bytes of the first operand is zero, we skip that cycle. Thus multiplies take 1-4 cycles to execute. For division, we send the operands to an iterative divider subcell that performs long division, producing one bit of quotient per-cycle. However, if the numerator has most-significant bytes of 0, we skip ahead, so division takes 8, 16, 24, or 32 cycles. The quotient and remainder are written to LO and HI.

Since the LO/HI registers already include an accumulator, it would be trivial to add multiply-accumulate instructions. We intend to further optimize the MD system, mostly to reduce its area. It currently accounts for 40% of the standard cells in the core.

## IV. MEM: Memory Subsystem Architecture

The memory subsystem contains four 16KB banks of SRAM, striped by word, a Master interface, a Target interface, and a crossbar-based datapath, as shown in Figure 6. The Master and Target each have input and output interfaces to the mesh network. The Master sends remote read/write requests and receives the load response. The Target services incoming read/write requests and sends out read responses.

The Target is the simpler of the two units. It has dedicated Address, Read, and Write channels to the crossbars. All data is either written to the Target's local SRAMs or read from the Target's local SRAMs.

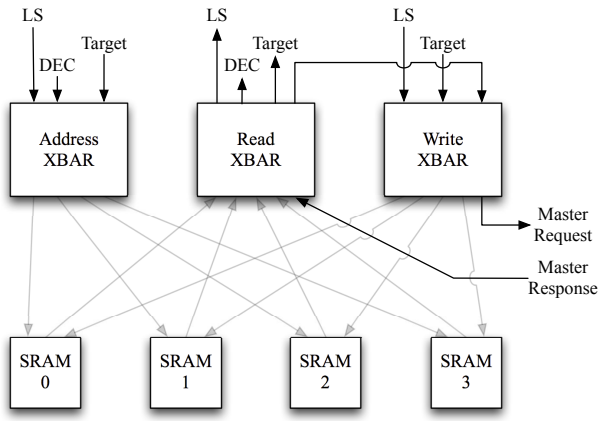


Fig. 6. The memory datapath is a crossbar-based datapath with three separate crossbars. The Address crossbar is 12 bits wide and the Read and Write crossbars are 32 bits wide.

The Master services requests from the BJ and LS units of the CPU. Instructions and data share the same memory. The Master routes dedicated Read and Address channels from the BJ and LS channels to the memory crossbars, as shown in Figure 6. If the data and instruction access map to different banks of SRAM, they complete in parallel. Otherwise, the instruction access is done first.

The Read crossbar has a port from the Master's response mesh interface to route remote instructions or data directly to the DEC or LS units. In the same way, the Write crossbar has an output port to the Master's request mesh interface to allow the LS unit to write to remote memory.

The channel from the Read to the Write crossbar is for block copy instructions. For CT4 and CT16, data is pulled from local SRAM and sent out on the Master request mesh as a write request to remote memory. For CF4 and CF16, data is received from the Master response mesh and written to local SRAM.

As discussed in Section III-A, each core has support for memory-based message passing. There are two types of messages, implemented by a remote write to a special memory address–denoted by a high-order bit. In effect, the memory space is aliased. A write to an unaltered address is treated normally, but a flag (implemented as a high-order memory address bit) encodes the presence of a message. Writing to the zero address with this bit set is a conventional preemptive interrupt, whereas a write any other address with this bit set raises a "dirty memory" flag, indicating that a remote process has written to the core's memory. The interrupt is required to reverse the effects of a HALT instruction, whereas raising the dirty flag will bring a processor out of the idle state set by a WAIT instruction . The Target interface is responsible for generating a token on the *Interrupt* channel, shown in Figure 4.

## V. NanoMesh Programming

Programming a NanoMesh core is done through standard R3000 assembly or C code compiled with a readily available

gcc toolchain. Implementing malloc() is relatively complicated because of all of the addressing modes described in Section II. A careful implementation would partition memory into disjoint free pools, so software could allocate appropriately. Currently we keep instructions, stack, and globals in local memory, and let software explicitly manage remote memory addresses. Compiler modification to support the HALT, WAIT, and the memory copy instructions is forthcoming. The interrupt and dirty memory bit message passing mechanisms described in Section IV are completely exposed to the programmer. One can simply take the logical OR of a memory address with a predefined bitmask to set the high-order bit for message passing.

We have written specialized assembly implementations for programs that require I/O or system call functions such as printf() and time(). These functions generate memory read/write requests to remote memory addresses, which resolve to a peripheral mesh port at the southwest end of the mesh. This port is attached to a debugging process that can service time() requests as well as consume strings generated by printf(). In software, this enables code running on a core to echo messages to a terminal for human consumption and debugging. Eventually, this would be replaced with some form of off-chip I/O.

Loading programs onto a core is a simple matter of writing the program to the appropriate location in the core's 64KB of memory. Each core starts in a HALT state on power-up, so an interrupt is needed to jump to the bootloader and begin program execution. Since any IP instance on the mesh network has direct access to any core's memory, any instance can write programs to and run programs on any target core.

This memory model allows the NanoMesh system to serve as a pool of dynamically configurable accelerators. As the SoC workload changes, cores can be dynamically programmed to perform accelerator-like functions. Dispatching data to the cores is as simple as writing it to each core's memory, as the "dirty" memory mechanism can handle synchronization. The striped memory access modes discussed in Section II also work with the dirty memory mechanism, enabling easy broadcast-reduce operations. Furthermore, due to the WAIT instruction, these core-based accelerators can be fed streams of data. Each core executes an infinite loop interrupted by a WAIT instruction. Data arrives and is processed, results are written back, and the core returns to an idle state until more data arrives. Once the workload no longer has need for the additional compute bandwidth, the core can be halted and its instructions rewritten.

## VI. Implementation

The NanoMesh uses the Fulcrum flow for asynchronous circuit design, which started at Caltech and was commercialized by Fulcrum Microsystems. The circuits use integrated pipelining (i.e. WCHB, PCHB, PCFB) templates [9], which rely on dual-rail (or 1-of-N) domino logic combined with asynchronous channel handshakes. The implementation satisfies a QDI timing model, which means it is robust to arbitrary timing of transistors and most wires except for certain "isochronic" forks.

Our design starts with high level CSP [23] and proceeds by recursive decomposition into smaller cells that communicate by FIFO channels. The very top level behavioral CSP in this project is just 3 cells: the 16-way router for the mesh and the CPU and MEM halves of the core. This level of detail is feature-complete although not performance accurate. It is the golden model against which we check behavior of the final circuit.

These CSP cells are manually decomposed several levels, resulting in a dozen or so CSP cells. At this point, many cells are ready to run through the Proteus flow [26], which is an asynchronous synthesis, place & route flow targeting a standard cell library of about 300 cells. However, certain types of cells are not supported by Proteus, and instead follow our custom design flow which continues CSP decomposition process down to thousands of leaf cells with custom transistor netlists. This is needed to implement SRAMs and nondeterministic circuits. The custom flow also gives much better results for certain large, regular structures, such as the memory crossbars, register file, and register file bypass.

The top-level assembly process is simple, as the FIFO channel interfaces at the boundary of most cells allow for a very modular, snap-together assembly process. While analog effects are of course important, correctness is generally preserved in all but the most pathological cases due to our robustness to PVT variation and wire and gate delays.

### A. Proteus

Most of the NanoMesh design is run through Proteus. Proteus can currently handle moderately large cells, corresponding to about 10K synchronous gates. This is conveniently about the size of CPU and MEM logic combined and flattened for place & route.

In the CPU, the instruction decoding and arithmetic is synthesized from CSP by Proteus. The source code is concise and easy to modify. The first level of decomposition is to DEC, REG, AL, BJ, LS, and MD units. As described in Section III, some of these units have been further decomposed for performance/power reasons. In particular cases, such as the interrupt hardware in DEC, certain cells have non-deterministic circuits and therefore must use the custom flow.

In the MEM, the computation of addresses, the Mesh protocols, and various control signals are all synthesized with Proteus. However, most of the MEM area is custom circuitry.

The Proteus parts of the core are a slightly too large to perform flat place and route quickly. Therefore we decided to harden the MD as a macro, since it contains about 40% of the total gate count but interacts simply with the rest of the design. Then we placed and routed the rest of CPU and MEM flat.

### B. Custom

The interrupt sampling in DEC is implemented with the custom flow, which has nondeterministic cells. This is a small

macro with only a few unique cell types. There is a similar, tiny non-deterministic custom macro in the MEM system, used to arbitrate access between the Master and Target halves to the local SRAM banks.

For optimization purposes, the CPU's REG is mostly custom. The actual storage uses a pair of dual-ported SRAMs configured as a 2-read, 1-write register file. The bypass datapath is custom, as its conditional communication blocks and crossbar-like structures are expensive to synthesize. The control of the register file and bypass is still synthesized.

The 4 banks of single-ported SRAM in the MEM system are of course full-custom. At the moment, we are using an in-house asynchronous SRAM design [27]. In the future we will probably switch to a commercial synchronous SRAM core, wrapped with a synchronous-to-asynchronous interface. There are inherent timing assumptions in this conversion, namely in the timing of delay lines, but this is well-studied and easily tunable.

Aside from the SRAMs, the read, write and address crossbars between the 4 banks are full custom. Again, these synthesize poorly but are quite regular and have a low unique cell count when built with the custom flow. Most of these crossbar cells have been in Fulcrum's library for years, and are easily reused.

The total number of unique custom leaf cells is expected to be under 100, which adds to the layout cost of the design and process porting, but is less than the effort required to port the standard cell library. Many of the custom cells will be used for other projects. The benefits of just a few custom blocks are quite large.

### C. Cables

Long distance channels between crossbars, CORE's, and neighboring tiles use our new "cable" chip assembly methodology. A user draws a guide path for a channel, binds it by name to the channel in the HDL, and our tool automatically draws dense wiring and pipelined buffers underneath the wires. Since pipelining is built-in, these cables can travel an arbitrary distance at high frequency without concerns such as clock skew. The wiring pattern interleaves two e1of4 channels between power supply shields, such that each signal wire never sees simultaneous coupling aggressors on both sides. The length between hops is bounded to maintain the target frequency and noise immunity, which is thoroughly characterized. The hop length is typically $300\mu\mathrm{m}$ between pipelined buffers. The latency is only about $0.6\mathrm{ns}$ per $1\mathrm{mm}$ in SS, 0.81V, $0^{\circ}\mathrm{C}$. For the 33 bit channels of the NanoMesh, using 3 layers of wiring in each direction, a cable is only $10\mu\mathrm{m}$ wide and will run over $2\mathrm{GHz}$ at nominal PVT, yielding about 64Gbps bandwidth.

### D. Analog Verification

The Fulcrum flow has many tools to verify the correctness and performance of asynchronous circuits. A crucial tool is *alint*, which extracts the relevant subcircuit for each net in a cell, and characterizes the delays, leakage, cap-coupling and charge-sharing bumps for that net. The delay information is used by *asta*, our static-timing tool, to check performance. The remaining tests check that the analog circuit will behave digitally over a range of PVT conditions. These tests happen at the leaf and mid-level cell stages of design hierarchy. Inter-cell routing capacitance is accounted for via *alint*'s extraction of the relevant circuits on a particular net of interest.

The circuits generated by our flow usually pass all these analog tests. The occasional failure typically results in a small routing ECO, or less often, a change to floorplanning or design. At the moment, only the leaf cells used in the NanoMesh have been verified, but we don't anticipate problems that cannot be addressed with place & route tuning given our prior experience and the relative maturity of our flow.

## VII. Characterization

The NanoMesh project is meant as a proof-of-concept, so we focused on system-level design as opposed to deep optimization passes. Therefore, there are no particular performance goals for the CPU in this phase. However, we do want to prove the performance of the mesh and memory system, which is more heavily custom-built and finalized.

Currently, our design is complete but poorly optimized. Nevertheless, it is close to providing estimates of performance, area, and power in simulations. The custom blocks and Proteus standard cells have been floorplanned and sized (with our in-house transistor sizing tool), and can be simulated with a Manhattan Steiner tree wiring model and approximate diffusion parasitics. Area of cells is estimated based on the transistor area and overheads, which has been fairly accurate for past projects. The P&R timing model should be also fairly accurate. We haven't had resources to start the leaf cell layout or custom wiring. Luckily, many of the cells we need will be created for other projects.

The analog verification flow guarantees that the analog circuit is at least as fast as the digital circuit. Thus, we can run most of our performance analysis on the digital PRS (i.e. gate-level) design. This runs in *dsim*, our in-house digital simulator.

We report performance estimates in transitions-per-cycle (tpc), which is a count of digital transitions between the arrival of data at a process and the when the process is ready to accept a new data token. This offers a technology-independent measure of performance, as the number of transitions is governed by the circuit topology and handshaking protocol. Since we constrain the average delay per digital transition, this can be converted to a more traditional throughput measure in $\mathrm{Hz}$.

Most custom blocks run at 18 transitions-per-cycle (tpc), except for the Nexus crossbar component of the mesh, which has an internal bottleneck of 22 tpc, and the register file, which reads at 20 tpc. Proteus blocks target 22 tpc. This is an aggressive cycle time target. Some custom CPUs of an earlier era (the Pentium 4) had small portions running at 10 tpc, which ran at $8\mathrm{GHz}$.

In our current TSMC $28\mathrm{nm}$ HPM process, we have simulated the entire mesh router at $2010\mathrm{MHz}$ at the nominal PVT of TT, 0.9V, $50^{\circ}\mathrm{C}$ or $1317\mathrm{MHz}$ at SS, 0.81V, $50^{\circ}\mathrm{C}$. Higher

voltages yield higher frequencies (i.e. 2282MHz at TT, 0.99V, 50°C). We expect other 22 tpc circuits will keep up. Energy per word transferred at nominal PVT for the mesh router is about 16pJ (giving a peak power of 0.52W per router at 2GHz).

Given limited resources, the only real performance benchmark we can currently demonstrate is that straight-line AL instructions without data dependencies executes at 22 tpc. Taken branches will result in a several prefetched instructions getting canceled (about 33% of Dhrystone instructions are canceled this way). Data dependencies can cause stalls. Load and stores can stall because of memory bank conflicts and load latencies.

The Dhrystone 2.1 benchmark is currently running at 0.364 VAX MIPS per 22 transition cycle or 0.469 if we allow function inlining between files (as many companies report). Our high-level CSP with no prefetched instructions or memory conflicts or data dependent stalls yields 1.073 and 1.317 VAX MIPS per cycle respectively. So there is a long way to go before this architecture is optimized.

The gate count and area of the two Proteus blocks is shown in Table II. Image gate count is only the LOGIC and DFF equivalents of the Proteus library, which would be the gate count for an unoptimized synchronous implementation (no inverters or clock distribution or gate cloning). As expected with this approach, the Proteus area is several times larger than a synchronous design, but the frequency is also higher, and half of the area is still in dense hard macros.

TABLE II
GATE COUNTS AND AREAS

| Block | TPC | Image Gates | Async Gates | Gate Area ($\mu m^2$) |
|---|---|---|---|---|
| CORE | 24 | 3523 | 37513 | 109509 |
| MD | 26 | 3282 | 28594 | 71428 |

The overall floorplan of one core is shown in Figure 7. Each core is currently $0.68mm^2$. An 8-core tile is shown in Figure 8. The tile is $6.8mm^2$, including the necessary $40\mu m$ gaps for the mesh wires and buffers. In 28nm, a 64-tile, 512-processor system would be a feasible $435mm^2$, plus some more for I/O. 1024 processors would be possible in 22nm.

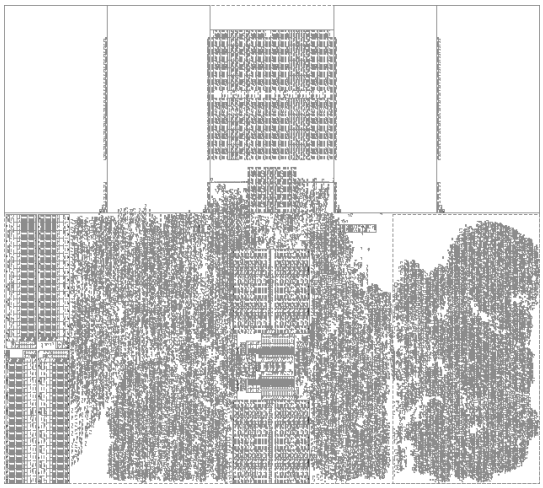

Fig. 7. The CORE floorplan after trial place and route. The 4 SRAM banks are empty boxes at the top, with memory crossbars in the the middle. The lower left edge includes the mesh interfaces, the lower middle is the REG, and the lower right corner is the hardened MD.

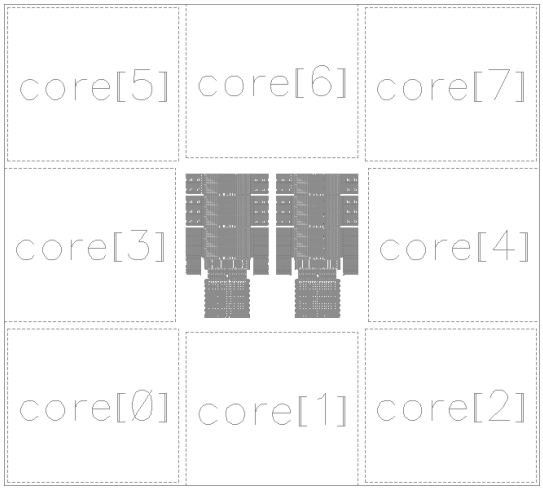


Fig. 8. An 8-core tile with mesh routers in the middle. Gaps are sufficiently wide for the wiring cables of the mesh.

## VIII. CONCLUSION

With a few man-months of effort, we were able to develop NanoMesh, a complete kilo-core-feasible, gate-level SoC design capable of running simple C programs such as the Dhrystone benchmark. Each subsystem is highly modular and the SoC as a whole is easily extensible with custom IP blocks.

The NanoMesh concept was partly inspired by the Caltech Mosaic [28], which was a (multi-chip) mesh of simple CPU's with 64KB of local memory each. Our processor is a follow-on to the Caltech MiniMIPS, with heavy reliance on synthesis, place & route made possible by Proteus. Comparisons can also be made to the Tilera architecture, the Single-Chip Cloud Computer demonstrated by Intel and of course the 61-core MIC products Intel sells today. We believe that asynchronous circuit techniques have significant benefits to offer for large SoC's such as these. In particular, we predict our design will achieve frequencies of several GHz, with up to 1000 cores per die, when ported to a cutting edge 22nm process.

NanoMesh's shared-memory, dynamically programmable computation model enables it to respond to changing SoC workloads and pushes more of the design effort towards software. New instructions for block data transfers and communication instructions enable efficient parallel programs. By saving hardware design time, we believe NanoMesh can help reduce time-to-market.

Our current performance numbers are not representative of a well optimized design. In particular, the CPU needs branch target prediction. Memory conflicts would be avoided by splitting code and data memory for the direct local access

(while still sharing memory for remote access). Further low-level latency optimizations should reduce stalling. A few more man-months of design effort will produce a test chip, expected in 2013.

In a TSMC 28nm process, we expect around 2GHz performance for typical parts, although currently only 938 VAX MIPS on (inlined) Dhrystone. Power numbers and additional characterization results are pending.

We intend to use the NanoMesh design primarily as a test case to improve both our asynchronous circuit templates and our tool flow. We hope some aspects of it may eventually find their way into production hardware.

## REFERENCES


[1] P. Teehan, *et al.* "A Survey and Taxonomy of GALS Design Styles." *Design & Test of Computers, IEEE*, 24(5), 2007.

[2] A. Martin and M. Nystrom. "Asynchronous Techniques for System-on-Chip Design." *Proc. IEEE*, 94(6):1089–1120, 2006.

[3] A. Lines. "Nexus: an asynchronous crossbar interconnect for synchronous system-on-chip designs." "IEEE HOTI," pp. 2–9. 2003.

[4] A. M. Scott, *et al.* "Asynchronous on-Chip Communication: Explorations on the Intel PXA27x Processor Peripheral Bus." *IEEE ASYNC*, pp. 60–72, 2007.

[5] P. Vivet, *et al.* "FAUST, an Asynchronous Network-on-Chip based Architecture for Telecom Applications." *Proc 2007 Design*, 2007.

[6] R. Dobkin, *et al.* "QNoC Asynchronous Router with Dynamic Virtual Channel Allocation." *IEEE NoCS*, p. 218, 2007.

[7] I. Miro-Panades, *et al.* "Physical Implementation of the DSPIN Network-on-Chip in the FAUST Architecture." *IEEE NoCS*, 2008.

[8] A. Martin. "The Limitations to Delay-Insensitivity in Asynchronous Circuits." "6th MIT Conference on Advanced Research in VLSI," Proceedings of the 6th MIT Conference on Advanced Research in VLSI, 1990.

[9] A. Lines. *Pipelined Asynchronous Circuits*. Master's thesis, California Institute of Technology, 1995.

[10] G. Blake, *et al.* "A survey of multicore processors." *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[11] M. Gries, *et al.* "SCC: A Flexible Architecture for Many-Core Platform Research." *Computing in Science & Engineering*, 13(6):79–83, 2011.

[12] S. Bell, *et al.* "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect." "IEEE ISSCC," pp. 88–598. 2008.

[13] E. Painkras, *et al.* "SpiNNaker: A multi-core System-on-Chip for massively-parallel neural net simulation." *IEEE CICC*, 2012.

[14] W. J. Dally. "Virtual-channel flow control." "IEEE ISCA," ACM, 1990.

[15] N. T. Kung and R. Morris. "Credit-based flow control for ATM networks." *Network, IEEE*, 9(2), 1995.

[16] J. Howard, *et al.* "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling." *IEEE JSSC*, 46(1):173–183, 2011.

[17] Y. J. Yoon, *et al.* "Virtual channels vs. multiple physical networks: A comparative analysis." *IEEE DAC*, 2010.

[18] D. Wentzlaff, *et al.* "On-Chip Interconnection Architecture of the Tile Processor." *IEEE Micro*, pp. 15–31, 2007.

[19] J. Navaridas, *et al.* "SpiNNaker: Impact of Traffic Locality, Causality and Burstiness on the Performance of the Interconnection Network." "ACM CF," pp. 11–20. ACM Press, New York, New York, USA, 2010.

[20] L. A. Plana, *et al.* "SpiNNaker." *J. Emerg. Technol. Comput. Syst.*, 7(4):1–18, 2011.

[21] P. M. Kogge, *et al.* "Pursuing a petaflop: point designs for 100 TF computers using PIM technologies." "IEEE FMPC," pp. 88–97. IEEE Comput. Soc. Press, 1996.

[22] A. Martin, *et al.* "The design of an asynchronous MIPS R3000 microprocessor." *Advanced Research in VLSI*, 1997.

[23] A. Martin. "Compiling Communicating Processes for Delay-Insensitive VLSI Circuits." *Distributed Computing*, 1986.

[24] C. T. O. Otero, *et al.* "Static Power Reduction Techniques for Asynchronous Circuits." *IEEE ASYNC*, pp. 52–61, 2010.

[25] A. Lines. "The Vortex: A Superscalar Asynchronous Processor." *IEEE ASYNC*, pp. 39–48, 2007.

[26] P. Beerel, *et al.* "Proteus: An ASIC Flow for GHz Asynchronous Designs." *Design & Test of Computers, IEEE*, 28(5):36–51, 2011.

[27] J. Dama and A. M. Lines. "GHz Asynchronous SRAM in 65nm." *IEEE ASYNC*, pp. 85–94, 2009.

[28] C. L. Seitz. "Mosaic C: An experimental fine-grain multicomputer." "Future tendencies in computer science," pp. 69–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.