Intel® Nehalem Processor Core Made FPGA Synthesizable

Graham Schelle[1], Jamison Collins[1], Ethan Schuchman[1], Perry Wang[1], Xiang Zou[1]
Gautham Chinya[1], Ralf Plate[2], Thorsten Mattner[2], Franz Olbrich[2], Per Hammarlund[3]
Ronak Singhal[3], Jim Brayton[4], Sebastian Steibl[2], Hong Wang[1]

Microarchitecture Research Lab, Intel Labs, Intel Corporation[1]
Intel Germany Research Center, Intel Labs, Intel Corporation[2]
Central Architecture and Planning, Intel Architecture Group, Intel Corporation[3]
Microprocessor and Graphics Development, Intel Architecture Group, Intel Corporation[4]
Contact: graham.schelle@intel.com

ABSTRACT

We present a FPGA-synthesizable version of the Intel Nehalem processor core, synthesized, partitioned and mapped to a multi-FPGA emulation system consisting of Xilinx Virtex-4 and Virtex-5 FPGAs. To our knowledge, this is the first time a modern state-of-the-art x86 design with the out-of-order micro-architecture is made FPGA synthesizable and capable of high-speed cycle-accurate emulation. Unlike the Intel Atom core which was made FPGA synthesizable on a single Xilinx Virtex-5 in a previous endeavor, the Nehalem core is a more complex design with aggressive clock-gating, double phase latch RAMs, and RTL constructs that have no true equivalent in FPGA architectures. Despite these challenges, we are successful in making the RTL synthesizable with only 5% RTL code modifications, partitioning the design across five FPGAs, and emulating the core at 520 KHz. The synthesizable Nehalem core is able to boot Linux and execute standard x86 workloads with all architectural features enabled.

Categories and Subject Descriptors

C.1.0 [**Processor Architectures**]: General

General Terms

Design, Measurement, Performance

Keywords

Intel Nehalem, FPGA, emulator, synthesizable core

1. INTRODUCTION

Intel Nehalem [4, 8, 14] is the latest microarchitecture design and the foundation of the Intel Core™ i7 and Core™ i5 processor series. Like its predecessor (Intel® Core™ microarchitecture), Intel Nehalem microarchitecture continues to focus on improvements in how the processor uses available clock cycles and power, rather than just pushing up ever higher clock speeds and energy needs. Its goal is to do more in the same power envelope or even reduced envelopes. In turn, Intel Nehalem microarchitecture includes the ability to process up to four instructions per clock cycle on a sustained basis compared to just three instructions per clock cycle or less processed by other processors. In addition, Intel Nehalem incorporates a few essential performance and power management innovations geared towards optimizations of the individual cores and the overall multi-core microarchitecture to increase single-thread and multi-thread performance.

In addition to backward compatibility to the rich Intel Architecture legacy, the Intel Nehalem sports several salient new features: (1) Intel Turbo Boost Technology which enables judicious dynamical management cores, threads, cache, interfaces and power, (2) Intel Hyper-Threading Technology which in combination with Intel Turbo Boost Technology can deliver better performance by dynamically adapting to the workloads which can automatically take advantage of available headroom to increase processor frequency and maximize clock cycles on active cores and (3) Intel SSE4 instruction set extensions that center on enhancing XML, string and text processing performance.

In this paper, we share our experience and present the methodology to make the Intel Nehalem processor core FPGA synthesizable. The emulated Nehalem processor core is partitioned across multiple FPGAs and can boot the standard off-the-shelf x86 OSes including Linux and run x86 workloads at 520Khz. Compared to the Intel Atom core that we previously made FPGA synthesizable, the Nehalem core is much more challenging due to the microarchitectural complexity and sheer size of the design. The key contributions of this paper are

- We present our methodology to synthesize and judiciously partition the fully featured Nehalem RTL design to an emulator with multiple Virtex-4 [18] and Virtex-5 [19] FPGAs.

- We demonstrate a systematic and scalable cycle-by-cycle verification methodology to ensure the functional and timing correctness of the synthesizable design.

The remainder of the paper is organized as follows. Section 2 reviews related work and provides background information on the Intel Nehalem processor core and a multi-FPGA emulator platform. Section 3 elaborates our experience in making the Nehalem core RTL FPGA synthesizable and introduces our verification methodology. Section 4 describes how we partition the Nehalem core design across multiple FPGAs and provide memory interface between the core and the DDR memory. Section 5 evaluates the functionality and performance of the synthesized Nehalem core in comparison with the Intel Atom core on the same emulator platform. Section 6 concludes the paper.

2. BACKGROUND

2.1 Related Work

Intel Nehalem processor is representative of a general trend in designing the modern high performance energy efficient CPU. Beyond the traditional incremental microarchitectural enhancements at processor pipeline level, the state-of-the-art CPU designs tend to incorporate a variety of new technologies through a high degree of integration. Resembling a system-on-a-chip (SoC), modern CPU usually embodies very sophisticated on-chip resource managements and complex interactions amongst many building blocks. These building blocks work in concert to deliver the architecturally visible features in energy efficient ways.

The increase in design complexity will inevitably impact the pace of silicon development. In particular, pre-silicon RTL validation has long been a vital yet time-consuming phase of microprocessor development. Due to the heavy reliance on the software based RTL simulation tools, despite the rich test environments, throughput of validation is usually limited by simulation speed, which is in the range of the single-digit hertz. Such speed of system level tests is prohibitive for long running simulations as processor designs are ever increasing in size and complexity. One alternative has typically been emulation. Emulation, however, requires expensive hardware and software tools. The cost of the emulation hardware and tools usually scale up when the size of designs reaches the level of modern CPU. Consequently the speedups achieved in large emulation platforms actually decrease as designs grow larger, potentially negating benefits of emulation

Thanks to Moore's Law, the capacity and speed of modern FPGAs have continued to improve. As more productive EDA tools become available for FPGA, the FPGAs have become well suited for implementing complex processor designs with their high density logic and embedded components. Should CPU design be made FPGA synthesizable throughout product development as part of the pre-silicon validation process, it would bring significant benefit to exercise the design with much more simulation cycles. For example, booting an off-the-shelf operating system can require execution of 100 million to 1 billion instructions. It is impossible to run such long instruction sequence within a reasonable amount of time with today's RTL simulators, however it only takes under an hour with FPGA synthesizable designs. With additional optimizations in FPGA mapping and partitioning, FPGA synthesizable designs usually can be emulated at speed of 10's Mhz thus making it possible to work as software development vehicle before silicon becomes available [10].

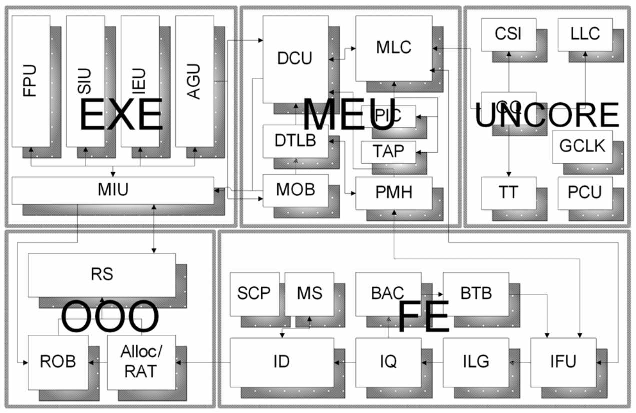


Figure 1: Intel Nehalem Microarchitecture [3].

While FPGA emulation has long been employed in SoC design and prototyping (SPARC [6], MIPS [7], ARM [9]), it is only recently that FPGA synthesis is taken into account for modern PC-compatible CPU designs. In [11] and [17], the Pentium processor and the Atom processor were made synthesizable to a single FPGA. However, compared to Atom, the Nehalem core requires roughly 4x more FPGA capacity. Due to this size increase, multiple-FPGA partitioning must be employed for the Nehalem core requiring time multiplexing of wires [2] between FPGAs and various partitioning tools [1] and techniques [5, 12].

2.2 The Intel Nehalem Processor

The Intel Nehalem processor is a 64-bit multithreaded processor with an aggressive out-of-order pipeline. Nehalem includes a 32KB L1 data cache, a 32KB L1 instruction cache and a shared 256KB L2 cache. Figure 1 shows the layout and clusters making up the Nehalem core. Four clusters make up the Nehalem core

FE	Frontend to fetch bytes, decode instructions
EXE	Floating point and integer execution
OOO	Out of order resource allocation
MEU	Memory execution unit (load/store handling)

The FE cluster fetches instructions, decodes the x86 instructions into an internal micro-op (uop) stream, and queues those uops for execution downstream. Branch prediction is performed in FE as well. The OOO cluster takes uops and schedules their execution to the various reservation stations. The EXE cluster holds all the ALUs and floating point execution units and there is highly optimized and specialized circuitry to complete these computations. The MEU handles all loads and stores from the shared L2 cache, known as the MLC (midlevel cache). The MEU is the sole interface to the Uncore. It contains many other miscellaneous components of the Nehalem processor including the interrupt controller and the TAP (test and access port).

The Nehalem Uncore, also shown in Figure 1, connects the cores to each other, holds the Last level cache, and contains an on-die memory controller. In this paper, our focus is on

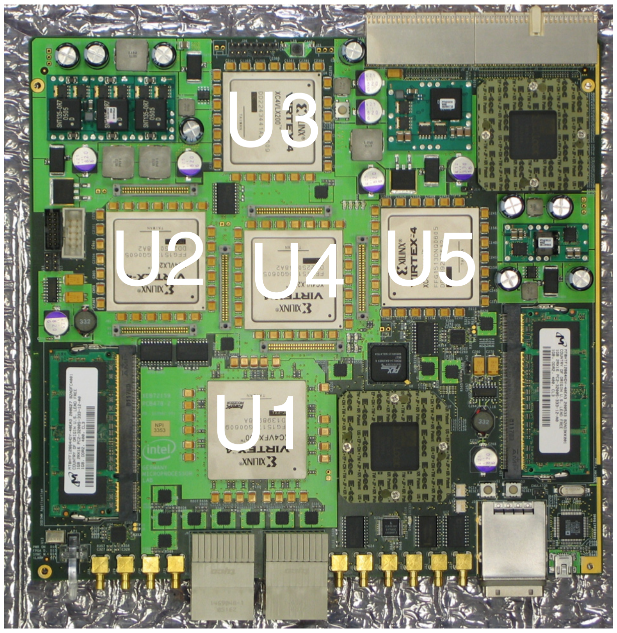


Figure 2: One MCEMU Board with Five FPGAs.

Table 1: MCEMU board FPGAs.

Name	FPGA	LUTs	BRAMs
U1	Virtex-4 FX140	126,336	552
U2	Virtex-5 LX330	207,360	576
U3	Virtex-4 LX200	178,176	336
U4	Virtex-5 LX330	207,360	576
U5	Virtex-4 LX200	178,176	336

the four core clusters in the Nehalem. The Uncore cluster is outside the scope of this paper.

2.3 The Many-Core Emulation System

The Many-Core Emulation System (MCEMU), is the emulation platform we targeted for this work. MCEMU is an FPGA emulation platform developed at Intel [13]. An MCEMU system consists of a series of identical rackable custom boards, each holding five FPGAs. Table 1 lists the name, type, and key resources for each of the five FPGAs, while Figures 2 and 3 show a single board and a full rackable system respectively. To expand capacity beyond five FPGAs, multiple boards are interfaced together using the Xilinx RocketIO high-speed serial transceivers connected by external cabling.

Within a single MCEMU board, board traces wire input pins on one FPGA to output pins of another, leading to a fixed number of physical wires between each FPGA pair. While the number of physical wires connecting two FPGAs is fixed and small, any arbitrarily large number of logical signals can be sent across the physical wires by time division multiplexing (TDM) using muxes at the sending FPGA and demuxes at the receiving FPGA. A greater ratio of logical signals to physical signals requires more time steps for TDM, and thus lowers emulated frequency.

Because of varying resources among the FPGAs and fixed physical traces on the boards not all FPGAs have direct access to all board level features. For example, each board contains 1GB DDR DIMM accessible by FPGA U1 and a 1GB DDR DIMM accessible by U5. Access to these DIMMs by logic within another FPGA would need to be routed through U1 or U5 to reach the appropriate DIMM. Similarly, only U1 contains the RocketIO transceivers that interface over the cabling. Therefore, signals destined for another board must pass through the U1 FPGAs on both the sending and receiving board.

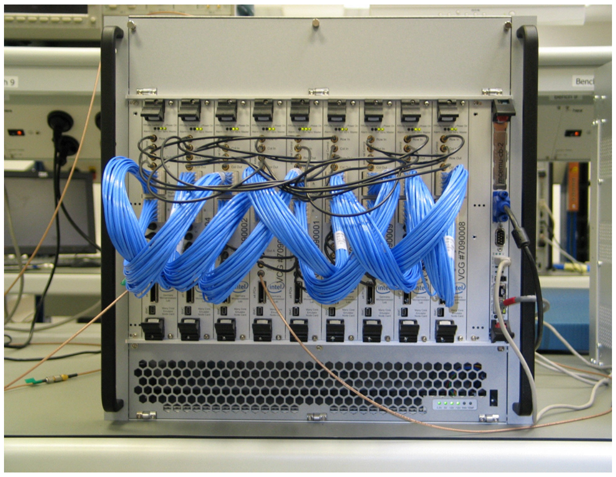


Figure 3: One MCEMU System with Nine Boards.

In addition, the number of physical pins interconnecting pairs of FPGAs is neither uniform nor symmetric. The MCEMU synthesis flow includes a sophisticated interconnect generation tool that when given a set of interconnected netlist modules generates and configures the TDM multiplex and demultiplex logic to properly connect the modules over the appropriate physical interconnects (intraboard traces and interboard cables). In the MCEMU flow, partitioning a large netlist into multiple modules (each suitable size for 1 FPGA) can be done either manually or with varying level of automation through partitioning tools.

Like most FPGA synthesizable designs, the choice of the emulator platform can affect the particular strategy to partition the design and interface the design to the memory system. In the Atom synthesis project, the platform was a single FPGA emulator that fits in a Pentium CPU socket. It was necessary to build a bridge between the Atom processor core and the Pentium front-side bus so as to allow the emulated Atom core to communicate with the memory and I/O resources on the motherboard. Similarly, with the MCEMU platform, which has on-board DDR memory, we also need to build a bridge between the Nehalem core and a DDR controller so that the emulated CPU core can boot from the OS image and execute code, all resident in the DDR memory. The original OS image and workload can be updated by the host CPU board on the MCEMU.

When a design is ready to be run on the MCEMU it is loaded on the FPGA boards by a control/debug host board that sits along side the FPGA boards. The host board is a full x86 computer with hard-disk and network interface and runs Linux. A Linux application running on the host board can program and reset the MCEMU FPGAs, write to control registers inside the FPGA, read and write the MCEMU DRAM DIMMs, and control the emulation clock all over the shared cPCI bus. As we show in Section 4.4, this built-in hardware/software interface can be a powerful tool for system bring-up and verification.

3. SYNTHESIS

Nehalem is designed in SystemVerilog, with wrapper Verilog code. Finding a tool that can parse the design is therefore of primary importance. There are actually only a few FPGA frontend synthesis tools that can parse SystemVerilog. While many tools do support a subset of SystemVerilog, from our experience, there are usually certain features of the language that either cause the tools to report syntax errors, die silently, or synthesize netlists incorrectly.

Even though Synopsys ceased its development and support of DC-FPGA [15] in 2005, it is the only tool that can correctly parse all SystemVerilog features used in Nehalem's RTL. Since DC-FPGA supports these features, we are able to minimize the code changes necessary to build netlists. With fewer modifications we must make to the Nehalem codebase, we are less prone to introduced bugs.

Ultimately, however, some source code modifications were still necessary due to some deficiencies in this tool. DC-FPGA works well enough for creating netlists from the SystemVerilog. However, as a discontinued tool, DC-FPGA sometimes is prone to create erroneous circuits from the given RTL. This was observed in the synthesized Atom core, and was observed in the Nehalem synthesis with new SystemVerilog constructs. For example, a certain XOR macro within the Nehalem execution cluster is not mapped correctly to the Xilinx FPGA architecture. We are able to discover these bugs as described later in Section 3.4, and replace the code snippet with a Xilinx macro block. Additionally, some modules that did not synthesize correctly were synthesized using Synopsys Synplify Pro [16] if the code is syntactically similar to Verilog. EDA tools are slowly being patched to handle SystemVerilog constructs, and we predict that SystemVerilog will be fully supported in the near future.

The entire synthesis flow is shown in Figure 4. Once the netlist is produced by DC-FPGA, the compete Nehalem netlist is partitioned using Auspy's ACE compiler v5.0, and the final bitstreams are generated using Xilinx ISE 10.1.03. This section will focus on preparing the Nehalem codebase to pass through DC-FPGA synthesis.

3.1 Clock Gating

Modern processor designs all use gated clocks to save on power. With latch-based designs tolerant to clock-skew, a gated clock is quite effective in driving a low-power clock tree through the design. In fact, this gated clock can hierarchically pass down into the processor subsystems, each subsystem having its own additional enable signal, providing designers the ability to clock gate the circuit at many levels.

For FPGA synthesis, we need to separate the enable from the original clock. FPGA architectures rely on a global clock tree that is low-skew to drive the flip-flops within FPGA slices. Most EDA vendors provide clock-gating removal synthesis that can do this separation automatically. This separation of the clock and enable signals allows the free-running global clock to travel along the FPGA's dedicated low-skew

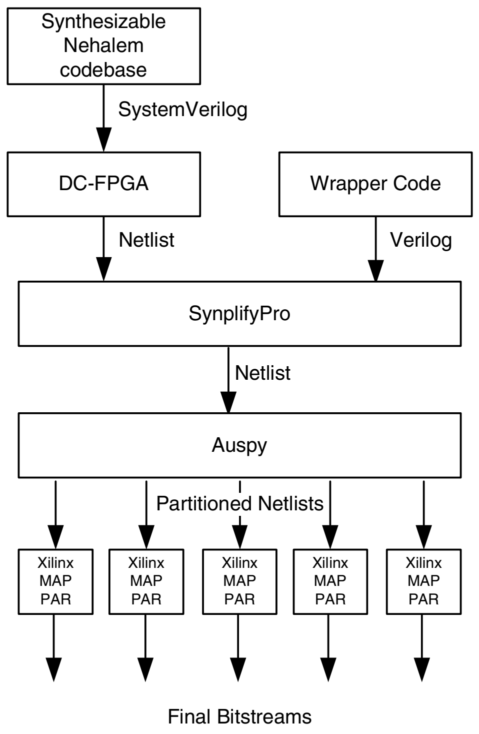


Figure 4: Toolflow: From Nehalem RTL to FPGA Bitstreams.

clock tree and have the enable signal travel over standard routing resources.

Every bit-wide clock is turned into a struct consisting of the global clock and its enable signal. The Nehalem RTL codebase consists of various macros that transform the clock as it is passed through the hierarchy of modules. Most transforms simply add another enable signal (e.g. powerdown or test port access enables) or invert the clock for double phase latch RAMs. The inversion clock macro is the only macro that modifies the global clock, while all other clocks affect the enable signal. To handle the clock inversion macro, we pass into all modules a global inverted clock that is driven from a DCM, not by actually inverting a clock signal in logic.

There are 400% more unique clock macros within the Nehalem codebase as compared to the Atom codebase. Our methodology of separating the clock and the enable is completely portable between the two synthesizable processors.

3.2 Latch Conversions

Due to the inability of DC-FPGA to correctly map transparent latches to the Xilinx primitives, we use two approaches to do latch conversion ensuring correct generation of netlists. The first approach is to directly instantiate Xilinx FPGA latch primitives (LDEs) forcing all latches as blackboxes during synthesis. However, the high count of latches in the Nehalem codebase makes applying this technique to all latches impossible. The backend place and route tools simply cannot meet the timing constraints in handling so many latches in the resulting netlist. The second approach is to convert latches to edge-triggered flip-flops – possible when the data and enable arrive one phase earlier than the clock edge. Most often this conversion is possible, however, in some instances, the data or the enable signal arrive late. Since the latches in Nehalem are instantiated in a macro form, we can detect this race condition while running the simulation and determine if data OR the enable is changing with the clock edge. If this behavior is seen, an edge triggered flip-flop conversion is incorrect and data will not propagated correctly through the latch. Therefore, we adopt a combination of both approaches. That is, for those latches with the input data or the enable signals ready before the leading clock edge, we convert those latches to flip-flops. For the remaining latches, we instantiate them directly to latch primitives.

Interestingly, the Nehalem core also has some latch structures that do not have an equivalent FPGA macro or clock-and-enable structure. For example, a LATCH_P macro is an inverted latch. By DeMorgan's theorem, the equivalent circuit is:

```
  LATCH  ==  CLK & ENABLE
LATCH_P  ==  ~LATCH
LATCH_P  ==  ~(CLK & ENABLE)
LATCH_P  ==  ~CLK || ~ENABLE
```

The latch can essentially be open by the inverted clock or the inverted enable signal. The latch can no longer be consistently opened by the clock since the inverted enable signal may change in either phase of the clock. In order to faithfully comply to the latching behavior of the original RTL, our solution is to use a `clock2x180` to produce a positive edged 2x clock that can capture data on each phase of the clock. This solution is feasible for most latches in the system, but leads to tight timing constraints, therefore it is used sparingly.

3.3 RAM Replacements

Latch and flip-flop RAM structures are used heavily within the Nehalem RTL. The latch RAMs are extremely power efficient, are tolerant to clock skew, allow time borrowing, and are amenable to clock-gated circuits. The flip-flop RAMs can be mapped to optimized cells in the ASIC backend flow, whereas the behavioral model is written in standard edge triggered SystemVerilog code. From looking at the RAM instantiations, it is clear that the memories that we end up replacing range in size and complexity across a range of parameters.

- **Size.** Memory structures within the Nehalem core range from several kilobytes down to bits. Small memories map better to distributed memory structures (granularity of `1bx16` or `1bx64` on Xilinx-V4 and Xilinx-V5 FPGAs per reconfigurable logic block), while larger memories map best to Xilinx block RAMs (granularities of 18Kb per RAM).

- **Read and write ports.** RAM structures found within Nehalem range from simple 1-read and 1-write port FIFOs, to highly complex banked register files. FPGA distributed memories natively can handle 1 shared read and write port (distributed memory) and up to 2 independent shared read and write ports (block RAMs).

- **Reset and set behavior.** The RAM structures in Nehalem have various flash reset, flash copy, and multiple-entry read behavior. Xilinx FPGAs have the connectivity available to connect these heavily interconnected and multiplexed structures. However, we want to take the effort to emulate more complex RAMs whenever possible, with the goal of better FPGA resource utilization.

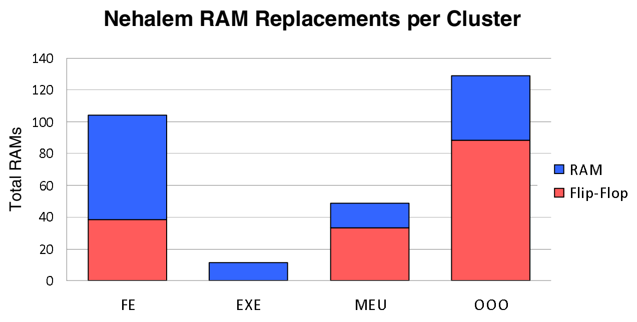


Figure 5: RAM Replacements for Nehalem Clusters.

As an example of RAMs that exhibits the behaviors described above, the out-of-order cluster holds many complex RAM structures. This cluster has many highly ported memories structures to implement the reorder buffer that vary in size from single to hundreds of bits per entry. The reorder buffer allows instructions to complete in any order but only affect machine state in program order. In an out-of-order machine, many instructions are in flight speculatively, are waiting for loads or stores to complete, or being used in various ALU operations. Instructions completing within the reorder buffer have to update other reorder buffers as quickly as possible to keep instructions retiring quickly. The RAM structures to hold the reorder buffer, therefore consist of highly ported memories. These RAMs also have flash reset behaviors, since a reorder buffer entry can be invalidated on a mispredicted branch.

We are able to emulate the complex read, write, and invalidate behaviors using the following techniques:

- **Entry Invalidations** We can keep a separate valid bit per RAM entry held in a flip-flop array. We can set this bit on a write and reset this bit on an invalidate. On a read operation, we can check this bit to determine whether to output the RAM contents or a reset value.

- **Multiple Write Ports** We can use 2 write ports per BRAM structure. For higher write ported memories, like found in the reorder buffer, we keep multiple copies of the RAM structures, each having 2 logical write ports attached to the physical write ports. The most uptodate write location is kept in another flipflop array that can be used to multiplex a read operation from the logical RAM structures.

- **Multiple Read Ports** We can also have 2 read ports per BRAM structure. We emulate higher read ports by again duplicating the BRAM structures by having each physical write port map to multiple write ports, where each logical BRAM structure is mapped to a multiplicity of read ports.

FPGA architectures are not suited to map these latch structures with complex behaviors. However, with our clock-gating methodology, we are able to separate out the read and write clocks from the enable signals and convert these RAMs all to flip-flop implementations. Once we are able to confirm all RAMs can be converted to a flip-flop implementation, we can translate the largest ones to either distributed or Block RAM implementations. Whenever these memory replacements are done using explicit Xilinx memory instantiations, these new instantiations are black boxed throughout the DC-FPGA synthesis step. Then later in the FPGA design flow, the generated memory netlists targeting FPGA architectures can be dropped in for the final FPGA bitstream assembly.

We observed 300 instances of these latch RAMs within the Nehalem code base and were able to convert them all to flip-flop RAMs, distributed memory RAMs, or BRAMs. Figure 5 shows the breakdown of how these RAMs were converted for each cluster. This number of RAMs is 8x over the number of RAMs seen in the Atom codebase. The synthesizable Atom core also had low count read and write ported RAMs structures, where in Nehalem, extremely high count write/read RAMs were observed in several instances. Again within Nehalem, the out of order cluster proves to hold a high count of RAM instantiations. The frontend cluster holds complex branch prediction RAMs, is multithreaded, and can decode multiple instructions in parallel. For this reason, FE holds a high count of smaller RAMs, with complex behavior.

3.4 Verification Methodology

With all these changes to the Nehalem codebase, special care has to be taken that our code does not break the functionality of the various clusters. The Nehalem model comes with a rich regression test environment with a large number of both full-chip and unit-level regression tests. These tests check not only for successful completion of the test case, but additionally instrument the code within the RTL, monitor produced output files, and poke and peek signals throughout the design.

Unexpectedly, due to the nature of the RTL changes necessary to for FPGA synthesis, such as converting the RAMs and converting 1-bit clock signals to clock structures, these regression tests frequently fail to execute unmodified due to non-existent or renamed signals and clocks that are no longer accessible in bit operations. Full-chip regressions are less invasive and more likely to continue working with minimal modifications, but take a significant amount of time to execute (on average 6 hours). Further, the full-chip regressions also interact with the Uncore code, which shares some macros and modules with our converted clusters, leading to naming mismatches. Given that most FPGA-related RTL changes are highly localized and only require changes to individual files, we used the following methodology for validating such changes, which yields both a rapid turnaround time on simulation execution but can be employed without requiring any changes to existing regression tests.

1. Modify the original Nehalem RTL to log all input and output values on every phase for a given target of interest (a full cluster or smaller RTL component)

2. Execute an existing full-chip regression test to generate the signal trace of interest

3. Modify the FPGA-synthesizable RTL to feed the logged inputs into the target of interest on each simulated phase, and check that produced outputs match those logged to the trace. Comment out all other RTL (e.g. other clusters) to speed compilation and simulation time

4. Simulate the reduced Nehalem design to test the correctness of the FPGA-synthesizable RTL changes

Additionally, we track every latch replacement's input and output signals. Within a simulation, a latch macro will report if its input data is not being latched correctly by our ported code. This is easy enough to do, by having the original code in place and comparing outputs. By having this fine grained verification in place, we can quickly see a bug and replace that latch macro with a Xilinx native latch. It is bad for timing to use too many latches, but the Xilinx tools can handle a few of them. Also we are running the FPGA at a relatively low clock rate and the tools can handle placing and timing some latches.

We have made extensive use of this strategy in this project. Doing so significantly reduces the time to verify a particular RTL change (e.g. one minute to recompile the EXE cluster compared to 10 minutes for the full model and three minutes to simulate a simple test on EXE compared to one hour for the full-chip) but also gives a more rigorous validation as any deviation from the baseline behavior, even changes which might not cause a particular test to fail, will be detected. We have written scripts to automatically insert the necessary trace generation and trace consumption code (steps 1 and 3 above), and no manual RTL changes are necessary to employ this methodology. This methodology was not used for the FPGA synthesizable Atom core. With a small codebase, the Atom core can run full simulations within minutes compared to Nehalem taking one hour for short system level tests. Therefore, this strategy is extremely beneficial for large circuits and scales extremely well as the design grows.

Additionally, this methodology can also be applied on FPGA to synthesized code, in order to validate that the synthesis flow and tools have produced the correct output. Inputs to the targeted module can be driven either by specialized software control or by an embedded ROM. We can typically synthesize a bitfile for testing an individual Nehalem RTL file in approximately 15 minutes, significantly faster than the time necessary to synthesize a full cluster or the full design.

Individual modules can be synthesized and tested on-FPGA with a similar methodology in order to validate that the synthesis flow and tools have produced the correct output. The MCEMU hardware and software platform provide a powerful logic analyzer and injector capability which allows signals on individual FPGAs to be read or written under software control. Each clock phase, the inputs to the synthesized logic block are read from the associated trace file and provided to the corresponding logic block via this signal injection mechanism, and the outputs which had been generated on the prior clock phase, are read out are checked against the signal trace to identify any deviation from the expected behavior.

We can typically synthesize a bitfile for testing an individual Nehalem RTL file in approximately 15 minutes, significantly faster than the time necessary to synthesize a full cluster or the full design. Additionally, by ensuring each module was tested using this methodology in additional to

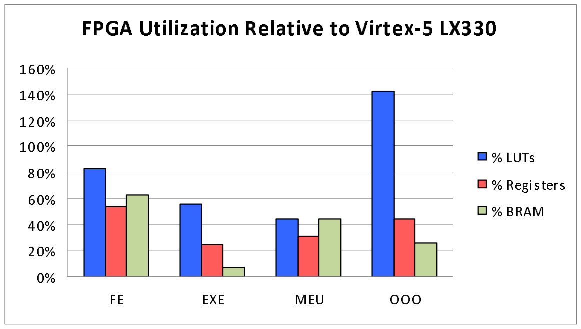


Figure 6: FPGA Resource Assessment of the Nehalem Clusters.

simulation testing, a handful of bugs due to tool misconfiguration and incorrect synthesis output were caught at an early stage, allowing fixes and workarounds to be quickly applied.

4. NEHALEM EMULATION IN MCEMU

Once our code changes have been verified through simulation and have a body of RTL ready for MCEMU, the netlists are taken and partitioned across the five FPGAs on one MCEMU board.

4.1 Initial Sizing of the Nehalem Clusters

With changes made to the clocking structure, RAMs converted, and the code verified, the various code bodies are combined. Preliminary synthesis results for the ported Nehalem code are gathered before partitioning the RTL across the five FPGAs. As shown in Figure 6, the clusters' FPGA utilization is presented. The synthesis tool targets all the clusters for the Virtex-5 FPGAs on the MCEMU platform, but of course some of the RTL will have to be targeted to Virtex-4 architectures. As can be seen, the Out-of-Order cluster clearly cannot fit on a single FPGA. Due to the high connectivity of the reorder buffers and the number of buffers themselves, the out of order cluster requires further partitioning.

4.2 Memory Interface

The standard Nehalem core connects to the Uncore through the MLC (midlevel cache). The Uncore at the interface to the MLC shown in Figure 7. This cut is necessary, as the FPGAs cannot fit the 256KB midlevel cache on the emulation platform or its the associated logic. Once that logic is cut out however, the necessary interfaces are created to our custom memory controller, which can communicate to the onboard DRAM. Any other signals that are driven by the Uncore must be emulated correctly such as clock synchronization and reset signals. This cut is similar to the synthesizable Atom core, where there the cut occurred at the L2 cache. The cut is chosen to maximize original functionality, but cut out the larger lower level caches and complex Uncore interfaces that are not inherently part of the processing pipeline.

This translation step is not trivial and the emulated interfaces are briefly described below.

- Memory Load / Stores. For memory operations in our emulated codebase, the Nehalem core is forced to allow only one outstanding memory request and translated to communicate with a standard DDR controller. Additionally, the bridge must respond correctly to data accesses that correspond to the cache coherency protocol. There are multiple memory requests types due to locks, read-for-ownership, and self-snoops that are handled as well.
- CRAB (Control Register Access Bus) Read/Writes. The CRAB bus is a distributed register file that allows a control registers to be read/written to by communicating over a ring. This ring has stops within the MLC and Uncore that are emulated.
- APIC (Advanced Processor Interrupt Controller) Access. The timer interrupt is the only functionality that must be emulated correctly. This timer is used for operating system functionality and must be emulated correctly. As the Uncore can run at a different clock-speed than the cores ondie, the Uncore clock synchronization signals are emulated as well. The functionality of the APIC timer can be verified in short tests.

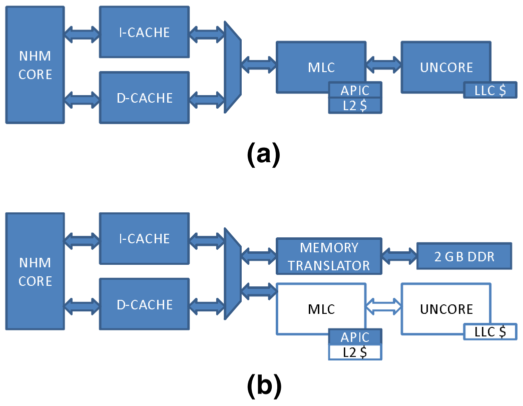


Figure 7: Nehalem Core to Memory Interface. (a) Original Design (b) Memory Translator.

4.3 Multi-FPGA Partitioning on MCEMU

Once the code changes are verified through simulation and a body of RTL is ready for MCEMU, the full design is partitioned into modules to target individual FPGAs on the MCEMU.

Before describing how Nehalem is mapped to the MCEMU emulator, its important to understand a fundamental difference between single- and multi-FPGA emulation. Although in single FPGA emulator it is the resulting critical path timing from the place and route tools that sets emulation speed, in a partitioned multi-FPGA design it is the critical path of a logical signals between FPGAs that sets the emulation speed. For a logical signal between 2 FPGAs, U1 and U3, this critical path depends on the degree of TDM sharing of the physical wires. If the TDM ratio is 10 logical signals per physical wire, the maximal emulation frequency is 1/10th the speed of the physical interconnect. In addition, if a logical signal must hop from U1 to U2 and then to

Table 2: Logical Connectivity and TDM Ratio between FPGAs.

	U1	U2	U3	U4	U5
U1	-	18	18	-	-
U2	24	-	18	21	-
U3	18	18	-	18	-
U4	-	21	21	-	24
U5	-	-	-	18	-

U3 in one emulation cycle, assuming both hops have TDM ratio of 10, the resulting frequency is cut by another factor of 2 because the logical signal must traverse both hops within one emulated cycle. Clearly, the frequency limit due to logical signal transmission quickly dominates the PAR frequency limit of the FPGA. As such, attaining a high emulation frequency becomes an exercise of mapping the logic across FPGAs in a way that minimizes the number of logical signals between FPGAs, minimizes the number of hops between the source and destination of logical signals and distributes logical signals to best balance TDM ratios. In other words a good partitioning of emulated logic should be found so that the partitioned topology most closely matches the emulator topology.

As shown in Figure 6, the cluster utilizations suggest that neglecting OOO's high LUT utilization, a cluster level partitioning would map very naturally to a single MCEMU board and minimize the number of logical signals traversing the on-board interconnect (i.e. logic within a cluster is more tightly connected than logic in two different clusters). Because FE and OOO are the largest clusters its clear to map them to our larger Virtex-5 FPGAs and map EXE and MEU to the smaller Virtex-4 FPGAs. As mentioned in Section 2.3, the number of physical wires between pairs of FPGA is not uniform. The pair U2,U4 and the pair U3,U5 have many more connecting wires than the other pairs of FPGAs. To best balance TDM ratios, more highly connected clusters are placed in the highly connected FPGA pairs. Analysis of the cluster level connections shows highest coupling between the pair EXE, OOO and the pair MEU, FE. This gives us the potential initial mappings of (FE$\rightarrow$U2, MEU$\rightarrow$U3, OOO$\rightarrow$U4, EXE$\rightarrow$U5) or (OOO$\rightarrow$U2, EXE$\rightarrow$U3, FE$\rightarrow$U4, MEU$\rightarrow$U5). In the end a high Block RAM utilization by auxiliary emulation logic on U5 (U5 is the central communication node that dispatches control/debug messages to other FPGAs) restricts us to mapping the lower Block RAM utilizing EXE to U5 and selects the former mapping above (FE$\rightarrow$U2, MEU$\rightarrow$U3, OOO$\rightarrow$U4, EXE$\rightarrow$U5). As mentioned above the OOO cluster is still too large for U4 (V5330). Here the resulting split occurs within the OOO at its sub-partition hierarchy. The OOO subclusters on U4 consist of the Reservation Station (RS), Register Alias Table (RAT), and Allocation (ALLOC), while the OOO's other cluster ReOrder Buffer (ROB) resides on U1 [3].

The Auspy ACE partitioning software is used to restructure the top level netlist using the given Nehalem netlist partitions. Because this methodology keeps the natural cluster-level partitioning, ACE's ability to automatically find good partitions is not used. The tool is still critically important though, as it allows us to pull lower hierarchy structures (e.g. ROB) up to top level entities. Without such a tool, this restructuring would be error-prone and tedious. In addition to using ACE to spit out netlists into multiple partitions, it can be used to route some internal signals to specific FPGAs. In particular, the memory controller signals are routed from MEU (U3) to the DRAM interface on U1, and internal architectural state (instruction pointer and architectural registers) and memory access signals is routed to U5 where auxiliary emulation logic records cycle-by-cycle traces of these signals into the other onboard DRAM module. After partitioning, the MCEMU interconnect configuration tool (see Section 2.3) runs to multiplex the interconnecting logical signals over the available physical wires. The end result of the partitioning and interconnect generation is an synthesizable fabric with the connectivity matrix and TDM ratio shown in Table 2. Empty entries show where no physical direct connection exists, though logical connection may still occur by using 2 or more adjoining physical direct connections. As shown in Table 2, the generated interconnect has a TDM critical path of 24 in the path from U2$\rightarrow$U1 and the path U4$\rightarrow$U5. These large TDM ratios are a direct result of the high number of logical signals passing between those FPGAs. The U4$\rightarrow$U5 connection is the signals from the OOO cluster to the EXE unit, which as described above, are very tightly coupled clusters. Interestingly, the connections between U2$\rightarrow$U1 is actually not dominated by the FE unit talking to the DRAM interface or to the OOO cluster, but is instead heavily utilized by signals passing through U2 from the OOO sub-clusters.

From this TDM data, potential emulation frequency can be calculated. The physical interconnect is able to run at 10ns period. It therefore requires 240ns (24 TDM cycles) for all logical signals to complete 1 hop, and the emulation period could be 240ns. Because there are paths that need to cross 2 FPGA hops with in a single emulation cycle, we need to exercise these paths 24 TDM cycles at least twice within every emulated cycle. This sets the maximum emulation period then to 480ns. If we could guarantee that all signals crossing the interconnect fabric could only change at the emulation clock edge then 480ns would be the final emulation period. This however is not the case, due to the design having registers clocked on `clk2x` and various level-sensitive latches. With this added clock and existing latches, there is a possibility that logical signals crossing the interconnect need to change before any edge of `clk2x`. To allow for this possibility (and maintain logical equivalence to the unpartitioned design) we need to allow for all logical signals to complete the 2 hops within each phase of `clk2x`. This means that the actual emulation period (clk1x) needs to be 4 x 480ns. This results in an emulation clock frequency of 520 KHz.

Interestingly, this partitioning step can be a one-time cost, barring any single FPGA running out of logic resources. Once the partition step is done, blackboxed Nehalem clusters can be synthesized to EDIF files and quickly linked into the bitstream generation step to create a new revision of the FPGA synthesizable design. This ability to drop in new synthesized clusters allows us to turn around a new design within the time it takes to synthesize and place and route a single FPGA (i.e. as opposed to synthesizing all the clusters, running a partition step across the entire circuit, and place and routing the individual FPGAs).

We take the five resulting netlists (each netlist includes the emulated cluster wrapped with the generated auxiliary interconnect logic) and push them through the typical Xil-

Table 3: FPGA Utlization after Xilinx Map.

	LUTs (%)	BLOCK RAM (%)
U1-ROB	81	62
U2-FE	87	83
U3-MEU	75	75
U4-RAT/Alloc/RS	89	0
U5-EXE	89	55

inx backend flow of ngdbuild, map, and par. Table 3 shows the resulting post-map resource utilization. All FPGAs are heavily loaded, but still meet timing because only the high-speed interconnect wrapper must run at 100MHz, and the emulated Nehalem clusters are relaxed to meet only 520KHz. With the set of bitfiles complete, we use the MCEMU control/debug application (Section 2.3) to load the bitfiles and write memory images into the DRAM DIMM accessed by U1. Similarly, we use the MCEMU control/debug applications to pull out trace data from the DRAM DIMM accessed by U5.

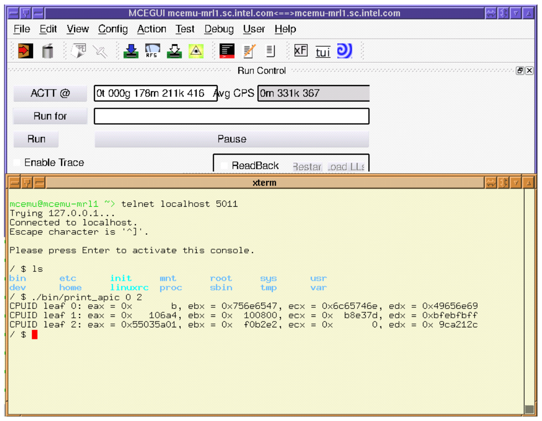
4.4 Nehalem Verification on MCEMU

As described in Section 3.4, individual modules and even entire clusters can be separately synthesized and tested on FPGA by feeding a trace of inputs to a synthesized chunk of logic. With slight modifications, this technique can be extended to similarly verify that the fabric which multiplexes the signals between the different clusters operates correctly. An initial partition of the Nehalem core is produced, such that each FPGA holds a portion of the core logic, but without any connectivity between FPGAs. That is, each ”island” of logic is fully standalone, with inputs to each FPGA controlled entirely through software which reads inputs and validates outputs from a trace. This verifies that, in isolation, the complete design produces behavior that matches 100% with simulation.

Following this, in a piecewise fashion, connectivity between the different clusters is progressively established, allowing signals to flow between the different FPGAs rather than being read from a trace file. Ultimately only those inputs to the Nehalem core top level (e.g. clock, reset) are read from the trace. Note that even in this configuration the outputs produced at each FPGA can still be read out and compared against the expected output recorded in the trace.

In fact, this technique can be (and was) applied even before the complete Nehalem core is FPGA-synthesizable, allowing the remaining portions of the design to be tested. Logic which cannot be currently synthesized to an FPGA due to, for example, capacity restrictions, can be freely removed from the design. Its functionality is provided, instead, by the MCEMU software, which injects the outputs of that missing logic by reading from a separately captured trace. This technique was used both in the early stages of this work, when accommodating the extremely large OOO cluster (which exceeds the capacity of a single Virtex-5 FPGA as shown in Figure 4), as well as in the later stages when the memory interface described in Section 4.2 was being written.

The correctness of the Nehalem core can be verified through these trace-based methods for core reset and simple workloads such as computing a Fibonacci number, but this mechanism is too slow for more involved workloads. Therefore, an


MCEGUI mcemu-mrl1.sc.intel.com<==>mcemu-mrl1.sc.intel.com
File Edit View Config Action Test Debug User Help
Run Control
ACTT @
Avg CPS
Run for
Run
Pause
Enable Trace
ReadBack
Restar
Load LLs
xterm

0t 000g 178m 211k 416

0m 331k 367

```
mcemu@mcemu-mrl1 ~> telnet localhost 5011
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

Please press Enter to activate this console.

/ $ ls
bin       etc       init      mnt       root      sys       usr
dev       home      linuxrc   proc      sbin      tmp       var
/ $ ./bin/print_apic 0 2
CPUID leaf 0: eax = 0x       b, ebx = 0x756e6547, ecx = 0x6c65746e, edx = 0x49656e69
CPUID leaf 1: eax = 0x   106a4, ebx = 0x  100800, ecx = 0x  b8e37d, edx = 0xbfebfbff
CPUID leaf 2: eax = 0x55035a01, ebx = 0x  f0b2e2, ecx = 0x       0, edx = 0x 9ca212c
/ $ 
```

Figure 8: Screenshot of Synthesizable Nehalem Core Booting up Linux.

EIP (instruction pointer) and register value tracing mechanism was also added. The original Nehalem RTL already has instrumentation code which is capable of generating a full log of retired instruction addresses and architected register values, which, used in conjunction with a separate x86 functional model, validates the correctness of the RTL in simulation. This capability can be added to the synthesized core by simply removing the `ifdef` which guard this code, and routing these signals to the dedicated MCEMU trace collection hardware described in Section 2.3. This tracing mechanism proved invaluable in fixing those final bugs in the memory interface which were not identified through simulation.

5. RESULTS

Our FPGA-synthesizable version of the Intel Nehalem core correctly preserves all instruction set and microarchitectural features implemented in the original Nehalem core. These include the complete microcode ROM, full capacity L1 instruction and data caches, SSE4, Intel64, Intel Virtualization Technology, and advanced power modes such as C6. As expected, the synthesizable Nehalem core is capable of executing the rich variety of legacy x86 software.

Figure 8 shows a screenshot of Nehalem core booting a version of Linux on the MCEMU. A simple program is shown to execute the CPUID instruction and display the result, revealing that the emulated CPU is indeed a Nehalem core.

As an example to illustrate microarchitectural difference between two families of Intel Architecture designs, Figure 9 shows a performance comparison the out-of-order Nehalem core and the in-order Atom core, both synthesized to the same MCEMU platform and to the same frequency. The five benchmarks represent, respectively from left to right, different optimizations of a handle-optimized compute-intensive Mandelbrot fractal computation using (1) x87 single-precision (2) SSE3 single precision (3) x87 double precision (4) SSE3 double precision, and (5) an integer workload designed to stress a processor’s out-of-order capabilities. In all cases these results show significant performance advantages for the Nehalem core, with speedups ranging from 1.8x to 3.9x over the Atom processor.

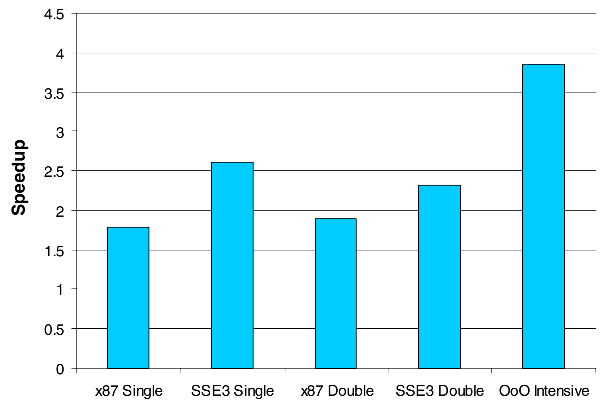


Figure 9: Nehalem vs. Atom: Performance Comparison of Microbenchmarks.

6. CONCLUSIONS

In this paper we have presented our experience in making the Intel Nehalem processor core FPGA synthesizable and partitioning the design on a multi-FPGA emulator platform. We also present the methodology for taking various complex constructs, mapping them efficiently to FPGA resources. The debugging methodology seamlessly employed from RTL simulation through bitfile emulation proves to be vital in ensuring high productivity. To our knowledge, this is the first time that a full-featured state-of-the-art out-of-order x86 processor design has been successfully made emulation ready on the commodity FPGAs using the existing EDA tools. With our previous work to make the Intel Atom core FPGA synthesizable, this latest milestone with FPGA-synthesizable Nehalem provides yet another cost-effective approach to improve the efficiency and the productivity in design exploration and validation for future x86 architectural extensions and microarchitectural optimizations.

7. ACKNOWLEDGMENTS

We would like to thank Joe Schutz, Steve Pawlowski, Justin Rattner, Glenn Hinton, Rani Borkar, Shekhar Borkar, Jim Held, Jag Keshava, Belliappa Kuttanna, Chris Weaver, Elinora Yoeli, Pat Stolt and Ketan Paranjape for the productive collaboration, guidance and support throughout the project.

In addition, we thank the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper.

8. REFERENCES

[1] Auspy. ACE Compiler. http://www.auspy.com/.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer Aided Design*, 16:609–626, 1997.

[3] B. Bentley. Simulation-driven Verification. Design Automation Summer School, 2009.

[4] J. Casazza. First the Tick, Now the Tock: Intel Microarchitecture (Nehalem). *Intel Corporation*, 2009.

[5] W.-J. Fang and A. C.-H. Wu. Multiway FPGA Partitioning by Fully Exploiting Design Hierarchy. *ACM Trans. Des. Autom. Electron. Syst.*, 5(1):34–50, 2000.

[6] J. Gaisler. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[7] M. Gschwind, V. Salapura, and D. Maurer. FPGA Prototyping of a RISC Processor Core for Embedded Applications. *IEEE Transactions on VLSI Systems*, 9(2), April 2001.

[8] Intel Core i7-800 and i5-700 Desktop Processor Series. download.intel.com/design/processor/datashts/322164.pdf, 2009.

[9] D. Jagger. ARM Architecture and Systems. *IEEE Micro*, 17, July/August 1997.

[10] H. Krupnova. Mapping Multi-Million Gate SoCs on FPGAs: Industrial Methodology and Experience. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1236–1241 Vol.2, Feb. 2004.

[11] S. L. Lu, P. Yiannacouras, R. Kassa, M. Konow, and T. Suh. An FPGA-based Pentium in A Complete Desktop System. In *International Symposium on Field Programmable Gate Arrays*, 2007.

[12] W.-K. Mak and D. F. Wong. On Optimal Board-Level Routing for FPGA-based Logic Emulation. In *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 552–556, New York, NY, USA, 1995. ACM.

[13] T. Mattner and F. Olbrich. FPGA Based Tera-Scale IA Prototyping System. In *The 3rd Workshop on Architectural Research Prototyping*, 2008.

[14] Intel Microarchitecture, Codenamed Nehalem. www.intel.com/technology/architecture-silicon/next-gen/, 2009.

[15] Synopsys. DC-FPGA. www.synopsys.com/products/dcFPGA.

[16] *Synopsys FPGA Synthesis Reference Manual.* Synopsys, December 2005.

[17] P. H. Wang, J. D. Collins, C. T. Weaver, B. Kuttanna, S. Salamian, G. N. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang. Intel Atom Processor Core Made FPGA-Synthesizable. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 209–218, New York, NY, USA, 2009. ACM.

[18] *Virtex-4 User Guide, v2.3.* Xilinx, August 2007.

[19] *Virtex-5 FPGA User Guide, v3.3.* Xilinx, February 2008.