Intel® Atom™ Processor Core Made FPGA-Synthesizable

Perry H. Wang[1], Jamison D. Collins[1], Chris T. Weaver[2], Belliappa Kuttanna[2]
Shahram Salamian[2], Gautham N. Chinya[1], Ethan Schuchman[1], Oliver Schilling[3]
Thorsten Doil[3], Sebastian Steibl[3], Hong Wang[1]

Microarchitecture Research Lab, Corporate Technology Group, Intel Corporation[1]
Atom Processor Architecture, Mobility Group, Intel Corporation[2]
Germany Microprocessor Lab, Corporate Technology Group, Intel Corporation[3]
Contact: perry.wang@intel.com

ABSTRACT

We present an FPGA-synthesizable version of the Intel® Atom™ processor core, synthesized to a Virtex-5 based FPGA emulation system. To make the production Atom design in SystemVerilog synthesizable through industry standard EDA tool flow, we transformed and mapped latches in the design, converted clock gating, and replaced non-synthesizable constructs with FPGA-synthesizable counterparts. Additionally, as the target FPGA emulator is hosted on a PC platform with the Pentium®-based CPU socket that supports a significantly different front side bus (FSB) protocol from that of the Atom processor, we replaced the existing bus control logic in the Atom core with an alternate FSB protocol to communicate with the rest of the PC platform. With these efforts, we succeeded in synthesizing the entire Atom processor core to fit within a single Virtex-5 LX330 FPGA. The synthesizable Atom core runs at 50Mhz on the Pentium PC motherboard with fully functional I/O peripherals. It is capable of booting off-the-shelf MS-DOS, Windows XP and Linux operating systems, and executing standard x86 workloads.

Categories and Subject Descriptors

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

General Terms

Design, Measurement, Performance

Keywords

Intel Atom, FPGA, emulator, synthesizable core

1. INTRODUCTION

In this paper, we present an FPGA-synthesizable Intel® Atom™[7, 11] processor core running on a single Xilinx Virtex-5 based FPGA emulator that resides in the CPU socket on a classic Socket-7 based Pentium PC system. The Atom processor is based on an entirely new microarchitecture designed specifically for small and low power devices, while maintaining the Intel® Core™ 2 Duo instruction set compatibility that consumers are accustomed to when using a standard PC and the Internet. The Atom processor also features multiple threads for better performance and increased system responsiveness. As Intel's smallest and lowest power processor, the Atom processor has been used to power a broad spectrum of devices ranging from netbooks and nettops to Mobile Internet Devices (MIDs). As revealed recently[12], Moorestown, an Atom based system-on-a-chip (SoC) design, offers a glimpse to the future of the Atom processor core, which will likely be used in a diverse variety of system-on-chip (SoC) designs. As the level of silicon integration increases, it is highly desirable to emulate the full RTL design and exercise the full software stack as early in the design process as possible. In doing so, it is possible to drastically accelerate the SoC design exploration, enable early software development, and significantly improve the validation efficacy.

The emulated Atom processor core can run at 50Mhz, a speed essential to allow the Atom core to boot the standard off-the-shelf x86 OSes including MS-DOS, Windows XP and Linux with fully functional I/O peripherals. At this speed, it achieves several orders of magnitude higher emulation performance and lower cost than the traditional emulation techniques [17].

The key contributions of this paper are three-fold:

- We present our methodology to synthesize a fully featured, state-of-art modern x86 processor RTL design to an FPGA target.
- We demonstrate how CPU architects can use the FPGA-synthesizable Atom processor core to rapidly prototype new architecture features and rigorously evaluate micro-architecture tradeoffs by running off-the-shelf x86 operating systems and workloads on the emulated processor core on a standard PC platform.
- We share our learnings on how the synthesizable atom core and FPGA can be used to enhance productivity in debugging and validating a modern production x86 microprocessor design at low cost.

The remainder of this paper is organized as follows. Section 2 reviews related work and provides background information on the Intel Atom processor. Section 3 elaborates

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FPGA'09, February 22–24, 2009, Monterey, California, USA.
Copyright 2009 ACM 978-1-60558-410-2/09/02 ...$5.00.

our experience in making the Atom core RTL FPGA synthesizable. Section 4 describes our effort to modify the bus interface in the Atom core such that it can work as an emulated x86 CPU in a Pentium-class PC platform. Section 5 highlights a set of debugging techniques we use. Section 6 evaluates the functionality and performance of the synthesized Atom core on the FPGA emulator and Section 7 concludes.

2. BACKGROUND

2.1 Related Work

There are a plethora of synthesizable processor designs of various architectures commercially available today, including ARM [13], MIPS [6], SPARC [4], and PowerPC [8]. Other notable ones are Tensilica's Xtensa [5], a highly configurable 32-bit IP core, and Xilinx's Microblaze [23], targeted specifically for FPGA. OpenRISC [18] is an open source IP core maintained by a team of developers at OpenCores.

For the x86-compatible family, Lu et al. [15] previously presented an FPGA-synthesizable version of the Intel Pentium processor. Their work mapped the Pentium design of the early 1990's to the Xilinx Virtex-4 FPGA, and evaluated the performance of several micro-architecture optimizations. However the Pentium architecture lacks a significant number of modern Intel Architecture (IA) features. This paper for the first time discloses Intel's effort to implement an FPGA-synthesizable version of a modern x86 processor. Compared to the Intel Pentium, the Atom processor implements Hyper-threading and the complete Core™ 2 Duo compatible instruction set architecture (ISA) [10], including Streaming SIMD Extensions (SSE3) and Intel64, therefore presenting a much higher challenge to map the design to a single FPGA. This paper provides a detailed discussion of the debugging techniques used to validate the correctness of the RTL changes introduced to make the Atom core FPGA-synthesizable.

Additionally, the Atom processor is one of the first modern processor designs in the industry that is written entirely in SystemVerilog [19]. However, even today, none of the shipping FPGA synthesis tools available in the market can support all the SystemVerilog features used in the Atom design. In comparison, Pentium as well as most processor cores in the standard SoC IP block ecosystem have been written in classic Verilog or VHDL that are fully supported by the existing FPGA synthesis tools. In this paper, we share our experience on how to overcome this unexpected challenge. Our learnings can potentially assist the EDA tool providers to prioritize and accelerate their development of the missing SystemVerilog features required by modern CPU designs like the Intel Atom.

2.2 The Intel Atom Processor

The Intel Atom is a multithreaded processor with an in-order pipeline [7, 11]. It supports two-way Hyper-threading technology. Figure 1 shows the Atom processor floorplan, with the key components highlighted, including the following four core clusters:

- FEC: front-end cluster (with L1 instruction cache)
- FPC: floating-point cluster
- IEC: integer execution cluster
- MEC: memory execution cluster (with L1 data cache)

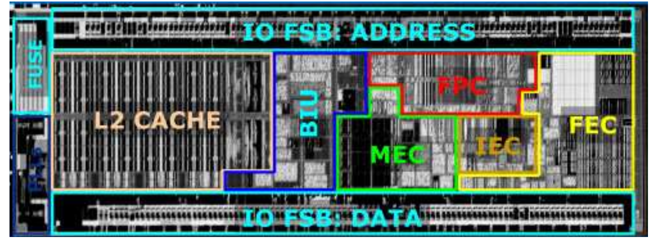

Figure 1: The Intel Atom Processor

In the Atom core, the FEC is responsible for performing instruction fetch. Instructions flow from the 8-way set-associative 32KB L1 I-cache into a set of fetch buffers, and from there into the decode unit. The processor's decode unit features two hardware decoders and one microcode decoder, and can decode up to two instructions per cycle. The FEC also includes the microcode ROM, which implements the more complex x86 instructions, e.g. string operations like `REPNZ`. The IEC and the FPC are responsible for integer and floating-point execution, respectively. On the data side, the address generation units in the MEC can perform two address calculations per cycle, and memory accesses are supported by the 6-way set-associative 24KB L1 D-cache.

While the Atom core consists of the four clusters listed above, all the remaining structures in the processor form what is referred to as the *uncore*. The uncore includes the L2 cache, fuses, pads, and the bus interface unit (BIU), which translates memory requests from the internal MEC format to the appropriate front side bus (FSB) protocol. The BIU cluster contains a 512KB on-die L2 cache with in-line error correction, and a FSB interface that can perform 400 MT/s or 533 MT/s.

Out of a total of 47.2 million transistors, the four clusters in the Atom core account for 13.8 million transistors. In comparison, the Pentium processor design used in [15] has about 3.1 million transistors.

2.3 EDA Tool Flow

At the time when this work was started, we evaluated several synthesis tools, such as Synplicity's Synplify Pro [21], Mentor's Precision [16], Xilinx's XST [22], and Synopsys' DC-FPGA [20], for our targeted Virtex-5 device. Since the Atom RTL was written entirely in SystemVerilog, of primary importance was to select a tool which could parse the design without requiring extensive syntax modifications. Even though Synopsys ceased development and support of DC-FPGA in 2005, DC-FPGA is the only tool that can parse all SystemVerilog features used in Atom design. For this reason we selected DC-FPGA as the synthesis tool. Unfortunately, DC-FPGA lacks some of the more advanced features, such as automatic conversion of clock-gating circuit to synthesis-friendly clock-with-enables, which are available in the more recent state-of-art synthesis tools like Synplify Pro. Therefore, manual modifications to the original RTL are required to enable resource-efficient synthesis with DC-FPGA. Once the netlist is produced by DC-FPGA, Xilinx ISE 10.1.03 is used to produce the final bitfile. However, it turns out that DC-FPGA in rare occasions can produce buggy netlist silently, even for a correctly developed RTL

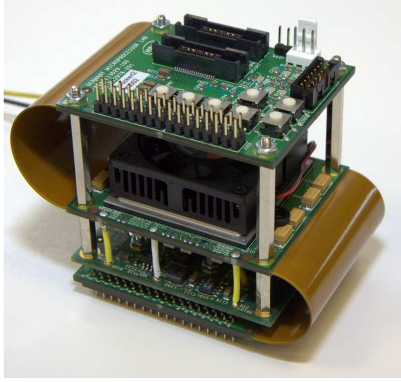

Figure 2: The FPGA Emulator

circuit that passes the RTL simulation. Since DC-FPGA is not supported by the vendor any more, once we root caused the bug in the netlist, we had to recode the RTL to work around the bug in the DC-FPGA itself. Fortunately, we only had to deal with just a couple of such situations in the Atom core design.

2.4 Virtex-5 Based CPU Emulation Platform

Shown in Figure 2, our emulation platform is an enhancement to the previous generation FPGA platform [15], which uses the Virtex-4 [24] LX200 FPGA. The current generation FPGA emulator employs the largest Xilinx device available – the Virtex-5 LX330 [25], which has 207,360 LUTs and 10,368 Kbits of block RAMs.

The FPGA emulator hosts all required logic, debug and supply circuitry besides the FPGA itself. The most challenging aspect of designing the emulator is the mechanical adaptation of the mainboard Socket-7 to the Virtex-5 FPGA. We decided to implement the FPGA emulator with a small board outline that does not exceed the Socket-7 outline itself. This allows us to support many different motherboards, including the single and dual socket types, regardless the available clearance area in the system.

Because of the limited board outline requirement, the conventional design with PCB-to-PCB connectors was not feasible and too expensive in terms of bare board costs. To tackle this problem, we adopted the *rigid-flex* technology – a multilayer PCB with both rigid and flexible layers laminated into a single package. It achieves the space and weight reductions by interconnecting via flexible substrates rather than multiple PCB-to-PCB connectors. The four-level vertically stacked PCBs were then populated with the components, and afterwards assembled and screwed in a zigzag fashion.

For the emulator, our work focuses on synthesizing only the Atom core for a single Virtex-5 FPGA, removing the original Atom's uncore, including the L2 cache and the BIU cluster. There are two reasons for this. First, the synthesizable Atom core is run on the older Socket-7 platform. In order for the Atom core to communicate with the emulation PC platform, a different front side bus protocol needs to be implemented, thus making the existing BIU cluster superfluous. Section 4 details a new veneer uncore we implemented to connect the Atom core to the Socket-7 FSB. Second, synthesized with DC-FPGA, the Atom core alone requires 85% of the total LUTs available in a single Virtex-5. If the BIU and the L2 cache were to be included in synthesis, the entire FPGA resources would have been overmapped beyond one Virtex-5.

3. SYNTHESIZING THE ATOM CORE

The original Atom source RTL was not developed with FPGA synthesis in mind. Our initial naïve attempts to directly synthesize the original RTL without any modifications were unsuccessful, and tools either failed to complete execution in any reasonable amount of time or required far more FPGA resources than were available on the Virtex-5. Therefore it is necessary to judiciously map the design to FPGA in a resource-efficient way. In this section, we present our mapping methodology, including RAM replacement, latch replacement, and clock gating conversion.

3.1 RAM Replacement

The Atom core contains a large number of memory structures to implement the various caches, buffers, queues that are prevalent in a modern microprocessor. For the most resource-efficient synthesis, such memory structures should be directly instantiated as FPGA RAM primitives, i.e. block RAMs or distributed RAMs. In the traditional ASIC design flow, for a given process technology node, these memory structures are usually implemented in behavioral RTL models and then custom mapped through various physical synthesis optimizations in the backend flow. Though these behavioral RTL models can be naïvely passed to the logic synthesis tools which may sometimes infer appropriate FPGA-specific memory structures, for most structures we observed an explosion in LUT utilization, far exceeding the capacity of a single Virtex-5 FPGA. Therefore, we found it necessary to replace the majority of memory structures in the Atom design with more resource-efficient FPGA-optimized implementations. As an extra benefit, doing so also drastically reduces synthesis time as well as significantly boosting the operating frequency of the resultant bitfile.

The memory structures we replaced range in size from as small as instruction queues with a handful entries to the large 32 Kbyte L1 instruction cache. Either single-port (1 read/write) or dual-port (2 read/write) RAMs were instantiated. None of the memory structures we replaced require more than 2 write ports. If more than 2 read ports are required, such as in the case of the integer register file, the required structure was replicated to increase the available read ports.

The Virtex-5 FPGA devices provide a large number of memory primitives, both block RAMs and distributed RAMs. The block RAMs are primarily geared towards deep memory structures, and the distributed RAMs, found in the `SliceM` primitive, are mainly targeted for shallow structures. In fact, the distributed RAM is built out of LUTs where each `SliceM` consists of four LUTs sharing one write port and three independent read ports [1]. In general, we replaced small and single-ported memory structures with the distributed RAMs, and the remaining with block RAMs. Judicious choice can further be made to flexibly deploy the block RAMs and distributed RAMs to optimally balance the FPGA resource utilization. For instance, when additional SoC IP blocks are to be integrated with the Atom core, tradeoffs can be made to use more block RAMs even for smaller structures, and spare the use of `SliceM` for distributed RAMs, thus freeing up LUTs for use in the other SoC IP blocks.

In some cases, due to the more complex behaviors that a memory structure has to support, making the memory structure synthesizable may be more nuanced than a straightforward task of mere RAM replacement. For example, to allow for write combining, the MEC maintains a multi-entry 64-bit wide memory structure to track the valid bits of the write buffers. Each valid bit corresponds to a byte within the cache line. When a store is merged into an existing write buffer entry, the MEC updates only the corresponding bit in the valid structure to 1. The custom logic is effectively `OR`'ing the updated bits in a vector into the memory structure.

```
ValidRAM [index][63:0] = ValidRAM [index][63:0]
                       | updatedVector[63:0];
```

This custom logic cannot be replaced with FPGA RAMs without significant RTL changes. We solved this issue by adding an extra read port to the memory structure. The RTL has been changed to read out the old value a cycle before the write, perform the logical `OR`, and then write the updated value back into the memory structure. Consequently, not only is the memory structure simplified but it can also be replaced with a standard FPGA RAM primitive.

Unlike some more recent EDA tools, DC-FPGA is not fully integrated with the latest Xilinx's backend flow. Therefore, we mapped the replaced RAMs explicitly by first declaring them as blackboxes in the DC-FPGA's synthesis flow. The replaced RAMs are later mapped in Xilinx's *ngcbuild* phase where different netlist sources are translated and consolidated into a single design format.

3.2 Latch Replacement

Optimized for a frequency range from 800MHz to 1.86Ghz, the original Atom design makes extensive use of latches to support time borrowing along the critical timing paths. With level-sensitive latches, a signal may have a delay larger than the clock period and may flush through the latches without causing incorrect data propagation, whereas the delay of a signal in designs with edge-triggered flip-flops must be smaller than the clock period to ensure the correctness of data propagation across flip-flop stages [3]. It is well known that the static timing analysis of latch-based pipeline designs with level-sensitive latches is challenging due to two salient characteristics of time borrowing [2, 3, 14]: (1) a delay in one pipeline stage depends on the delays in the previous pipeline stage. (2) in a pipeline design, not only do the longest and shortest delays from a primary input to a primary output need to be propagated through the pipeline stages, but also the critical probabilities that the delays on latches violate setup-time and hold-time constraints. Such high dependency across the pipeline stages makes it very difficult to gauge the impact of correlations among delay random variables, especially the correlations resulting from reconvergent fanouts. Due to this innate difficulty, synthesis tools like DC-FPGA simply do not support latch analysis and synthesis correctly.

Even though the Virtex-5 FPGA does provide a set of native latch primitives, DC-FPGA by default does not automatically infer such native latch primitives in the targeted FPGA. Instead, when a latch in the Atom design is synthesized, DC-FPGA outputs a synthetic latch construct that is essentially cobbled together from logic blocks which inherently exhibit race conditions. These synthetic latches are very difficult to achieve reasonable quality of results in place-and-route. Due to the large number of latches throughout the Atom design, should these latches be left untouched, it would be practically impossible to successfully place-and-route the design.

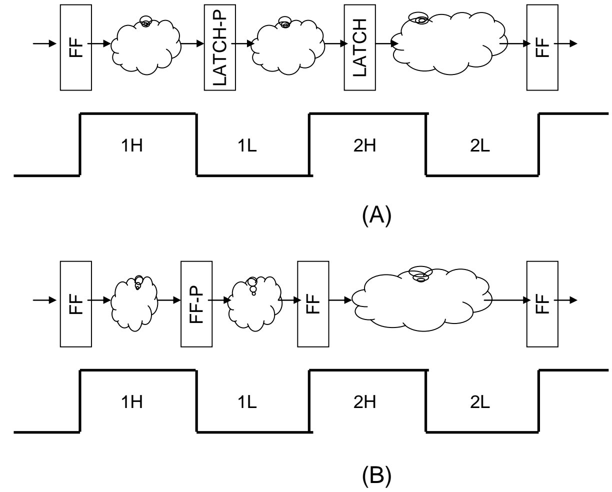

Figure 3: Sequencing with Latches and Flip-flops

To tackle the latch problem, we experimented with two approaches. One approach is to convert all level-sensitive latches to edge-triggered flip-flops. This would in effect remove all latches in the design. The other approach is to directly instantiate Virtex-5 latch primitives in the Atom RTL source code. There are interesting tradeoffs between these two approaches.

3.2.1 Latch-to-Flip-Flop Conversion

Since latches allow time borrowing, conversion of latches to flip-flops must be performed meticulously in order to meet the timing requirements in the original design. Nevertheless, since flip-flops allow register elements to change only on clock edges, such conversion can significantly ease the critical path timing analysis by the synthesis tools.

Figure 3(A) depicts a typical sequencing methodology implemented in the Atom design. The example pipeline stages have an edge-triggered flip-flop, followed by two level-sensitive non-overlapping latches, then by another flip-flop. For the purpose of illustration, each cycle is denoted with high phase (H) and low phase (L). Since this example requires two cycles to propagate the data through, the half cycles are marked 1H, 1L, 2H, and 2L accordingly. The positive level-sensitive latch is shown as `LATCH`, while the negative level-sensitive latch is `LATCH-P`. Similarly, `FF` is the positive edge-triggered flip-flop, and `FF-P` the negative edge-triggered flip-flop. In the figure, since `LATCH-P` is open during 1L, the combinational logic between the first `FF` and `LATCH-P` can borrow time into 1L. Similarly, `LATCH` is open in 2H, so the logic between `LATCH-P` and `LATCH` also borrow time into 2H. When time borrowing accumulates across multiple cycles, timing analysis can become quite challenging as explained earlier in this section. In Figure 3(B), `LATCH` is replaced with a `FF`, and `LATCH-P` is replaced with a `FF-P`. The clock now sharply demarcates the cycle boundary, making the timing analysis much easier.

There is an important issue that needs to be addressed in the LATCH-P replacement. It is related to the combined use of a gated clock and LATCH-P (or a gated complement clock with LATCH). Since LATCH-P is open when the clock is low, when such latchs are converted to flip-flops, the input data are captured at the falling edge of the clock. With a gated clock being a free-running clock AND'ed with an enable signal, a problem arises when the enable signal is activated (i.e. switching from low to high) during the phase when the latch is transparent (i.e. the free-running clock is low). In this scenario, the gated clock remains low during this transition so that no falling edge is ever seen when LATCH-P gets surreptitiously enabled. Therefore, the latch-converted FF-P would never be enabled under such condition. Our solution lies with the gated clock conversion, which will be discussed in Section 3.3. With all latch-converted flip-flops being controlled categorically by a free-running clock and an enable signal, the falling edge of the free-running clock can be observed every cycle. In addition, since the enable signal may change any time the latch is transparent, one must ensure that the valid enable signal to the latch-converted flip-flops arrives before the rising clock edge. Therefore, all the enable signals for the latch-converted flip-flops are computed half cycle early. Fortunately, the Atom RTL was rigorously coded and such modification only affects a handful of global SystemVerilog macros.

3.2.2 *Direct Instantiation of FPGA Latch Primitive*

The second approach is to directly instantiate Xilinx native latches. The methodology resembles the RAM replacement as described in Section 3.1; that is, all latches are instantiated as blackbox components first, followed by the integration of Xilinx native latches in the *ngcbuild* phase. Xilinx provides several variants of latch primitives, i.e. LD, LDC, LDCE, etc. Since latches in the Atom RTL are written as SystemVerilog macros, we can use the $bits system function to first determine the width of the data output, and then instantiate the number of latches based on the width.

For example, a simple latch can be declared as follows:

```
#define LATCH(out,in,clk)    \
  always_latch               \
    if (clk) out <= in ;
```

With direct instantiation of latches, the macro would be rewritten as follows:

```
#define LATCH(out,in,clk)                          \
  genvar i;                                        \
  generate                                         \
    for (i=0 ; i<$bits(out) ; i++) begin : ins     \
       LD ld(.Q(out[i]), .D(in[i]), .G(clk));      \
    end                                            \
  endgenerate
```

Since DC-FPGA treats latch primitives as blackboxes in the design, certain synthesis optimizations like constant propagation may be restricted thus resulting much longer synthesis time. The limitation to apply synthesis optimizations may also degrade the quality of the synthesis result. We observed that direct latch instantiation increased both the synthesis time and the place-and-route time by 2x over the latch-to-flip-flop conversion approach. Fortunately, the overall LUT usage was minimally affected, increasing only by 3%.

3.3 Clock-Gating Conversion

Global free-running clocks in Virtex-5 are distributed using dedicated interconnects specifically designed to connect and supply clock inputs to various resources in the FPGA. These clock networks are optimally designed to have low skew, low power, and improved jitter tolerance. Gating a global clock signal with a logic circuit forces the gated clock signal to traverse the much slower routing network, thus introducing significant skew. This further complicates place-and-route, and can lead to hold errors.

Optimized for low power, the Atom processor makes extensive use of clock gating throughout the entire design. The state-of-art EDA tools like Synplify Pro usually support automatic conversion from clock-gating circuits to clock-enabling counterparts as long as the clock signals are clearly indicated to the tools. Unlike clock-gating, circuits using clock-enables are far more friendly to the standard FPGA synthesis and place-and-route EDA tools, which can produce significantly improved timing results. Even DC-FPGA, a discontinued legacy tool, has an option, set_fpga_clock_gate_removal, to automatically convert a gated clock to a free-running clock with a clock enable. However, we found that the structure of local clock trees in the Atom design prevented this DC-FPGA feature from correctly converting most instances of clock gating in the design. Consequently, we had to resort to manual transformation of all instances of clock gating while maintaining identical functionality.

Manual conversion of clock gating involves separating the gating logic from the gated clock so that a single global clock tree is used for all gated clocks that were derived from the global clock. It is achieved by re-declaring every gated clock instance as a packed structure of two nodes, a free-running clock node and an enable node. Then the gating control is explicitly kept in the packed structure and propagated throughout the entire clock tree. The following simple example demonstrates how gated clocks can be thereby converted. We begin with the unmodified code of two hierarchical gated clocks derived from global_clock, and gated by enable_1 and enable_2 respectively:

```
node gated_clock_1, gated_clock_2;
gated_clock_1 = global_clock  & enable_1;
gated_clock_2 = gated_clock_1 & enable_2;
```

The clock declarations are first modified as follows:

```
typedef struct packed {
   node clock;
   node enable;
} clock_struct;
clock_struct gated_clock_1;
clock_struct gated_clock_2;
```

Then, for the first gated clock, instead of AND'ing the global clock with the first gate, the global clock and the gate are assigned to the two nodes in the packed structure respectively,

```
gated_clock_1.clock  = global_clock;
gated_clock_1.enable = enable_1;
```

and the gating logic is propagated to the second gated clock structure by

```
gated_clock_2.clock  = gated_clock_1.clock;
gated_clock_2.enable = enable_2
                     & gated_clock_1.enable;
```

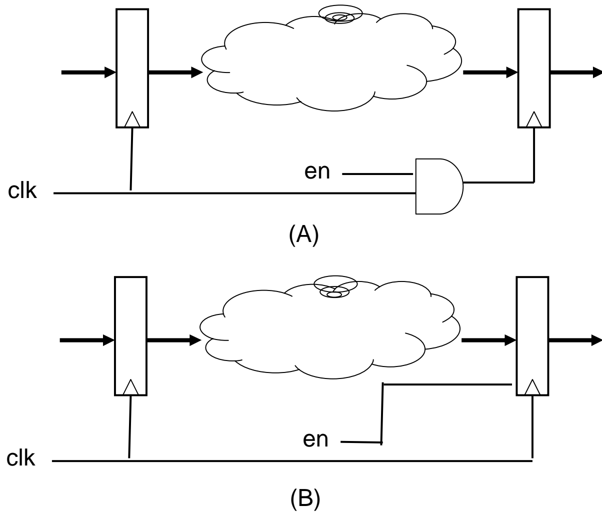

Figure 4: Clock Gating Conversion

Note that the clock node in the packed structure remains as a free-running clock node throughout the design. Figure 4(A) shows an example of gated clock logic. With the new clock structure, all register elements are modified accordingly as shown in Figure 4(B). That is, flip-flops that use the gated clocks are changed to enabled flip-flops. Such flip-flops are then clocked by the free-running clock and enabled by the gate signal.

Fortunately, thanks to the highly disciplined and consistent coding convention practiced throughout the Atom design, only a very small number of global SystemVerilog macros are used to code all the instances of the clock-gating circuits, so only a handful of the macros were modified.

4. EMULATION ON A PC PLATFORM

As discussed in Section 2.4, the Socket-7 motherboard uses the Pentium FSB protocol, which differs significantly from the modern FSB that the Atom processor uses. In this section, we describe a design adaptation to allow the synthesizable Atom core to interface with the Socket-7 FSB.

In the Atom processor, one of the uncore functions is to bridge between the Atom core and the processor's pins. Of particular importance is the uncore's responsibility to translate memory access requests from the internal format used by the MEC (i.e. memory execution cluster as described in Section 2.2) into the bus transaction format used by the Atom processor FSB and to manage those transactions in conformance to the FSB protocol.

In order to run the synthesizable Atom core in the Socket-7 based FPGA emulation board (as shown in Figure 2 and discussed in Section 2.4), we develop a veneer uncore interface to translate between the internal memory access request signals and the bus transaction signals for the Socket-7 FSB. As shown in Figure 5, the uncore veneer design consists of two components. The first component, the *signal translator*, translates memory access requests between the MEC format and the Socket-7 bus format. The second component, the *bus manager*, actually implements the state machine honoring the Pentium FSB protocol. More detail on these two components are as follows.

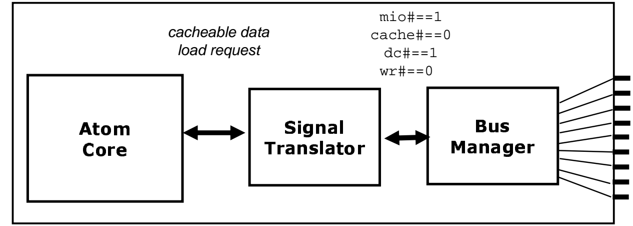

Figure 5: MEC-to-FSB Veneer Uncore

4.1 Signal Translator

The signal translator is responsible for translating the internal memory access request signals to the Pentium FSB signals and vice versa. For example, as shown in Figure 5, if a cacheable data load request is issued by the Atom core, the signal translator will send `mio#`=1 (memory access), `cache#`=0 (cacheable), `dc#`=1 (data), and `wr#`=0 (read) to the bus manager. The bus manager would then send out the new set of signals onto the emulator pins according to the Pentium bus protocol [9].

The veneer uncore supports three essential Atom core functionalities in the signal translator:

- handling all memory transactions, such as loads and stores, initiated by the MEC.
- handling all incoming platform interrupts to the IEC.
- transferring the control register values via Control Register Access Bus (CRAB).

The MEC is responsible for sending memory transactions to the outside world, and to receive and handle any return data. These transactions include instruction and data fetches, bus I/O cycles, and special cycles, such as interrupt acknowledgement, SMI acknowledgement and shutdown. Because x86 supports self-modifying code, the MEC must also issue self-snoop requests to invalidate potential stale instruction data. The signal translator in the veneer uncore takes appropriate action whenever such a request is observed.

The IEC handles interrupt requests, such as `intr`, `nmi`, and `smi`, received from the outside world. The signal translator latches these interrupts and routes them to the IEC through the appropriate internal signals. The CRAB is an internal interconnect which shuttles control register values through the major blocks in the processor. The signal translator is responsible for implementing those essential control registers that were originally implemented in the Atom uncore.

4.2 Bus Manager

The bus manager takes the translated signals and sends them onto the Socket-7 FSB. The Pentium FSB protocol [9] was designed to handle at most two outstanding memory requests. Figure 6 shows the implemented 6-state bus state machine. The first one (`Ti`) is the idle state. The next two (`T1`,`T2`) are for handling one outstanding memory request, and the other two (`T12`, `T2P`) are for the second outstanding memory request. As a special timing scenario, the last state (`TD`) accounts for an extra dead cycle to turn around between consecutive reads and writes.

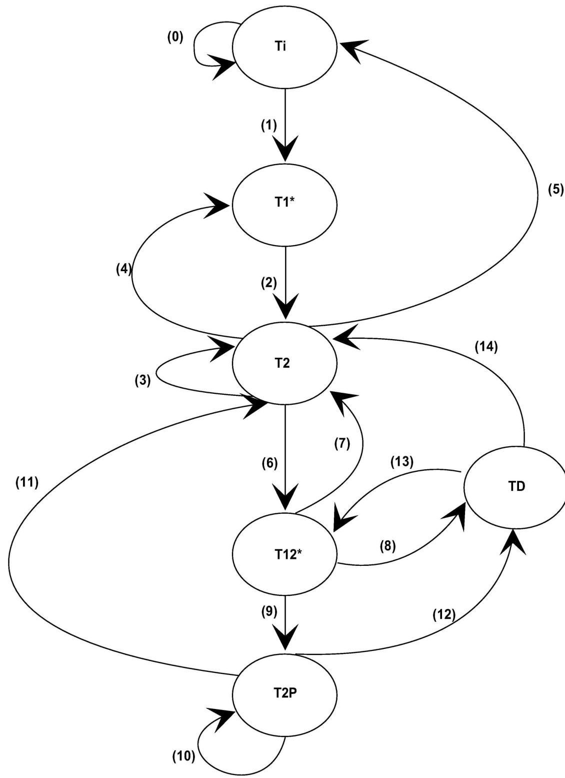

Figure 6: Pentium Bus Protocol

Continuing the earlier example, when the bus manager receives the translated signals of a cacheable data load request (`mio#`=1, `cache#`=0, `dc#`=1, `wr#`=0), it transitions from the `Ti` idle state to the T1 state while sending out the translated signals along with `ads#` to indicate that a new bus transaction is requested. On the next cycle, the bus manager transitions to state `T2` to wait for the fetched data by polling `brdy#`. Once `brdy#` asserts, the transition back to the `Ti` idle state is taken. Two counters are implemented in the bus manager. One is for the burst count when a cacheable request is received. The other counter is for the special lock and split requests.

In addition to honoring the Pentium bus state transitions, the bus manager also handles other related signals, such as `boff#`, `hold`, and `ahold`. Signals related to the interrupts (e.g. `nmi`, `intr`) are passed directly to the signal translator, which in turn forwards to the IEC.

5. DEBUG SUPPORT

Any time changes are made to the RTL it is possible to inadvertently introduce bugs. Throughout this project, we used three techniques to debug RTL: (1) RTL simulation (2) execution comparison (3) online debugging. RTL simulation is the first line of defense. For the x86 core, running legacy regression tests is the best way to check if any new RTL changes break x86. However, since RTL simulation runs at very slow speed on the scale of single digit hertz, the turnaround time from failure detection in batch regression to reproduction of bugs in interactive simulation is very long, even though the number of simulated cycles are rather short. Since FPGA emulation runs at 10's of MHz which is several orders of magnitude faster than the RTL simulation speed, it is possible to synthesize an early, albeit buggy version of RTL still in development, and then stress it by running a selection of the rich repertoire of legacy x86 software including BIOSes, OSes and applications on the FPGA emulator. Due to the fast emulation speed, this approach makes it possible to expose bugs much sooner and more bugs due to much more emulated cycles for a given wall clock period. Both execution comparison and online debugging techniques are built on the FPGA emulation approach. More detail on these techniques are presented next.

5.1 RTL Simulation

Our initial RTL checkout included a number of high quality regression tests providing good coverage over a large fraction of the x86 CPU and Atom core functionality. These tests could be run in a few hours across our batch-simulation pool. We ensured that these regression tests continued to pass after any RTL change and before the next code check-in. In some cases, even after RTL changes made to ease FPGA synthesis, it is necessary to add extra non-trivial RTL changes to ensure the code can still pass regression tests. For example, many regression tests explicitly are designed to probe specific signals in the design. Since the signals of our synthesizable RTL may have been modified (e.g. due to clock-gating conversion described in Section 3.3), regression tests need to be modified accordingly to probe the proper signals.

We used RTL simulation as the primary approach to debugging and validating the RTL modifications until the entire core RTL became synthesizable through the tool chain end to end, producing the initial bitfiles. Once the synthesized design became healthy enough to allow the core to fetch and later retire the first instruction during booting the BIOS image, we shifted focus towards online debugging techniques on the FPGA emulator.

5.2 Execution Comparison: Atom vs Pentium

Because Atom implements the latest x86 ISA that is compatible to the Intel Core 2 Duo, it is fully backward compatible with all prior generations of x86 processors. In other words, when executing the same code that an older generation processor can run correctly, a more recent processor should have identical architectural behavior. Therefore, comparing a trace of the retired instruction addresses and register values between a prior x86 processor, such as the Pentium, and our Atom design would provide an ideal checking mechanism; any divergence between the sequences would indicate a symptom of incorrect execution by the modified Atom processor.

Since both Pentium core and the synthesizable Atom core under development can be synthesized to run on the FPGA emulator at speed between 20MHz to 50MHz, this approach allowed for the quick detection of a number of bugs by assuming the Pentium as the golden reference of correct x86 behavior. The effectiveness of this methodology depends on the ability to present the Atom to the test software as if it were a Pentium. Various BIOSes and OSes explicitly use `CPUID` instruction to inquire ISA support in a given processor and perform ensuing tasks differently based on the enumerated capability. Compared to Pentium, the Atom processor supports much more architectural features, such as SSE3 and Intel64.

Current EIP / Instruction Count
Core Pause

VIO Console - DEV:3 MyDevice3 (XC5VLX330) UNIT:0 reset (VIO)

Bus/Signal	Value
pause	0
data collection EIP target	000E92C0
current EIP	000F21DD
instruction count	1246

Register read / write

VIO Console - DEV:3 MyDevice3 (XC5VLX330) UNIT:5 MyVIO5 (VIO)

Bus/Signal	Value
EAX	00001168
EBX	00000002
ECX	00013214
EDX	01C20500
write reg	
reg addr	00
reg val	12345678

Memory read / write

VIO Console - DEV:3 MyDevice3 (XC5VLX330) UNIT:8 memory inject (VI...

Bus/Signal	Value
byte enable	00000000
address out	0000000000
dbus out	878B6766A566F367
bus transaction mode	9
override	0
dbus in	F486FF7FF023EF7F

Figure 7: Online Debugging Interface for Atom Core

In order to apply the execution comparison methodology, we initially modified Atom's RTL and microcode to present identical behavior of the `CPUID` instruction to the reference Pentium core. This preserved identical execution between the two processors during BIOS and OS boot. Once satisfied with the stability of our design we restored the Atom's original `CPUID` behavior to re-enable the detection and use of those features we sought to temporarily disable.

A key benefit of this approach is to identify and root cause bugs without significant foreknowledge of where or when a symptom might occur. Once divergence between execution traces is detected, the nature of the problem usually became apparent after inspecting the instruction addresses and disassembly, and comparing values of architectural registers in that code region. In our experience, this approach was effective in diagnosing a variety of bugs, including those affecting either the core or the bridge logic, and those affecting the interaction between both.

5.3 Online Debugging

Analogous to debugging software program execution under *gdb*, online FPGA debugging allows a selection of signals of interest in a design to be inspected and modified while the synthesized design is being emulated in the FPGA. Online debugging of the synthesizable Atom core was significantly enhanced by a number of powerful general mechanisms that we introduced to the core RTL. For example, by exposing the architectural states and allowing control of key architectural mechanisms (e.g. suspending execution and injecting new instructions), we significantly reduced the number of re-synthesis iterations necessary to root cause a bug. If we suspected that a problem existed with, say, control register `cr0`, in order to inspect whether that register holds a corrupted value at some point in time, normally we would have to modify the design to route that signal to where we could directly observe it, and then re-synthesize the design at significant turn-around time cost.

Alternatively, the enhanced online debugging support provides built-in configurable circuits to observe architectural states like the values of the general purpose registers or to control key microarchitectural events. For example, we can pause and then re-direct the core to execute a `mov` instruction (i.e. `mov eax, cr0`) at the critical time, then the value of `cr0` would immediately become known, and a determination as to its status could be made.

The enhanced online debugging support utilizes the existing CPU functionality in implementing the necessary capabilities, which typically involve only simple RTL changes to multiplex signal values onto existing control and data paths. For example, performing a write into the register file involves overriding the write-enable signal and the destination register id, and providing the new data value.

Figure 7 shows a screenshot of the signal interfaces (via Xilinx Chipscope Pro) to the following online debugging enhancements.

- **Tracking Retirement EIP / Instruction Count**: In order to accurately record the progress of the Atom core, either for comparison with the reference Pentium or for use with other online debugging mechanisms, it is necessary to know the most recently retired instruction address as well as an accurate count of the number of retired instructions. The Atom core does not explicitly generate the EIP of each retired instruction, but it does record the information necessary to reconstruct this value when needed, such as in the event of an exception or a page fault. By replicating this logic, and leaving it always enabled, a continuous trace of retired EIPs and instruction counts can be generated, and possibly used to qualify additional debugging or signal gathering logic.

- **Pausing Processor**: The signal translator can be configured to decline any new memory requests which miss in the L1 cache. Doing so prevents the processor from making forward progress, and execution quickly comes to a halt, providing the opportunity to inspect or update processor state and to re-configure additional debugging mechanisms. This proved especially useful when disabling caching, as it provided an easy mechanism to gain almost immediate control of the processor.

- **Reading and Writing Registers**: Shortly after the processor has been paused, all execution stalls. At this time, the control and data paths to the register files are unused. By multiplexing in a small number of control

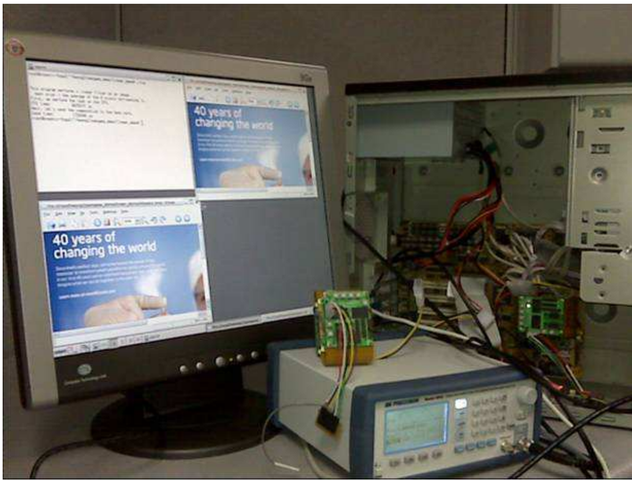

Figure 8: Emulation PC Platform Setup

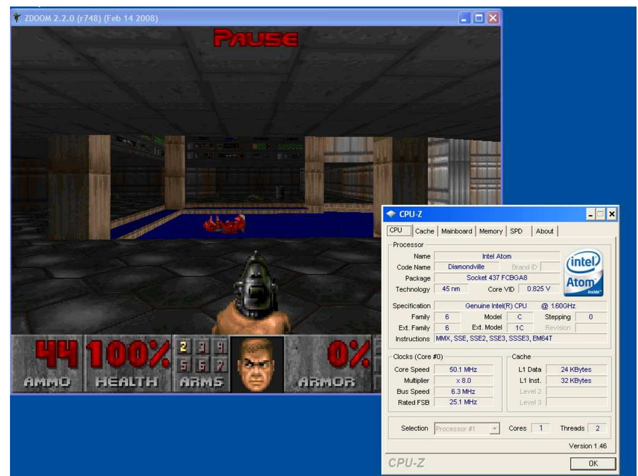

Figure 9: Screenshot of Synthesizable Atom Core running Windows XP, DOOM, and CPU-Z

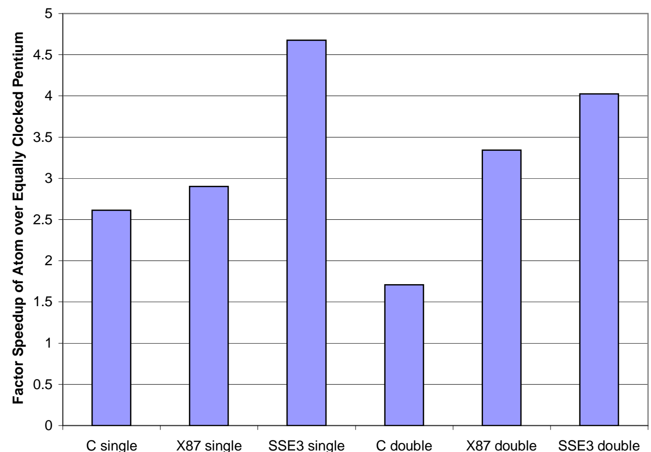

Figure 10: Atom vs. Pentium: Performance of Mandlebrot

and data signals, the values within the register files, including the non-architected registers, can be probed or modified.

- **Reading and Writing Memory**: When the core is paused, all memory requests attempted will be declined by the signal translator. This provides the window of opportunity to inject any memory operations across the memory bus.

Taken together, the above four mechanisms provide powerful and flexible debugging support to root cause a wide range of bugs without requiring re-synthesis. Essentially, these mechanisms greatly enhance both observability and controllability of the RTL under debugging. For example, the core can be dynamically paused when the retired instruction count reaches a particular value close to where a problem has occurred, providing the opportunity to inspect registers and memory, and to possibly make code or data modifications to aide in the debugging before resuming the original code execution. Though the Atom processor is an in-order processor, similar debugging techniques could be applied to an out-of-order design, which still retire instructions in-order so that precise exceptions are maintained.

6. ATOM CORE ON VIRTEX-5 FPGA

6.1 Functional Completeness

Our FPGA-synthesizable version of the Atom core correctly preserves all instruction set and microarchitectural features implemented in the original Atom core. These include the complete microcode ROM, full capacity L1 I/D-caches, SSE3, Intel64, Intel Virtualization Technology, and advanced power modes such as C6. We have modified the `CPUID` instruction to indicate the absence of the L2 cache in the modified uncore.

As expected, the synthesizable Atom core is capable of executing the rich variety of x86 software. In particular, we have successfully booted various off-the-shelf operating systems, including MS-DOS, Linux, and Microsoft Windows XP, and executed a number of applications on the emulation PC platform. Figure 8 shows a typical setup of the emulation PC platform where the FPGA emulator fits snugly in the CPU socket of an ASUS Socket-7 motherboard with 256 MB of memory. Figure 9 shows a screenshot of the CPU-Z application and the game DOOM running on the emulation platform after having booted Microsoft Windows XP. CPU-Z is a utility program that reports detailed information on the current platform, including processor capabilities and motherboard configurations.

6.2 Performance Evaluation

As an example to demonstrate the performance impact of ISA differences, we highlight two simple experiments to compare the FPGA-synthesizable Atom core which supports SSE3, and the Pentium core which does not. For this comparison, both Atom core and Pentium are synthesized to clock at the same frequency (50MHz) and the Pentium's L1 caches are temporarily increased to the same size as the Atom processor.

Figure 10 shows performance comparison for a compute-intensive Mandlebrot fractal computation. Six configurations are shown; the first three show single-precision floating-point performance for a C-based implementation, a

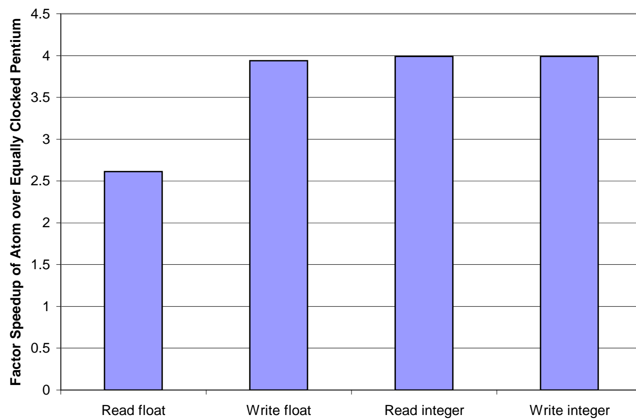

Figure 11: Atom vs. Pentium: Peak Data Access Bandwidth

hand-optimized x87 assembly implementation and a hand-optimized SSE3 assembly implementation. Also shown are these same configurations but for double-precision floating-point computations. Because the Pentium does not implement SSE, the hand-optimized x87 performance results are used. In all cases the Atom core demonstrates significant performance improvements over the Pentium, and the SSE-based single-precision and double-precision implementations achieve performance improvements ranging from 1.7x to 4.6x.

Figure 11 compares the peak data cache bandwidth achieved between the Pentium and the Atom. In all cases hand-optimized assembly implementations were used. The Atom core achieves up to 4x boost in peak bandwidth due to the use of SSE instructions. These results corroborate the significant performance advantages of new ISA features in the modern x86 ISA, even on a vintage PC platform that long precedes the advent of the Atom core.

7. CONCLUSIONS

In this paper, we detail our experiences using a modern FPGA to emulate a state-of-art x86 microprocessor. We share a methodology of taking the original RTL design and making it FPGA synthesizable through the standard EDA tool flow. In effect, our work provides an existence proof to demonstrate (1) the current FPGA technology can provide adequate capacity for a synthesizable design of a modern x86 processor core such as the Intel Atom, and (2) with in-depth knowledge of the processor architecture and through judicious FPGA resource mapping, the FPGA-optimized emulation can deliver several orders of magnitude higher emulation performance at orders of magnitude lower cost than the traditional emulation techniques. For future designs of x86 architectural extension and microarchitectural optimizations, our FPGA synthesizable Atom core offers a cost-effective approach to improve the efficiency and the productivity in both design exploration and validation.

8. ACKNOWLEDGEMENT

We would like to thank Stephen Robinson, James Allen, Jonathan Tyler, Michael Konow, Ketan Paranjape, Nick Samra, and Brian Klass for the productive collaboration throughout the project. We also appreciate the guidance and support from Elinora Yoeli, Joe Schutz, Shekhar Borkar, Justin Rattner, Abel Weinrib, Jim Held, Gadi Singer and Anand Chandrasekher. In addition, we thank the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper.

9. REFERENCES

[1] P. Alfke. Memories are Made of This. *Special Edition on Xilinx Virtex-5. Xcell Journal*, 2007.

[2] T. M. Burks, K. A. Sakallah, and T. N. Mudge. Identification of Critical Paths in Circuits with Level-Sensitive Latches. In *ACM/IEEE International Conference on Computer Aided Design*, November 1992.

[3] M. C. Chao, L. Wang, K. Cheng, S., and Kundu. Static Statistical Timing Analysis for Latch-based Pipeline Designs. In *Proceedings of the 2004 IEEE/ACM international Conference on Computer-Aided Design*, November 2004.

[4] 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.

[5] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, March 2000.

[6] 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.

[7] T. R. Halfhill. Intel's Tiny Atom: New Low-power Microarchitecture Rejuvenates the Embedded x86. *Microprocessor Report*, April 2008.

[8] *PowerPC Embedded Cores.* IBM Corp, Hopewell Junction, NY, 2000.

[9] *Pentium Processor Family Developer's Manual, Volume 1: Pentium Processors.* Intel Corp, 1995.

[10] *Intel64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation, November 2008.

[11] Intel Atom Processor. www.intel.com/technology/atom.

[12] Intel Demonstrates World's First Working Moorestown Platform. www.intel.com/pressroom/archive/releases/20081019comp.htm.

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

[14] J. Lee, D. T. Tang, and C. K. Wong. A Timing Analysis Algorithm for Circuits with Level-Sensitive Latches. In *ACM/IEEE International Conference on Computer Aided Design*, November 1994.

[15] 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.

[16] Mentor Precision RTL. www.mentor.com/products/fpga_pld/synthesis/precision_rtl.

[17] Mentor Veloce. www.mentor.com/products/fv/emulation/veloce/.

[18] OpenCores. OpenRISC 1000 Architecture Manual.

[19] S. Sutherland, S. Davidmann, and P. Flake. *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer, June 2003.

[20] Synopsys. DC-FPGA. www.synopsys.com/products/dcfpga.

[21] *Synplicity FPGA Synthesis Reference Manual.* Synplicity, December 2005.

[22] Xilinx. XST. www.xilinx.com/products/design_tools/logic_design/synthesis/xst.htm.

[23] *Microblaze Processor Reference Guide, v6.0.* Xilinx, June 2006.

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

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