

# Logic and Physical Synthesis Methodology for High Performance VLIW/SIMD DSP Core

Jagesh Sanghavi, Helene Deng, Tony Lu

Tensilica, Inc

sanghavi@tensilica.com

## ABSTRACT

We describe logic and physical synthesis methodology to achieve timing closure on a high-end VLIW/SIMD DSP processor core. The design comprises of approximately 200,000 placeable instances. The target frequency goal was to achieve 250 MHz in 130 nm technology. The VLIW/SIMD DSP is described using TIE (Tensilica Instruction Extension) language, which is a Verilog-like language for description of Instruction Set Architecture (ISA) extensions. The synthesizable Verilog (or VHDL) is generated using TIE Compiler. TIE Compiler automatically generates complex hardware structures such as multi-ported register files and pipeline management logic. This empowers a designer to easily add custom instructions to suit the target application. However, this also leads to physical design challenges. For example, the DSP core has 7-stage pipeline, 160-bit VLIW datapath comprising of ALU, MAC, and SHIFT units, a 160-bit wide multi-ported register file with 16 entries. This leads to complex bypassing logic, which results in a highly congested design. The traditional flow of using synthesis followed with placement and routing lead to a serious timing convergence issue. The use of Physical Compiler helped close the timing gap. In particular, the best results were found using timing driven congestion with high map effort compile for gates-to-placed-gates flow. One of the major challenge is the turnaround time which can be anywhere from few days to a week depending on the speed of the compute servers. In this paper, we describe the gates-to-placed-gates based Physical Compiler methodology that enabled our team to achieve the target frequency.

## Table of Contents

<b>1.0 Introduction .....</b>	<b>3</b>
<b>2.0 Design Overview .....</b>	<b>4</b>
<b>3.0 Design Methodology .....</b>	<b>6</b>
<b>4.0 Implementation Challenges and Solutions .....</b>	<b>9</b>
<b>5.0 Results, Recommendations, and Future Work .....</b>	<b>10</b>
<b>6.0 Acknowledgement .....</b>	<b>11</b>
<b>7.0 References.....</b>	<b>11</b>

## Table of Figures

<b>Figure 1 Datapath of DSP Core.....</b>	<b>5</b>
<b>Figure 2 Implementation Flow.....</b>	<b>7</b>

## 1.0 Introduction

The complexity of building a multi-million gates System-on-Chip (SoC) ASIC is a driver for semiconductor Intellectual Property (IP). A semiconductor IP is typically classified as hard or soft. However, any meaningful soft IP must have a predictable performance in a given target technology. Hence, a soft IP must have an implementation flow that "hardens" it for at least one ASIC library if not for one process technology. In general, a soft IP is expected to be portable across multiple process generations. Hence, implementation scripts for a soft IP are expected to easily support any arbitrary technology library that represents multiple ASIC library providers, multiple foundry vendors, and spans multiple technology generations. The IP implementation scripts provide a reference flow and in large number of cases, a subset of scripts—logic synthesis and physical synthesis—are used "as is" in the ASIC flow. Compared to logic and physical synthesis scripts for a specific ASIC design project, the scripts for implementing an IP are expected to be more robust and versatile.

An embedded processor IP core is at the heart of SoCs from applications as diverse as digital camera, networking equipment, digital satellite system. A processor core offers a programmable platform to design in presence of evolving standards and changing product specifications. In addition to the portability requirements, the implementation scripts must recognize and optimize processor-specific logic structures such as register file and decoder.

A processor core with fixed instruction set architecture is ill suited for different applications. A configurable processor IP provides a set of parameters that can customize the processor for a specific application. For example, the size of the register file and width of the processor interface can be chosen at the configuration time to suit an application. This imposes requirement on the implementation scripts that the performance of resulting hardware is predictable across configuration space defined by different parameters.

A configurable processor offers limited customization by choice of specific parameters. However, a substantial gain in application performance, i.e. number of cycles times clock period, is possible by adding new instructions to tailor-make the processor for a specific application[1]. The custom instructions are described using instruction set architecture (ISA) extension language. The configurability and extensibility together provide the end user to design their systems at a higher-level of abstraction using processor platform, which is emerging as a key system design building block. However, it is very important that the high-level description of custom instructions translates into timing and area efficient hardware. The major portion of onus of converting the generated RTL into efficient hardware relies on the implementation scripts, which are now expected to be scalable.

In this paper, we describe implementation of a high-end DSP processor core that is described using TIE (Tensilica Instruction Extension) language, which is a verilog-like ISA extension language. The DSP core was a brand new design with changes to TIE language and TIE compiler. Given the tight project schedule and complexity of the design, achieving the timing closure was a significant risk.

The paper describes the "war story" on development of implementation flow, application of the flow to achieve the timing goal, and productization of the flow for arbitrary processor extensions. Compared to logic synthesis and physical synthesis flows described in the literature [2,3,4], the implementation flow is different in the following aspects. The implementation flow is for an example of extensible soft IP. Hence it is portable, predictable, and scalable. Secondly, the implementation flow provides an option to perform behavioral retiming of the registers that is used by the example design to balance the pipeline stages and improve the area. Finally, the implementation flow is able to mitigate the high degree of congestion that is inherent in a deeply pipelined processor with wide datapath and complex forwarding logic.

The rest of the paper is described as follows. In Section 2, we provide an overview of the design. In Section 3, we present the implementation methodology. In Section 4, we describe the tactical challenges and solutions in meeting the timing goals. Finally, we present results and provide recommendations in Section 5.

## **2.0 Design Overview**

The high-end DSP described in this paper is a Very Long Instruction Word (VLIW) machine with 64-bit instructions. Each 64-bit instruction is organized as 3 VLIW slots. The ALU and MAC VLIW slots support Single Instruction Multiple Data (SIMD) operations. The DSP datapath is shown in the Figure 1. The ALU can support 8 x 20-bit or 4 x 40-bit arithmetic and logical operations. The MAC can support 4 16 x 16 multiplications with 40-bit accumulator. The processor has 128-bit interface with multiple load and store units. The processor has 16 entry, 160-bit wide vector register file. Each vector register file entry can be viewed as either 8 20-bit elements or 4 40-bit elements. The VLIW organization leads to the vector register file to have 6 read ports and 3 write ports. In addition to vector register file, the machine has 4 entry, 128-bit wide register file to support unaligned loads. Due to SIMD nature of the machine, it has 160-bit wide datapath. The ALU unit and MAC unit support the corresponding SIMD operations. The DSP has a SHIFT unit to support instructions that allow arithmetic right shift and logical left shift on 20 and 40-bit elements in form a SIMD operation. In addition, SELECT unit selects 8 20-bit output elements from 2 160-bit entries in the register file.

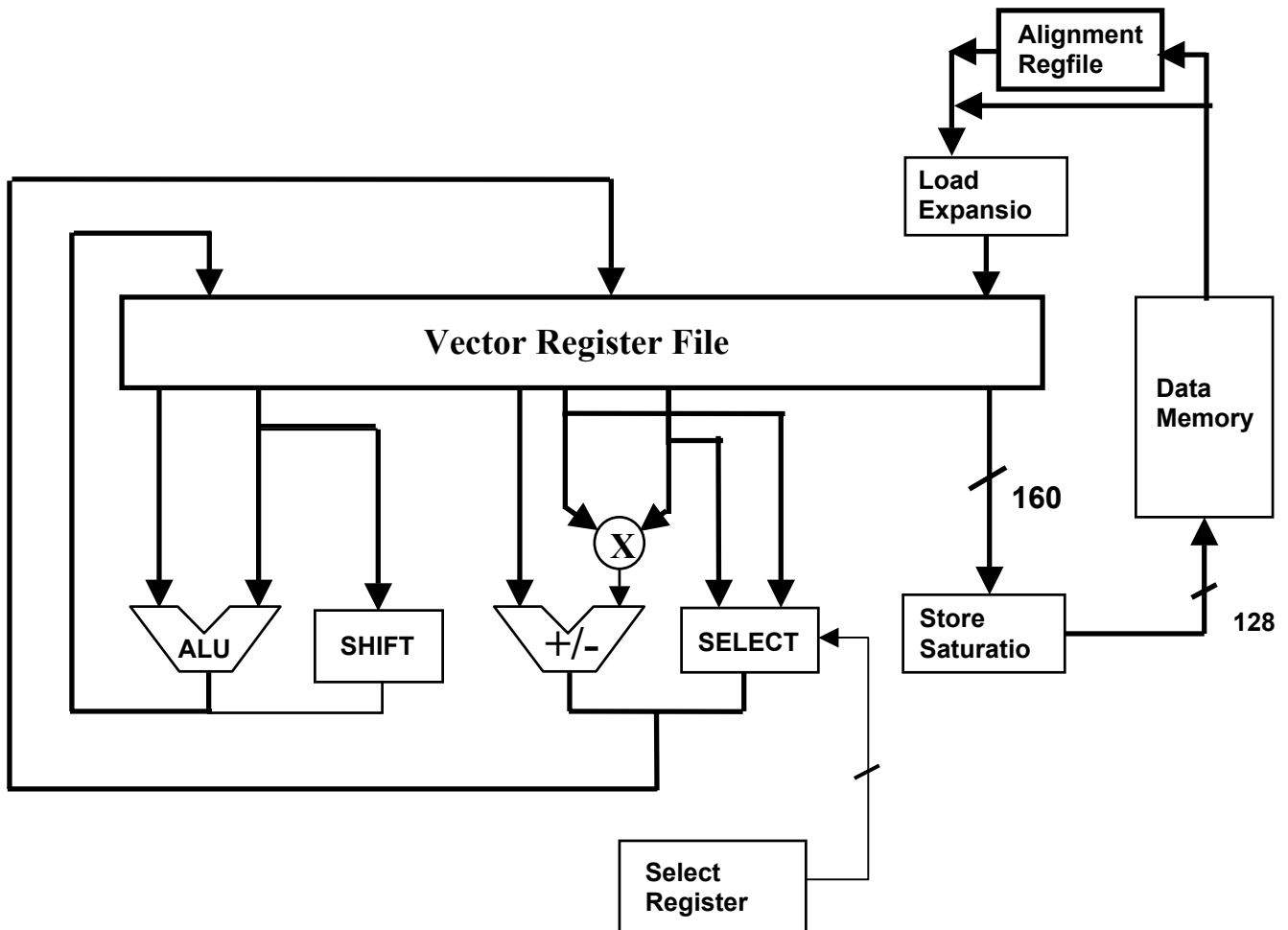


Figure 1: DSP Datapath

The DSP core is described in TIE. TIE Compiler generates a synthesizable hardware description. TIE Compiler automatically adds the logic to decode the instructions; manage the pipeline; multiplex results generated by various execution units; forward the results from one execution unit to another; stall the machine to avoid resource conflict or to wait for result to become available. Since the TIE Compiler automatically generates logic to manage the complex processor pipeline, it relieves the system designer to focus on accelerating their application by designing custom instructions, specifying register files to hold the operands, and execution unit to perform the instruction. However, a high-end DSP core designer using this approach has wide datapaths with complex bypassing logic that leads to congested design.

The target frequency goal for the DSP was 250 MHz at 0.9 V in 130 nm technology. The design has approximately 200,000 placeable instances. Due to 128-bit wide interfaces to multiple instruction RAMs, data RAMs, and data port has more than 1000 pins. Furthermore, TIE Compiler automatically inserts complex clock gating for power management. Hence, the design has more than 400 gated clock elements. In addition to power management clock gates, the design has certain clock gates, which have a timing requirement as they are used to stall the processor.

### **3.0 Design Methodology**

The implementation flow is shown in Figure 2. The DSP core is a portable IP. Hence, it is independent of the synthesis, placement, and routing tools. The implementation specification file captures information about the library, constraints, and optimization switches. The flow described here is based on Design Compiler for logic synthesis and Physical Compiler for placement and optimization. The rest of the flow comprises of third party tools to perform Clock Tree Synthesis (CTS), post-CTS placement optimization, and routing.

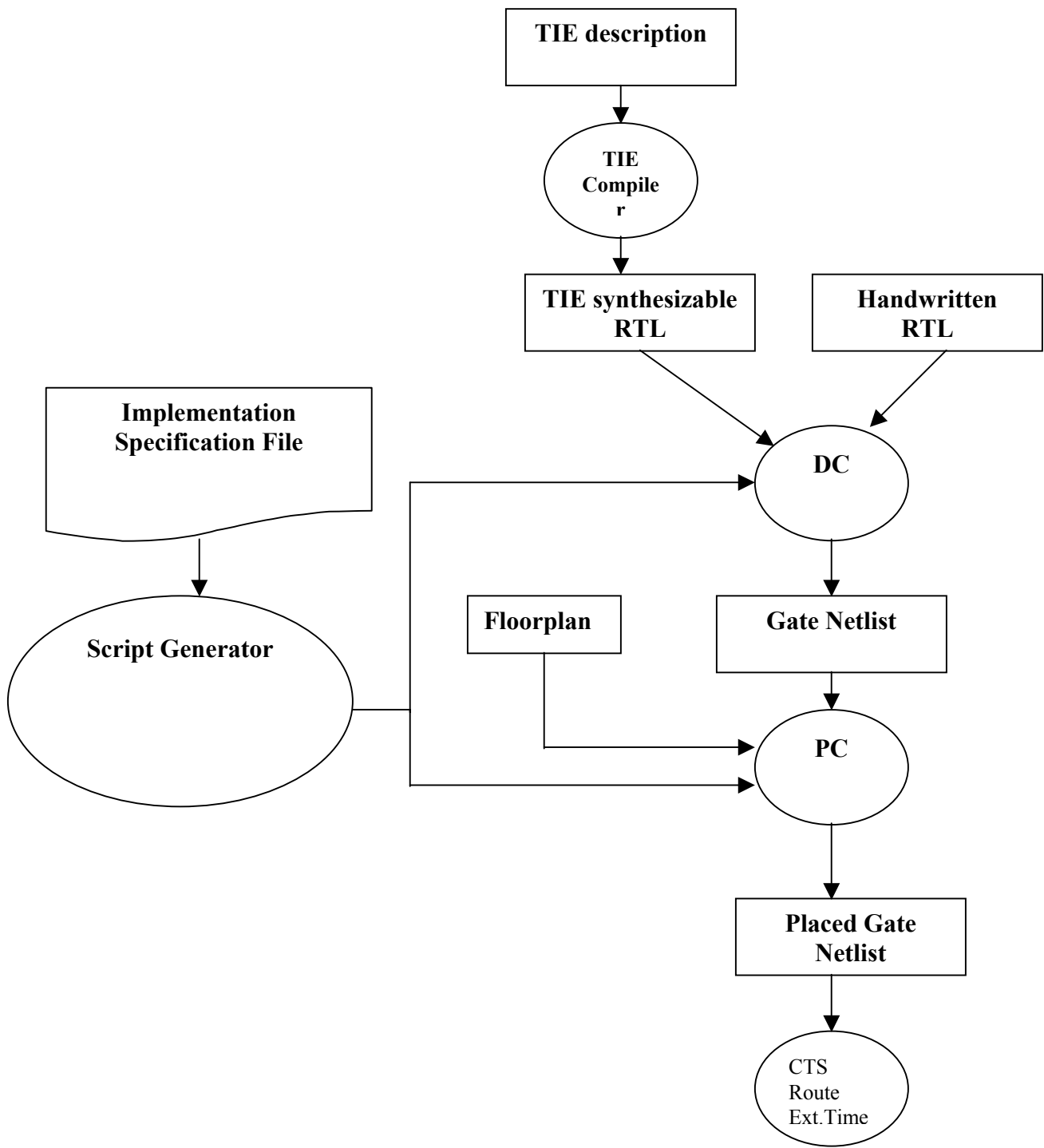


Figure 2: Implementation Flow

The flow starts with an implementation specification file that is organized as a set of variable, value pairs. As the name suggests, the implementation specification file allows the user to specify variables that affect the synthesis, physical synthesis, and layout. It provides a mechanism to centralize the implementation specific information that can be used to drive tools from different tool vendors. The implementation specification file allows user to specify following classes of information. First, it allows a user to specify different library views such as .lib, .pdb, .lef, .tlf, and .tf. This enables a user to easily experiment with libraries from different vendors for same or different process technology. Second, the user can specify design constraints and design environment. For example, the user can specify the target clock period, the driver cell, driven load, library cells to avoid, relative pin assignment, metal layers to use, utilization ratio, aspect ratio. Finally, it provides mechanism to specify optimization switches. For example, the user can specify high-level optimization choices, flat or hierarchical synthesis choice, and placement optimization mode.

The information specified in the implementation specification file is converted into tool specific scripts. For example, dsh or tcl script can be generated that capture various Design Compiler (DC) or Physical Compiler (PC) related variables. The DC scripts are expected to run on a design varying in size from 25,000 gates to 500,000 gates. The number of gates can be arbitrarily large in theory based on the hardware generated by the TIE Compiler. Hence, DC scripts are expected to be scalable. A simple top-down compile strategy might work for 25,000 gates design. However, in our experience, a bottom-up compile strategy with multiple top-level compile passes performs significantly better in practice.

Using bottom-up compile strategy, the DC script performs compilation based on the type of the design for the intermediate levels of hierarchy. For example, the DC compiles storage elements of the processor such as state and register file separately from the execution units such as ALU, MAC, and SHIFT units. This allows DC to divide-and-conquer by focusing on types of modules generated by TIE Compiler. Based on the knowledge about the flow of the data in the processor hardware generated by TIE Compiler, we are able to budget the time for different types of TIE modules. For example, a certain input signals come directly out of a flop, then an input delay of 5% to 10% of a clock cycle is adequate to model the clock-to-q. Conversely, if certain output signals are registered then an output delay of 5% to 10% of a clock cycle is adequate to model the setup time for the flop. For other signals, the time is budgeted manually as a certain percentage of the clock period. The constraints for a type of module are generated automatically from specified timing budgets.

The DC provides Behavioral Retiming (BRT) as an option to balance the pipeline stages and reduce the area by merging flops. The BRT causes issues with the formal verification flow, hence, it is provided as an option. The user can balance the pipeline by using specification in TIE description. Furthermore, complex clock gating that controls the gating for clock for each pipeline stage may preclude the retiming algorithm from moving the logic across flop boundary. Based on the description, the retiming algorithm believes that clock belong to separate domain, hence the logic can not be moved across flops, although it is possible to redistribute the logic to balance the pipeline. We ran into this situation while implementing a two cycles MAC operation. The solution was to break the MAC operation into the partial product computation from multiplication operation in one cycle, followed with 3 operand addition operation which is

implemented as CSA and a adder in the following cycle. The DC flow produces a flattened netlist that is used by the rest of the flow.

The Minimal Physical Constraint (MPC) floorplanning feature was unavailable in PC during the project. A third party floorplanning tool was in use. However, the third party floorplanning license was bundled with layout operations of placement and routing. Hence, although floorplanning step would require only few minutes, the flow had to pend on license availability while the layout tool performed time consuming steps of placement or routing. This was a serious under-utilization of PC license resource. Hence, we developed a Xtensa floorplan initialization tool internally. The Xtensa floorplan initializer is a simple floorplanning script that draws the standard cell matrix based on utilization ratio, allows specification of IO pins, puts down the tracks, and allows specification of power stripes / rings.

The PC flow was a gates-to-placed-gates (G2PG) flow. The various options such as `map_effort`, `congestion`, `timing_driven_congestion`, and `area_recovery` are exported to implementation specification file. We found timing driven congestion option to be most effective in managing the congestion of the design while meeting the timing requirements.

The clock tree synthesis, post-CTS placement optimization, routing, extraction, and post-layout timing verification were performed using third party tools. They are not described in this paper.

## 4.0 Implementation Challenges and Solutions

The project involved new development of TIE code to describe the DSP processor, changes to TIE Compiler to support Flexible Length Instruction Xtension (FLIX) architecture. The timing convergence was recognized as a project risk right from the beginning. Hence, RTL designers and Physical Design team worked together from early on in the design cycle.

The flow development for Design Compiler posed many challenges. The hardware description of TIE Compiler generated synthesizable DSP code had a small number of large designs and a very large number of small designs instantiated at the top-level. The small number of large designs represented register files, states, execution units, and decoder. The very large number of small designs represented flop banks and muxing logic that are described using primitives. A blind bottom-up compile runs out of steam due to extremely long run times. Some of the early runs were in a Taxicab mode where execution units were identified by specific name. The top-down compile ran out of memory. Once design hierarchy was clear, it was easy to construct a scalable script that relied on grouping. The grouping was also used to optimize Stall signal that can be generated from various modules such as register file or state to wait for data to become available or from execution unit to avoid resource conflict. The Stall signal is timing critical and by its nature spread across different modules in the logical hierarchy. The scripts used over constrained clock to make DC work harder. The objective is to compensate for the inaccuracy in the wire load model.

Early experiments with standard synthesis, placement, and routing using standard flow gave a strong indication that it would not be viable to achieve the timing goal. Bottom-up compile strategy to support scalability and presence of BRT option meant that the rtl-to-placed-gates

(RTL2PG) flow was not a realistic choice for Physical Compiler. The gates-to-placed-gates (G2PG) flow was developed. The development of gates-to-placed-gates flow required additional scripting because the placed format produced by PC was not acceptable to layout tools. The G2PG flow required approximately 5 to 7 days on old Solaris boxes. The Linux support was a must. The early Linux port was available and had issues with grid precision. This caused the routing to fail. The issue was later fixed. The Linux port helped reduce the turnaround time to about 2 days.

Given the long turnaround time for PC, the DC was used to measure the timing progress on RTL. It is clear that PC is required to model the interconnect delay accurately. However, DC run could be completed in about 12 to 16 hours. The DC regressions were every night. This allowed us to closely monitor the impact of RTL changes on timing. When the RTL design is in a flux and especially when a large amount of RTL is automatically generated, the nightly regression helps keep a very good lock on the timing.

The design had a stall signal to shut down the machine by disabling the clock. The synthesis and physical synthesis assumes ideal clock, however, the time equal to insertion delay is required to propagate the signal in the routed netlist. This was modeled as a constraint on the internal pin.

The other issue we faced is that the `set_critical_range` command is unable to provide sufficient resolution between the most critical and near critical paths. Any of number of near critical paths could show up as the most critical. This causes some amount of consternation from RTL designers trying to fix the critical path. The solution was to focus set of critical paths instead of simply top one or two. We developed a simple script to collapse the critical paths that belonged to the start point and end point of a bus. Hence, a large number of top paths can be dumped out and the number can be compressed down to focus on unique paths.

## **5.0 Results, Recommendations, and Future Work**

We observed that timing driven congestion mode was effective in managing the congestion of the design in most cases. We observe a good correlation between PC and layout timing when congestion is under control. The team was able to achieve post layout target of 250 MHz. In general, we see that Physical Compiler timing is approximately 15% to 20% better compared to standard flow even for other designs that have low congestion.

To manage the timing goal, it is important to have reasonable timing budget negotiated among different blocks. This helps to identify who needs to look at their RTL—hand written or generated--in order to fix the issue. Furthermore, an overnight regression is a must. When the design is in a flux, a seeming innocuous change to fix functional issue can introduce a timing bug that may not be identified till very late. Having a nightly run provides a tight handle on managing the impact of changes on the timing. Hence, although an inaccurate measure in the present 130 nm or emerging 90 nm technology nodes, the synthesis timing provides a good mechanism to track the timing progress.

The turnaround time of Physical Compiler is an issue. The use of fast linux boxes helped mitigate the lengthy runs. We did not experiment with `-quick` option, which might be useful in

the early phases of the design if the accuracy of the produced results is reasonable or if the errors are consistently in one direction.

A smoother integration with downstream CAD tools is a good thing. Usually, format translators are not the main IP for a CAD vendor. Furthermore, a CAD vendor does not have access to tools from the other CAD vendors. This causes significant amount time to be wasted in identifying the issue, finding a fix, automating the fix with a glue script.

We had a large number of gated clocks for power management in the generated RTL. We played with clock gating area multiplier switch in Physical Compiler. However, the results were inconclusive.

## **6.0 Acknowledgements**

We would like to thank Eliot Gerstner for developing the Xtensa floorplan initialization utility. Thanks also to Xtensa RTL, DSP TIE code development, and TIE Compiler teams.

## **7.0 References**

- [1] R. Gonzales, "A Configurable and Extensible Processor", IEEE Micro, March 2000.
- [2] A. Gupte, "Physical Synthesis Methodology for High Performance Embedded DSP Core", SNUG India 2001.
- [3] C. Michon, "Timing Closure using Physical Compiler", SNUG Singapore 2001.
- [4] S. Jones and Michael Robinson, "Improving Timing Predictability for Soft Cores", SNUG San Jose 2002.