COMPARING RTL AND HIGH-LEVEL SYNTHESIS METHODOLOGIES IN THE DESIGN OF A THEORA VIDEO DECODER IP CORE

Leonardo Piga[*], *Sandro Rigo*

Institute of Computing, University of Campinas
Av. Albert Einstein, 1251, 13083-970, Campinas, SP, Brazil
email: leonardo.piga@students.ic.unicamp.br, sandro@ic.unicamp.br

ABSTRACT

An important share of the consumer electronics market is focused on devices capable of running multimedia applications, like audio and video decoders. In order to achieve the performance level demanded by these applications, it is important to develop specialized hardware IPs in order to cope with the most computational intensive parts. Nowadays, designers are facing the challenge of integrating several components, including processor, memory, and specialized IP cores, into a single chip, giving raise to the so called Systems-on-chip (SoC). The high complexity of such systems and the strict time-to-market in the electronics industry motivated the introduction of new design methodologies during the last years. This work presents a comparison between two hardware development methodologies in order to design a Theora video decoder IP core from algorithm down to FPGA. We first implemented it in hand-written RTL code using VHDL, resulting in a 56% time reduction in the decoding process when compared to a software library. The second methodology implements the same hardware using SystemC and behavioral synthesis. The second IP core was developed in 70% less time with satisfactory results. We compare the two approaches in terms of area and latency.

1. INTRODUCTION

Since the last decade, the electronics industry has been challenged by the complexity growth of their systems and by a tight time-to-market. Consequently, new methods and techniques for hardware implementation have been developed. In this paper, a comparative study between two hardware design methodologies for a Theora [1] video decoding IP is shown: the first one is a hand-written implementation completely programmed in VHDL [2]; the second one is based on behavioral synthesis of a SystemC [3, 4] model directly to Verilog RTL, VHDL RTL or gate-level using a tool called Cynthesizer, by Forte Design Systems[5].

[*]This project was supported by FAPESP (06/61348-0)

Decoding a Theora video is a nontrivial task and it requires significant computational processing, making a software based decoding process prohibitively slow for real-time video applications on embedded systems. In this way, decoding video with hardware assistance is particularly interesting and it provides a way to develop embedded systems that are able to decode videos in real-time using reasonable computational resources.

This paper is organized as follows: section 2 shows related works; section 3 explains briefly the encoding and decoding of a Theora video and describes the implemented modules; section 4 presents the design methodology; sections 5 and 6 report the VHDL and the SystemC implementation, respectively; and section 7 discuss the results.

2. RELATED WORKS

Hardware and software co-implementation has been used to speed up MPEG-4 [6] videos and to improve computation performance on computer graphics algorithms [7]. So, Diniz, and Hall [8] have developed a compilation system that translates from high level algorithms programmed using SystemC to specific FPGA systems. However, their benchmark was made using small applications as, for example, matrix multiplication. Chtourou and Hammami [9] have developed a methodology for behavioral synthesis of SystemC code. They evaluated small applications, such as, Fast Fourier Transform.

This work differs from the above mentioned research due to its focus on analyzing the performance of a complex IP core designed using both the classical approach developed with RTL VHDL, and a more recent high-level methodology based on behavioral synthesis using SystemC. Not only do we show that a hardware/software co-implementation of Theora decoder brings a large speed up on design time, but also, we present the flexibility and optimization possibilities brought by the higher level of abstraction.

3. THEORA ARCHITECTURE

Theora video compression is done following steps that remove redundancy in the spatial, temporal, and frequency domains[10].

In order to remove spatial and frequency redundancy, the first step consists on dividing the color planes of a frame in matrices of 8x8 elements. On the second stage, the type II discrete cosine transform (DCT [11]) is applied on each block, separately. The third step, which is called quantization [12], removes the redundancy on the frequency domain by cutting down high frequency components from the frame. The fourth step linearizes the sixty four elements of the matrix in a zig-zag way. The fifth stage applies the Huffman [13] algorithm. These five steps remove redundancy of spatial and temporal domains. When a frame is encoded as described before, it is called as a golden frame or an intra frame, depending on how an inter frame refers to it. Theora makes predictions just based on former frames, to encode an inter frame. This technique consists on subtracting the current frame from the frame used as predictor, encoding just the differences or minimizing the differences between current frame regions and reference frame regions. Thus, the difference is encoded in the same way of an intra frame and for the latter case, a shifting vector that describes how the difference must be shifted is kept, too. The reverse process is called decoding. When it finishes, the frame is completely decoded and ready to be displayed.

We used a profiling tool called *gprof* [14] to analyze the Theora library version alpha 6, running on a x86 platform. This analysis showed us which functions should be implemented in hardware, due to the effort spent by the software on them. We chose Reconstruct, LoopFilter, and their related functions because they accounted for up to 70% of all decoding time. Most of them had their behavior completely implemented in hardware. The exception is ReconRefFrames, which is divided in two parts. We chose to keep one part in software because of the lack of data processing. This part of ReconRefFrames just makes some decisions on which parameters must be forwarded to hardware. Figure 1 shows the architecture of these modules.

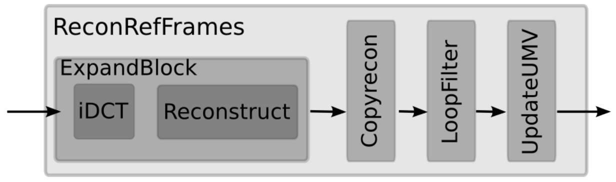

Fig. 1. Theora Modules

A module called *iDCT* receives blocks of 8x8 pixels coded as a quantization matrix. The blocks represent the coefficient resulting from DCT evaluation on a block of pixels that has not been decoded yet. At first, iDCT module must dequantize the coefficients that were received. Next, it does the iDCT based on these coefficients. The result is a block of pixels similar to the original that is sent to the next module in the chain, ExpandBlock.

ExpandBlock is responsible for deciding which procedures must be applied depending on the frame type. It reconstructs and decodes a block of pixels.

LoopFilter is a module that soften defects. It applies to the reconstructed frame a non-linear function that blurs the edges of the blocks giving a better appearance to the video. Theora video frames have a border that is not displayed. It exists to protect LoopFilter from accessing an invalid buffer region. After the application of LoopFilter, invalid values are left in this border. The *UpdateUMV* module restores this area.

The *ReconFrames* module is responsible for applying the ExpandBlock function on the frame until all necessary blocks have been reconstructed. *CopyRecon* is a module that copies a reconstructed frame to another buffer region depending on an offset parameter.

ReconRefFrames receives parameters from the software and forwards them to the IP modules. It is responsible for managing all other modules in the Theora IP core.

Some software optimizations that are used in the Theora software implementation can be replaced with generic implementations since, in hardware, these functions would not increase the performance. Besides, in order to save memory, we replaced matrices that kept pre-processed values by modules that process them on-the-fly.

4. METHODOLOGY

We have independently developed, tested, and validated each module of the theora video decoder. We used the *libtheora* software library as a reference model. For verification purposes, each function that was ported to hardware had its input parameters generated by its software implementation and written on an input file. A wrapper module was used to sent these inputs to the modules. In addition, data modified by the modules were kept on files and used to compare outputs against the reference model.

Our goal was to compare the effort and performance of a Theora IP core designed following two different methodologies. We first adopted the hand-written RTL approach, using the VHDL language to design the hardware modules. In contrast, we used the SystemC [3, 4] language along with Cynthesizer behavioral synthesis tool by Forte Design Systems [5] for the second approach.

For practical purposes, in the behavioral methodology we coded all Theora functions into a single module which is fed to Cynthesizer for RTL synthesis. We have gradually designed this single module, starting by iDCT and going towards the end of the decoding process, following the same architecture adopted by the RTL.

A testbench [15] is responsible for verification and validation. The Theora software library compiled for x86 is our reference model. The same testbench structure was applied to VHDL, SystemC, and FPGA tests. The reference model generates inputs for a component and saves them on a file called IN.TB. It also keeps the outputs on another file called OUT.EXPECTED.

Simulation requires the development of some modules that are not synthesized. These components read the input files and forward the data to the components under test, which we call design under verification (DUV), using 32 bit packages. When a DUV component is done with its work, it sends the output data to the testing module which writes them on a file called OUT.DUV. These files are compared with the reference (OUT.EXPECTED) ones. The verification process is successfully finished if they are equal, which means that the hardware coded with VHDL or SystemC has the same behavior as the software reference model. Figure 2 illustrates how this process works.

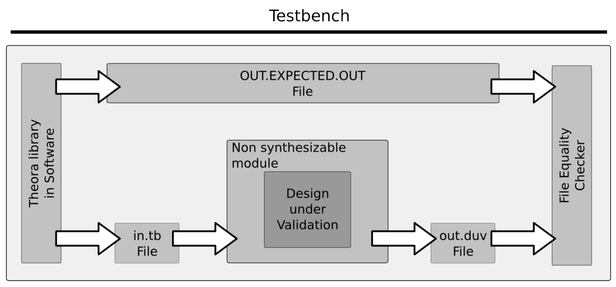

Fig. 2. Verification Methodology: Simulation and FPGA

This methodology is also used for validation of the RTL design on FPGA. In this design stage, the Altera NIOS II processor receives output data from the top-level module and it forwards them to a JTAG interface connected to a computer that records them on a file called OUT.DUV.

5. THE RTL DESIGN

A bus connected to the top-level module does the communication between the hardware and the software. A YCbCr to RGB converter implemented in VHDL receives the output from the top-level module. Hence, it is possible to display an image on a VGA monitor. Figure 3 shows the architecture.

Beginning with ReconRefFrames, all modules were synthesized and completely tested on FPGA. Altera Stratix II EP2S60F672C5ES was used. The hardware was synthesized and connected to a NIOS II processor using an Avalon Bus. The Theora IP acted as a NIOS II peripheric and could be accessed by a software running on this processor. The operation frequency was 50MHz and the software was a customized Theora library in which the functions converted to hardware were replaced by calls to the IP core.

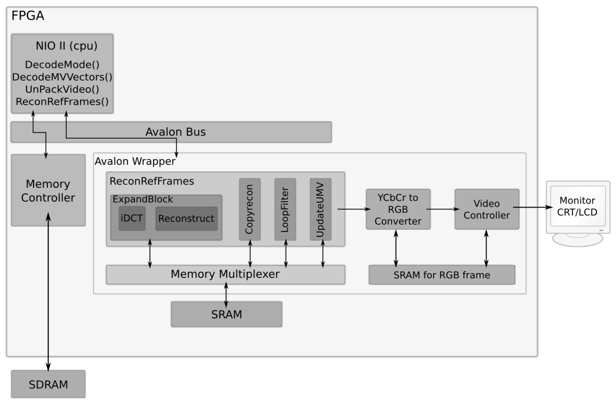

Fig. 3. Complete Hardware Architecture

6. THE BEHAVIORAL DESIGN

Behavioral SystemC code is close to the algorithm form of the reference model that was implemented using a high level language. As higher is the abstraction level, as easier is to explore different design possibilities. Hence, more hardware varieties can be tried without being concerned about the hardware description. Design exploration is much faster. Changes are done in the algorithm, abstracting the low level of a hardware description. It is possible to completely change the algorithm without spending a long time repairing the RTL. Thus, there is more freedom for development.

The RTL code is difficult to maintain and reuse because it describes a module in an very low abstraction level, with too many details and a structure very close to the generated hardware. The code tends to become huge and complex. Also, implementation decisions might turn the code dependent on factors like FPGA technology (or ASIC) used for synthesis of the modules.

In the behavioral synthesis approach, the development and the synthesis process become more dynamic, allowing more design variations with a variety of structural decisions without increasing development time and cost. For our Theora IP, these different implementations were tested and compared to the RTL VHDL implementation shown on section 5.

We use a tool called Cynthesizer, developed by Forte Design Systems [5]. It is able to convert a behavioral SystemC code into SystemC RTL, Verilog, or VHDL. This tool requires some constraints in order to synthesize the behavioral SystemC code. Although these rules create some limitations, there is more freedom than coding in RTL. We show some of the rules that should be followed below. Readers that are not familiar with SystemC should refer to [16, 17] in order to be able to fully understand the meaning of these constraints.

- The design must be represented as a SC_MODULE with only one SC_CTHREAD and the algorithm functionality must be implemented as an infinite loop nested on SC_CTHREAD.
- One cycle initialization data must be implemented on Reset phase. Multi-cycle initialization must be implemented inside the SC_CTHREAD.
- The input/output handshake must be implemented nested to a SC_CTHREAD in a cycle-accurate manner. Its behavior is defined using the SystemC `wait()` command.
- Dynamic memory allocation and complex function calls, as for example `sqrt()`, are forbidden. Also, OO programming features available on C++ are not permitted.

In addition, Cynthesizer defines a code style that should be followed on the SystemC code. Nevertheless, programming the algorithm that will be inside the SC_CTHREAD is not as complicated as programming the RTL. `While` and `for` loops are permitted. Vectors are automatically converted into memory. The program can access the memory as if it were a C program. The programmer does not need to worry about clock cycles and read or write states. Figure 4 shows a sample code.

```
void my_modulo::thread0() {
  int i;
  // initialization logic
  { CYN_PROTOCOL( "Reset" );
    // one cycle initialization signals
    wait();
  }
  // Multi-cycle initialization

  // main loop
  while(1) {
    { CYN_PROTOCOL("Input");
      // Input managed by the clock
    }
    // Evaluations
    for (i=0; i < 4; i++) {
      CYN_UNROLL (COMPLETE, 4, "mem_ini");
      r[i] = i;
    }  ...
}
```

Fig. 4. Code Example

The synthesis process using Cynthesizer is different from the regular approach that has been adopted up to now. At first, the tool converts the behavioral SystemC code to a SystemC RTL code. Then it translates the latter into a Verilog RTL code, respecting the operation schedule in each clock cycle. In this process, the designer can try many possibilities for the state machine elaboration by setting optimization parameters, like enabling loop unrolling or requesting the tool to allocate the resources in a pipelined way. Also, there are other restrictions like maximum latency and minimum output flow that may be configured. When these parameters are set, the tool tries to satisfy all the restrictions. However, the clock cycles used during the handshake are constant and they should be defined by the designer because the tool does not change them. In addition, there are other options such as inference of synchronous memory or register banks from vectors.

Varying these parameters, the designer can try different structure combinations. By analyzing reports created by the tool, he/she can choose which one is the best for the design. For example, if a module needs to be fast, the combination with the highest output flow speed is chosen. It might be a pipelined implementation. On the other hand, if area is a key concern, the fastest option will probably be prohibitive large in area, but one with a good trade off between area and performance may be obtainded by fine tuning the synthesis paranmenters.

Tool constraints which restrict the usage of dynamic memory, pointers, and so on has caused changes on the library that we used. We have replaced all the pointer access with static memory use. Because we have substituted vectors that keep pre-processed data by functions that calculate their result on-the-fly on the RTL code, we have adopted the same strategy here to ensure a fair comparison. Therefore, the behavioral SystemC code is not the same as the Theora software library, but it is very similar.

We have created a testbench in the same way we have done for the RTL. It reads an input file and generate an output file. The process of verification follows the same rules described on section 4.

7. EXPERIMENTAL RESULTS

In this section we present a comparison between the co-implementation developed in RTL VHDL and the software library running on FPGA with the Nios II processor to demonstrate the results on performance gain for this approach. Also, we show a comparison between the hardware developed in RTL VHDL and Behavioral SystemC using simulation.

In order to compare performance between the decoding process done totally by software and the decoding process done by the HW/SW co-implementation, we have developed a system that has a Nios II processor and YCbCr to RGB converter module that is connected with an Avalon bus. The data flow from the processor to the converter without using Theora hardware decoder. We have compiled Theora library to the Nios II processor. All decoding process is done by software. Figure 5 shows the architecture.

We used 50 MHz as the maximum frequency operation. As a result, both architectures must work on this frequency. We have used a video of 35 seconds and 96x80 pix-

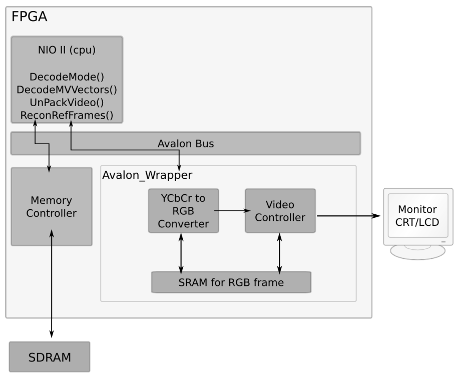

Fig. 5. Reference Architecture

els. The video size is small due to FPGA internal memory constraints. The video was encoded in a range of bit-rates from 56 kbps to 686 kbps. Thus, we can evaluate the results for videos of different qualities. Table 1 shows the decode time for a video decoded by software and by our HW/SW co-implementation decoder.

Bitrate (kbps)	**Time (s)**			**%**
	Real	**Without Hardware**	**With Hardware**	
56	35	22	15	48
91	35	26	17	51
182	35	33	23	46
364	35	44	33	36
636	35	61	48	27

Table 1. Decoding Comparison

Variation	**Area** (mm^2)	**Latency** (cycles/block)
RTL	11.40	1021
C_BASIC	7.35	1934
C_UNROLL_IDCT	8.35	1757
C_UNROLL_LF	8.07	1933
C_UNROLL_RF	7.45	1787
C_UNROLL_CR1	7.36	1796
C_UNROLL_CR2	7.29	1766
C_UNROLL_UMVB	7.84	1843
C_UNROLL_IDCT_RF	8.55	1610
C_UNROLL_IDCT_RF_CLR	8.64	1603
C_UNROLL_IDCT_RF_CR2	8.73	1413

Table 2. Synthesis results

Table 2 shows that not only is the hardware able to hasten the process in approximately 40%, but also it can decode a 364 kbps video in real time. Only Nios II running at 50 MHz and without any hardware assistance cannot decode videos with a bit-rate grater than 182 kbps. Hence, if high bit-rate videos must be decoded on an embedded system, a hardware assistance is necessary.

We did several synthesis alternatives in order to explore the range of possibilities available in Cynthesizer. We tested them in the most critical regions of the decoder where less latency is more likely to be affected. The most common parameter that we used was CYN_UNROLL. This option does complete or partially loop unrolling, therefore, reducing latency because of an increase in parallelism. Below, a brief description of each synthesis variation.

- C_BASIC: Standard synthesis configuration.
- C_UNROLL_IDCT: iDCT loops are unrolled
- C_UNROLL_LF: LoopFilter loops are unrolled
- C_UNROLL_RF: ReconFrames loops are unrolled
- C_UNROLL_CR1: Internal CopyRecon loops are unrolled
- C_UNROLL_CR2: External CopyRecon loops are unrolled
- C_UNROLL_RF: UpdateUMV loops are unrolled
- C_UNROLL_IDCT_RF: ReconFrames and iDCT loops are unrolled
- C_UNROLL_IDCT_RF_CLR: ReconFrames, iDCT, and memory initialization loops are unrolled
- C_UNROLL_IDCT_RF_CR2: ReconFrames, iDCT, and external CopyRecon loops are unrolled

Table 1 shows the area and the latency of the RTL and of each behavioral synthesis variation. In order to obtain the area of the RTL design, we used Cadence RTL Compiler [18] and Cynthesizer to acquire the areas of the behavioral synthesis. We measured the latencies by counting the number of clock cycles spent to decode a frame and dividing it by the number of blocks on a frame.

The synthesis have different areas and latencies. To depict the range of possibilities, we plotted a graph shown in Figure 6. Each point represents one synthesis. As close to the origin as better is the design because we want to minimize the latency and the area. The most striking point is C_UNROLL_CR2, which has the lowest latency and area.

The tool did not allow us to try more variations because they became too complex, exhausting the computer memory during the synthesis process. As a consequence, we were unable to synthesize a design with the same or better latency as the hand-written RTL. If it were possible, probably the synthesized circuit would consume less area than the RTL. This conclusion is reinforced if we do an extrapolation on the graph.

In addition, the time that we spent developing the behavioral model was 3 times faster, and obtaining several RTL

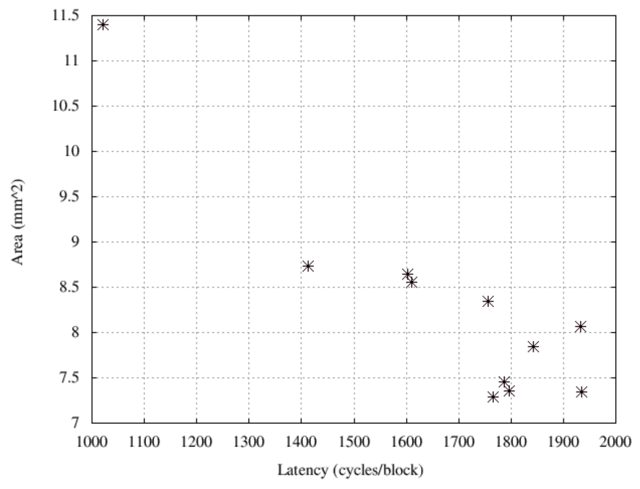

Fig. 6. Graph for synthesis variation: Area x Latency

design alternatives, instead of just one. This fact shows that the behavioral synthesis is a flexible and efficient way for IP core design.

8. CONCLUSION

This paper shows two hardware implementations for Theora video decoding: one using hand-written RTL VHDL and another one using behavioral SystemC synthesized with Forte Cynthesizer tool. Both of them are based on HW/SW co-implementation concept. We synthesized the former on FPGA and compared it with a pure software decoding library.

By comparing both approaches, we have shown that hardware IP design using behavioral synthesis can enable design space exploration in a way that is impossible to achieve using hand-written RTL code. By adjusting the optimization parameters on the synthesis tool, we could develop a number of design variations in one third of the time spent to write a RTL VHDL model for the same IP core. However, the RTL VHDL hardware is 38% faster than the quickest IP synthesized from the behavioral SystemC. The RTL was coded to ensure a low latency. On the other hand, minimizing the area is the default goal of the behavioral synthesis tool. Despite of its great performance, the amount of area occupied by the hand-written RTL (30% greater than the largest behavioral SystemC variation) would probably be unrealistic for a real-world embedded system design. By applying the optimization techniques in the behavioral synthesis tool, the designer is capable of seeking a good compromise between area and latency in a reasonable amount of time.

9. REFERENCES

[1] X. Foundation, "Theora I specification," May 2006.

[2] V. S. Committee, "IEEE standard VHDL language reference manual." *ANSI/IEEE Std 1076-1993*, pp. i+, 1994. [Online]. Available: \url{http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=392561}

[3] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Desing with SystemC*. Kluwer Academic Publishers, 2002.

[4] OSCI, *SystemC Version 2.1 User's Guide*, 2005, http://www.systemc.org.

[5] F. D. System, "Cynthesizer and high-level design," 2006, http://www.forteds.com/.

[6] S.-H. Wang, W.-H. Peng, Y. He, G.-Y. Lin, C.-Y. Lin, S.-C. Chang, C.-N. Wang, and T. Chiang, "A software-hardware co-implementation of mpeg-4 advanced video coding (avc) decoder with block level pipelining," *J. VLSI Signal Process. Syst.*, vol. 41, no. 1, pp. 93–110, 2005.

[7] P. Zemcik, "Hardware acceleration of graphics and imaging algorithms using fpgas," in *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*. New York, NY, USA: ACM, 2002, pp. 25–32.

[8] B. So, P. C. Diniz, and M. W. Hall, "Using estimates from behavioral synthesis tools in compiler-directed design space exploration," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM, 2003, pp. 514–519.

[9] S. Chtourou and O. Hammami, "SystemC space exploration of behavioral synthesis options on area, performance and power consumption," *Microelectronics, 2005. ICM 2005. The 17th International Conference on*, pp. 5 pp.–, December 2005.

[10] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, 2003.

[11] K. R. R. e P. Yip, *Discrete Cosine Transform*. Academic Press, 1990.

[12] M. K. e. H. M. A. Hallapuro, "Low complexity transform and quantization – part I: Basic implementation," JVT document JVT-B038, Geneva, February 2002.

[13] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[14] J. Osier, *GNU profiler, gprof manual*, 1993.

[15] J. B. Q. D. Corporation, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publisher, 2000.

[16] F. D. Systems, *Cynthesizer User Guide*, September 2005.

[17] ——, *Cynthesizer Style Guide*, September 2005.

[18] Cadence Design System, Inc., "Encounter RTL compiler datasheet," 2007.