Chapter 7
“All-in-C” Behavioral Synthesis and Verification with CyberWorkBench
From C to Tape-Out with No Pain and A Lot of Gain

Kazutoshi Wakabayashi and Benjamin Carrion Schafer

Abstract This chapter introduces the benefits of C language-based behavioral synthesis design methodology over traditional RTL-based methods for System LSI, or SoC designs. A comprehensive C-based tool flow, based on CyberWorkBench™ (CWB), developed during the last 20 years at NEC’s R&D laboratories is introduced. This includes behavioral synthesis and formal verification and hardware–software co-simulation of entire complex SoC. First we introduce the “all-in-C” concept based on CWB.

Then we discuss the behavioral synthesis for various types of circuits and examine the advantages of behavioral synthesis on the hand of commercial ICs. We show that currently entire SoCs are created using this flow in a fraction of the time taken by traditional approaches.

Behavioral IP and C-based configurable processor synthesis and automatic architecture exploration is explained next. At the end we demonstrate a real world example of a mobile phone SoC where most of the modules are synthesized from C descriptions using CWB.

Keywords: Behavioral synthesis, Control and data intensive flows, All-in-C, Behavioral C level formal verification, Hardware-software co-simulation, Automatic system exploration, Behavioral IP, Configurable processor

7.1 Introduction

The design productivity gap problem is becoming more and more serious as VLSI systems become larger. In the mid-1980s, gate-level design shifted to register transfer level (RTL) design for designs that typically exceeded 100K gates (we assume a hundred thousand gates is the upper limit for hand coded modules to be designed in several months).

Currently, several million gates circuits are commonly used just for random logic parts of a design, which equate to more than several hundreds thousand lines of RTL
code. It is therefore needed to move the design abstraction one more level in order to cope with this increasing complexity. Behavioral synthesis is a logic way to go as it allows “less detailed design description” and “higher reusability”.

A higher level of abstraction description requires smaller code and provides faster simulation times. For example a one million gates circuit requires about 300K lines of RTL (Verilog or VHDL) code, but only around 40K lines of C code. The RTL simulation of 300K lines, we observed in [1], is on average 10–100 times slower than the 40K lines of equivalent behavioral code (it is important to note that in order to benefit from higher level of abstraction the entire design needs to be modeled at the behavioral level).

It is sometimes claimed that behavioral synthesis is only useful for dataflow intensive circuits, but not for control dominated circuits. We believe that behavioral synthesis can and should be used for all hardware modules in order to truly benefit from it. We will demonstrate this by an example of a real complex SoC design where all custom design modules, except the analog ones, have been designed using behavioral synthesis. NEC Electronics adopted behavioral synthesis as standard design methodology since 2003 and taped out since then several hundreds million Dollars worth of “C-based” chips every year.

Since the benefits of behavioral synthesis are palpable through multiple commercial chip successes, Behavior Synthesis, or High Level Synthesis, is gaining acceptance within the design community, especially in Japanese industries. Various commercial chips for printers, mobile phones, set-top-boxes and digital cameras are designed using behavioral synthesis these days. ANSI-C is the preferred programming language for behavioral synthesis because embedded software is often described in C and design tools like compilers, debuggers, libraries and editors are easily available and there is a big amount of legacy code.

In this paper, we first provide an overview of our C-based design flow where we compare the efficiency and simulation performance against pure RTL as well as co-simulating it with embedded software. We show the advantages of C-based behavioral IPs over RTL IPs and how application specific processors can benefit from it. We present a hardware architecture explorer at the behavioral level allowing a fast and easy way to study the area, performance and power trade-offs of different designs automatically. Finally we demonstrate on a real complex design, how behavioral synthesis can be used for any hardware module (data and control intensive).

### 7.2 C-Based Design Flow

We have been developing C-based behavioral synthesis called “Cyber” since the late 1980s [2] and developing C-based verification tools such as formal verification and simulation around Cyber during the last 10 years [3]. All these tools are integrated into an IDE, where designers execute these tools upon the C-source code. We named this IDE tool suite “CyberWorkBench™”.
7.2.1 Basic Concept of CyberWorkBench

The main idea behind CyberWorkBench is an “all-in-C” approach. This is built around two principal ideas (1) “all-modules-in-C” and (2) “all-processes-on-C”.

(1) All-modules-in-C: means that all modules in a VLSI design, including control intensive circuits and data dominant circuits, should be described in behavioral C language. Our system supports legacy RTL or gatenetlist blocks as black boxes, which are called as C functions. At the same time it allows designers to create all new parts in C, although this is not recommended as the designer will need to use two different programming languages and RTL parts will slow down the simulation.

(2) All-processes-on-C: means that synthesis and verification (including debugging) tasks should be done at the C source code. As an example we can compare this with a software compiler. In a software compiler, a designer does not have to debug the generated machine language (or, assembler language) directly. Similarly, in behavioral synthesis, a designer should not have to debug the generated RTL code. Our CWB environment allows a designer to debug the original C source code and the CWB model checker allows designer to write properties or assertions directly on the C source code.

7.2.2 Design Flow Overview

CWB targets general LSI systems which normally contain several CPUs or DSPs, dedicated hardware modules and some pre-designed or fixed RTL- or gate level IP modules, which are directly connected or through buses.

Initially, each dedicated hardware module such as an ECC encryption module is described in behavioral C. Once its functionality is verified using the C simulator and debugger, the hardware module is synthesized with our behavioral synthesizer. Configurable processors are also synthesized from their C description in our environment. Legend RTL modules are described as function, and handled as a black box. The CPU bus and bus interface circuits are automatically generated using a CPU bus library. After synthesizing and verifying each hardware module, our design environment allows designers to create a cycles-accurate simulation model for the entire system including CPUs, DSPs and custom hardware modules. With this simulation model, designers can verify both functionality and performance of their hardware design as well as the embedded software run on the CPU, DSP and/or generated configurable processors. Behavioral synthesis is quick enough to allow designers to repeatedly modify and synthesis the hardware modules and embedded software. The behavioral C source code can also be debugged with our formal verification, property/assertion model checker tool. Global properties and in-context (immediate) assertions are described for/in the C source code. The equivalence between behavioral C and generated RTL can be verified both in dynamic and static
way, as described later. Currently, the architectural level parallelization is left to the designer. The designer partitions the C source code into individual hardware modules and embedded software based on the performance result of the cycle simulation or FPGA emulation.

7.2.2.1 Synthesis Flow

Our design flow is shown in Fig. 7.1. A hardware design in extended ANSI-C (called “BDL”, or “Cyber-C”) [4], or SystemC is synthesized into synthesizable RTL with our “Cyber” behavioral synthesizer [1] with a set of design constraints such as clock frequencies, number and kind of functional units and memories. Usually RTL is handled as a black box, but if necessary, the RTL can also be fed to the behavioral synthesizer. The behavioral synthesizer can insert extra registers to speed up the original RTL and generate new RTL of smaller delay. It also generates a cycle accurate simulation models in C++ or SystemC. The behavioral synthesis can therefore be considered as a Verilog, VHDL, C, C++, and SystemC unification step.

The “Library Characterizer” generates delay and area information of the functional units and memories on a particular technology or FPGA.

A Behavioral IP library, called “Cyberware”, is also included in the synthesis environment. Any part of the behavioral IP can be encrypted for security purposes.
Wire delays of global wires between modules need to be analyzed carefully since those delays can be significant when the connected modules are placed far away. Our “RTL FloorPlanner [3]” takes the RTL modules generated by the behavioral synthesizer. Accurate timing information is extracted from the floorplanner and fed back to the behavioral synthesizer. The behavioral synthesizer reads the timing information and re-schedules the C code considering the timing information.

7.2.2.2 Verification Flow

The functionality of the hardware described in C can be verified at the behavioral level, while performance and timing are verified at the cycle-accurate level (or RTL) through simulation. Debugging the generated RTL is however not an easy task since C variables are shared in a register, and various optimizations are applied. We therefore provide a behavioral C source code debugger linked to our cycle-accurate simulation and FPGA emulation tool. After verifying each hardware module, the entire SoC is simulated in order to analyze the performance and/or to find inter-modules problems such as low performance through bus collision, or inconsistent bit orders between modules. Since such entire chip performance simulation is extremely slow in RTL-based HW-SW co-simulation, CWB generates cycle accurate C++ simulation models which can run up to hundred times faster than RTL models. Our HW-SW co-simulator [3] uses the generated cycle-accurate model for this purpose. The simulator allows designers to simulate and debug both hardware and software at the C source code level at the same time. If any performance problems are found, designers can change the hardware-software partitioning or algorithm directly at the C level, and can then repeat the entire chip simulation. This flow implies a much smaller and therefore faster re-design cycle than in a conventional RTL methodology. The C description is the only initial and final SoC description language of the entire design. This entire chip simulation can be further accelerated using an FPGA emulation board [5]. A “Testbench Generator” helps designers to run an RTL simulation with test patterns for behavioral C simulation faster and easier. Its inputs are test patterns for the C simulation and output a Verilog and/or VHDL testbench, which generates stimulus for the RTL simulation. It also creates a script to run commercial simulators to feed the behavioral test patterns and check the equivalence of outputs patterns between the behavioral and RTL simulation.

Another important feature of CWB is the formal verification tool, which is tightly linked to the behavioral synthesizer. With the behavioral synthesis information the formal verification tools can handle larger circuits than usual RTL tools and have C-source level debugging capability even though the model checker works on the generated RTL model. “C-RTL equivalence prover” checks the functional equivalence between a behavioral (un-timed or timed) C description and the generated RTL, using information of the optimizations performed such as loop unrolling, loop merge and array expansion performed by the behavioral synthesis. Without such information, the equivalence check is almost impossible for large circuits.
Designers can specify assertions or properties at the behavioral C level, similar to our cycle accurate simulator. Such behavioral level properties/assertions are converted into RTL ones automatically, and are passed to our RTL model checker.

CWB generates a power enhanced RTL model which estimates the power consumed by the design. A set of power libraries for different technology are provided and used with the generated RTL estimates that power for the selected technology.

A “QoR” synthesis report of the generated circuit shows a quick overview of the design quality. The report file includes area, number of states, critical path delay, number of wires and routability. This information is used for quick micro-architectural exploration as well as system architectural exploration. The system architecture explorer automatically generates different hardware architectures based on the preferences and constraints entered by the user (area, latency, power) at the C level. The designer can analyze the different generated architectures and finally choose the one that meets the design constraints at the smallest cost.

7.3 Behavioral Synthesis

To support the “all-modules-in-C” paradigm presented before, our behavioral synthesizer must cope with three types of circuits: (i) data-dominated, (ii) control-dominated, and (iii) control-flow intensive (CFI) ones. Data-dominated descriptions have many arithmetic operations and less control structures (e.g. only one loop), while control-dominated descriptions have many control-flow operations such as I/O activity in every cycle. A CFI description has a mix of arithmetic operations and control-flow constructs such as loops, conditional operations, jumps (’goto’ statements) and functions. Our synthesizer has three types of synthesis engines in order to support these varieties of circuit types: (i) automatic scheduling for CFI and data-flow circuits, (ii) fixed scheduling for control-dominated circuits, and (iii) pipeline scheduling for automatic pipelining or loop folding. Figure 7.2 shows a block diagram of CWB’s behavioral synthesizer. CWB supports various C-based language (e.g. BDL, SystemC, SpecC), and RTL as an input description. BDL is directly translated into our tree-structured Control Flow Graph (tCFG) [4], which is a kind of abstract structured expressing control structure of the behavior. Since SystemC and SpecC have different synthesis semantics than BDL, our “Parser/Translator” translates them into BDL semantics and generates the tCFG. In the same way, Verilog-HDL or VHDL is translated into the tCFG. A unique Control Data Flow Graph [2] is then created from the tCFG. All synthesis tasks are performed on those two data structures.

Control dominated circuits such as PCI I/F, DMA controller, DRAM controller, bus bridge, etc, require cycle-by-cycle behavioral description. For this type of circuits, specifying timing constraints for all inputs and outputs is a tough and complex job. Our extended C language called BDL can describe clock boundaries in a behavioral description, and is able to express very complex timing behaviors concisely. Such descriptions are synthesized with a “fixed scheduling” engine, which is fit for
complex control sequence with exceptional tasks with strict timing constraints. For the circuits, which require fixed sequential communication protocols but all other computations can be freely scheduled, “automatic scheduling” engine is used for synthesis.

For CFI circuit synthesis, the “automatic scheduling” engine is used. The quality of the synthesis is affected by the control flow structure, not just by the data flow. A smart scheduling algorithm is designed to overcome the effects of the programming style. For instance, Fig. 7.3 shows an example of global parallelization among multiple data-dependent conditional branches. These two branches cannot be parallelized in the form given in Fig. 7.3a, because of the control dependency between them. However, if the conditional operations “if (F1)” and “if (F2)” are transformed while scheduling, then they can be parallelized as shown in Fig. 7.3b. This implies that the scheduler will have to modify the control logic in order to obtain circuits with less latency while maintaining the data-flow intact.

Merging two branches into a single one using CDFG transformations is not as effective because the procedure is complex and the merging does not always lead to better results. In contrast, our approach uses a systematic scheduling algorithm without CDFG transformations. In other words, our scheduler schedules all operations in several basic blocks and several branches at the same time in a unique way, as if they were all operations in a single basic block. Our approach handles many other types of speculations, global parallelization with a method called “Generalized Condition Vector [6]”, which is extended version of “Condition Vector [2]”.

The “Pipeline scheduling” engine generates pipelined circuits from the initial C code with stall signals, which have various “Data Initial Intervals (DII). It also
Fig. 7.3 Parallelization of multiple branches for control-flow intensive applications (CFI)

speeds up loop execution by folding loop bodies like software loop pipelining. Global parallelization capabilities are very important even for loop pipelining. Loop carry variables that will be read in the next loop iteration should be scheduled into the states within the given DII cycles sequence. Parallelization beyond control dependencies is one key technique to make loop pipelining possible with a small DII.

7.4 Behavioral Synthesis Advantages Over Conventional Flows

The next sections describes in detail some of the advantages of behavioral synthesis over conventional RTL methodologies like hardware-software co-design, source code re-usability, application specific processor optimizations and automatic architecture exploration.

7.4.1 Shorter Design Period and Less Design Cost

Since C-based behavioral synthesis automates the functional design of hardware, it shortens the design cycle and at the same time shortens the design time of embedded software. Figure 7.4 shows the design cycle of two designs. The first uses the traditional RTL-based design flow and the second the proposed C-based design flow. The total design period and design men-month for the RTL-based design is larger than the C-based one, even though the gate size for RTL design (200K) is one third of that
for the C-based (600K) one. The hardware design period of the C-based design is 1.5 months, much shorter than the RTL-based design which takes 7 months. It needs to be stressed that the software design in the C-based design takes only 2 months while it takes 6 months for the RTL-based. This is due to the fact that the embedded software can be debugged before the IC fabrication using the hardware-software co-simulator. In RTL design, the software is usually verified on the evaluation board since RTL co-simulation is too slow even for this size of circuits. Lastly, C-based design allows very quick generation of simulation models for embedded software at a very early stage, allowing hardware and software to be concurrently designed both in C.

7.4.2 Source Code Reusability and Behavioral IPs

Another important aspect of C-based behavioral design is the high-reusability of behavioral models; we call this “behavioral IPs” or “Cyberware”. An RT level reusable module, called “RTL-IP”, can be successfully used for circuits of fixed performance such as bus interface circuits. However, RTL-IPs for general functional circuits such as encryption can only be used for a specific technology, since the RTL-IP’s “performance” is hard to adapt for newer technologies. For instance, an encryption RTL-IP at 200Mbps is difficult to be “upgraded” to perform encryptions at 800Mbps, because the RTL-IP structure is fixed and the logic synthesis tool is not able to reduce its delay by a forth. On the contrary, a behavioral IP is more flexible and more reusable than RTL-IPs, since it can change its structure...
and behavior allowing the synthesis tool can generate circuits of different performances by simply changing high level synthesis constraints such as number of functional units and clock frequencies. Table 7.1 shows how various circuits of different “clock-frequency” can be generated from a single behavioral IP. This IP is a BS broadcast descramblers (Multi2). All generated circuits satisfy the required performance (more than 80 Mbps) at various frequencies. Note that the highest clock circuit (108 MHz) uses less number of gates than the slow circuit (33 MHz). This never happens in RTL-IPs, which follow the area-delay tradeoff relation of logic synthesis. However, it is natural that a behavioral synthesizer generates a smaller circuit of higher clock frequency for the same performance, since less parallel operations are necessary to achieve the same performance at higher clock frequency.

Another important aspect is that for behavioral IPs it is much easier to modify their “functionality” and “interface” than for RTL-IPs. We designed two types of “Viterbi” decoders for mobile phone and satellite communications. The two required different Bit Error Rate, which is defined by several parameters such as encode rate and constraint bit length. Changing these parameters requires significant modification of the RTL-IP; however, only slight modification is necessary for the behavior IP.

Lastly it has to be noted that behavioral IPs sometimes generates smaller circuits than RTL IPs as behavioral synthesis shares registers and functional units for sequential algorithms such as the Viterbi decoder, but recent RTL designers do not share registers since such time multiplexed sharing makes RTL simulation and debug very difficult.

### 7.4.3 Configurable Processor Synthesis

Since chip fabrication cost has risen considerably, SoC are becoming as flexible as possible. For this purpose, recent SoC usually have several configurable processors besides a main CPU. These configurable processors should be small, have a high performance and low power consumption for a specific application. Such a configurable processor is also called Application Specific Instruction set Processor (ASIP). ASIPs employ custom instruction-sets to accelerate some applications. There are several commercial ASIPs, such as Xtensa [7] from Tensilica and Mep [8] from Toshiba. Their base-processor and co-processors for adding instructions are described in RTL and they are logic synthesized. In CWB we provide ASIP’s base
Table 7.2 Behavioral base-band DSP synthesis results

<table>
<thead>
<tr>
<th>MIPS(clock)</th>
<th>STB stream 72(108 MHz)</th>
<th>Base-band DSP 15(15 MHz)</th>
<th>Application DSP 60(60 MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Adding: 17</td>
<td>+ Adding: 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gate size</td>
<td>43K</td>
<td>20K</td>
<td>120K</td>
</tr>
<tr>
<td>Behavior</td>
<td>2.1KL</td>
<td>1.3KL</td>
<td>2.5KL</td>
</tr>
<tr>
<td>Generated RTL</td>
<td>13.0KL</td>
<td>11.4KL</td>
<td>26.0KL</td>
</tr>
<tr>
<td>Man-power</td>
<td>1.5 m-m</td>
<td>0.5 m-m</td>
<td>0.8 m-m</td>
</tr>
</tbody>
</table>

Table 7.3 Behavioral configurable processor synthesis

<table>
<thead>
<tr>
<th></th>
<th>Behavioral C-based</th>
<th>Manual RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>1.3 KL (1/7.6)</td>
<td>9.2 KL</td>
</tr>
<tr>
<td>Simulation</td>
<td>61.0 Kc/s(203×)</td>
<td>0.3 Kc/s</td>
</tr>
<tr>
<td>Pentium3@1 GHz</td>
<td></td>
<td>UltraSparc-II@450 MHz</td>
</tr>
<tr>
<td>Gate size</td>
<td>19 KG</td>
<td>18 KG</td>
</tr>
</tbody>
</table>

processor and supplementary instructions that are described fully in behavioral C, which are behavioral synthesized. This allows the base-processors and the addition of instructions to share functional units. This sharing leads to much smaller circuits than the conventional RTL-based ASIPs. For an ASIP base-processor, we added 24 instructions suitable for stream processing, such as CRC calculation, with only 25% area increase (34KG to 42KG) due to the of FU sharing.

C-based ASIPs are more flexible than RTL-based ones in terms of public register number, pipeline stages or interrupt policy. In Table 7.2, the synthesis results of three ASIPs are presented. All ASIPs were relatively small, but had enough performance to run the specific application due to the addition of custom instructions. All C-based ASIP designs required only as one tenth man-power of the RTL-based designs.

Table 7.3 shows comparison of C-based and manual RTL design for a configurable DSP design. RTL design flow. The two designs had comparable gate size and delay (RTL design is slightly better). The code efficiency of C-based design flow is shown to be 7.6 compared to the RTL design flow and a simulation speed-up of approximate 200, which leads to high reliability. We believe such advantages are much more important than slight area loss.

7.4.4 Automatic Architecture Exploration

Behavioral synthesis allows the creation of multitude hardware architecture for a unique C design. The user can specify a set of constraints which all architectures have to meet (e.g. area, latency, power) and a set of different architectures that meets those constraints will automatically be generated. The area-performance-power
trade-offs can be easily analyzed and the architecture that meets the constraints with the lowest cost can be chosen by the designer. This task is extremely time consuming if it is done at the RTL level as every single architecture requires a major re-work in the RTL code including component types and number of component instantiations. At the behavioral level this can be done by exploring the C code “attributes” of the most significant C code operations (those that will have the highest impact on the final architecture) like functions (e.g. inline expansion, sub-routine), loops (loop merge, unroll, unroll x-times, unroll completely) and mapping arrays as wired logic, registers or memories. Another aspect that is explored is the “global” synthesis options. What kind of scheduling policy is performed such as speculative scheduling, ASAP, ALAP scheduling of inputs and outputs, and which optimization algorithms (e.g. area-, latency-, delay-oriented) should be performed during behavioral synthesis. The third exploration step involves the maximum number of functional units available. This has a significant effect on the scheduler and therefore on the final design. To facilitate the trade-off analyzes the different architectures are displayed as a graph in the IDE’s GUI as shown on Fig. 7.5.

The exploration engine is based on a weighted probabilistic search algorithm, where the target options (area and performance) entered by the user are the probabilities that a specific synthesis option or attribute is selected. Each possible synthesis option and attribute has therefore been previously characterized in a library depending on its “usual” contribution to increase performance or area. A unique list of new attributes and synthesis options is generated for each new architecture, avoiding repetition of two equal designs.
Table 7.4 AES core system exploration example

<table>
<thead>
<tr>
<th>Design</th>
<th>Gates</th>
<th>Registers</th>
<th>Muxes</th>
<th>States</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>223,973</td>
<td>59,336</td>
<td>135,891</td>
<td>37</td>
<td>2.06</td>
</tr>
<tr>
<td>2</td>
<td>304,203</td>
<td>68,774</td>
<td>186,964</td>
<td>62</td>
<td>1.78</td>
</tr>
<tr>
<td>3</td>
<td>80,892</td>
<td>29,940</td>
<td>36,265</td>
<td>61</td>
<td>2.74</td>
</tr>
<tr>
<td>4</td>
<td>283,687</td>
<td>8,774</td>
<td>184,015</td>
<td>64</td>
<td>1.78</td>
</tr>
<tr>
<td>5</td>
<td>244,997</td>
<td>53,150</td>
<td>173,175</td>
<td>67</td>
<td>2.30</td>
</tr>
</tbody>
</table>

Table 7.4 shows an example of the architecture exploration of an AES core function which has about 800 lines of C code. The system explorer generates a user defined number of unique architectures (five in this case) based on the target selected by the user (e.g. minimize area, maximize performance).

7.5 System VLSI Design Example Using C-Based Behavioral Synthesis

Figure 7.6 shows a design example of a real complex SoC used at NECs cell phones generated with our behavioral synthesizer. This SoC is called MP211, or Medity [9], which has three ARM cores, one DSP, several dedicated hardware engines and various applications of mobile phone such as audio and video processing, voice recognition, encryption, Java and so on.
Wide ranges of circuits including control dominated circuits and data-intensive circuits were successfully implemented. The grey boxes (including bus) indicate modules that have been synthesized from C descriptions with the proposed behavioral synthesizer, while the white boxes are IP cores given in RTL format (some are legacy RTL components and some are commercial ones). All newly developed modules are designed with our C-based design flow. This example clearly illustrates that our C-based environment is able to design entire SoC designs, and not only algorithmic modules. C-based design flow became a standard ASSP development flow since 2003 at NEC, and several billion dollars worth of ICs have been taped out since.

7.6 Summary and Conclusions

This paper introduced the advantages of behavioral synthesis over traditional RTL methodologies in system LSI design on the hand CyberWorkBench. Faster development time, hardware-software co-simulation and development, easier and faster verification as well as automatic system exploration are some of these. Although many hardware designs are still very skeptical regarding behavioral synthesis the facts show that it is necessary and will sooner or later be a must in every complex hardware design flow. Winners will be early adopters of this methodology.

Currently, we are using behavior synthesis for most of our new designs and more system LSIs are verified with our C-based simulation.

Behavior synthesis tool is as mature as logic synthesis in the late 1980s, when designers started to use them widely RTL level design flows. However, it is taking time to make designers adopt this new design paradigm shifting from RTL “structural” domain thinking to “behavioral” domain thinking. Education and training on behavioral thinking for RTL designers is a crucial and difficult task.

Acknowledgments The authors would like to acknowledge the work of everyone at EDA R&D center, Central Research Laboratories at NEC Corporation, and NEC Information Systems Ltd., NEC Electronics Corp. NEC-HCL-ST for all their work developing CyberWorkBench and designing various chips with it.

References

7 “All-in-C” Behavioral Synthesis and Verification with CyberWorkBench


