An Interactive Design Environment for C-Based High-Level Synthesis of RTL Processors

Dongwan Shin, Andreas Gerstlauer, *Member, IEEE*, Rainer Dömer, *Member, IEEE*, and Daniel D. Gajski, *Life Fellow, IEEE*

***Abstract*—Much effort in register transfer level (RTL) design has been devoted to developing "push-button" types of tools. However, given the highly complex nature, and lack of control on RTL design, push-button type synthesis is not accepted by many designers. Interactive design with assistance of algorithms and tools can be more effective if it provides control to the steps of synthesis. In this paper, we propose an interactive RTL design environment which enables designers to control the design steps and to integrate hardware components into a system. Our design environment is targeting a generic RTL processor architecture and supporting pipelining, multicycling, and chaining. Tasks in the RTL design process include clock definition, component allocation, scheduling, binding, and validation. In our interactive environment, the user can control the design process at every stage, observe the effects of design decisions, and manually override synthesis decisions at will. We present a set of experimental results that demonstrate the benefits of our approach. Our combination of automated tools and interactive control by the designer results in quickly generated RTL designs with better performance than fully-automatic results, comparable to fully manually optimized designs.**

***Index Terms*—Embedded systems, high level synthesis, interactive design environment, register transfer level (RTL) processor, system-on-chip (SoC).**

I. Introduction

With ever increasing complexity and time-to-market pressures in the design of embedded systems, designers have moved the design to higher levels of abstraction in order to increase productivity. Ideally, the design process starts at the system level. However, each design must be refined through various design processes and implemented, eventually, at the lower levels. The task of register transfer level (RTL) design has been recognized as one of the major system design steps [1], [2].

Many years of research in RTL synthesis have been dedicated to the development of automatic synthesis tools. In these systems, designs are obtained with minimal user interaction. Typically, the only means of controlling the output of such systems is via cumbersome constraints expressed in terms of clocking, area, and timing.

Automating RTL synthesis is a very complicated issue. It is well known that the majority of synthesis tasks are NP-complete problems. Hence, the design time becomes large, the results are suboptimal, or resulting designs cannot satisfy the performance or area demands of real-world constraints. To develop a feasible approach for RTL synthesis, we have substituted the goal of a completely automated, "push-button" synthesis system with one that allows to maximally utilize the human designer's insights. This approach is called *interactive synthesis methodology*. In this approach, the designer can control the design process at every stage, observe the effects of design decisions, and manually override synthesis decisions at will. This is facilitated through a convenient graphical user interface (GUI).

The main contribution of this paper is to provide a design environment that supports interactivity of the design process. Due to the complexity of the design space, keeping the designer in the loop at every step is necessary to achieve required productivity and quality of results. This allows us to leverage human experience while automating tedious and error-prone parts of the process. We have divided the design process into a set of necessary steps and we provide a corresponding flow and tool set. An intuitive interface together with analysis and estimation tools for designer feedback provide both controllability and observability of the design at every step of the process, lacking in existing high-level synthesis solutions.

Hardware description languages (HDLs) such as Verilog HDL and VHDL are most commonly used as input to RTL design. However, system designers often write models using programming languages such as C/C++ to estimate the system performance, to verify the functional correctness of the design, and even to refine the design into an implementation [3]–[5]. C/C++ offers fast simulation as well as a vast amount of legacy code and libraries which facilitate the task of system modeling. To implement parts of the design modeled in C/C++ in hardware using synthesis tools, designers must then manually translate these parts into a synthesizable subset of an HDL. This process is well known for being both time consuming and error prone, but can be eliminated completely. The use of C-based languages to describe both hardware and software will accelerate the design process and facilitate the software/hardware migration. Hardware synthesis tools from C/C++ can then be used to map the C/C++ models into logic netlists.

The rest of this paper is organized as follows. Section II shows related work and Section III introduces our RTL design environment and the program flow of the proposed RTL synthesis tool. In Section V, we will go over our design methodology with a simple example. Section VI shows the experimental results. Section VII concludes this paper with a brief summary.

Manuscript received January 31, 2007; revised June 9, 2007.

D. Shin, A. Gerstlauer, and D. D. Gajski are with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697-2625 USA (e-mail: dongwans@cecs.uci.edu).

R. Dömer is with the Electrical Engineering and Computer Science Department, University of California, Irvine, CA 92697-2625 USA.

Digital Object Identifier 10.1109/TVLSI.2007.915390

1063-8210/$25.00 © 2008 IEEE

II. Related Work

RTL design and behavioral synthesis [a.k.a., high-level synthesis (HLS)], have been studied for more than a decade now [1], [2], [6]. In recent years, a few projects have been looking at means to use C/C++ as an input to current design flows [3]–[5]. Constructs are added to model coarse-grain parallelism, communication, and data types. These constructs can either be defined as new syntactic constructs, hence creating a new language [3]. They can also be implemented as part of a C++ class library [4]. In order to facilitate the mapping of C/C++ models into hardware, several tools exist that automatically translate C/C++ based descriptions into HDL either at the behavioral level or the RTL [5], [7].

Many automatic synthesis tools (a.k.a., *push-button* synthesis) have been developed, including HYPER [8], Olympus [7], OSCAR [9], SPARK [10], Synopsys Behavioral Compiler [11], Mentor Catapult-C [12], Forte Cynthesizer [13], and Cyber [5]. However, these tools provide no means to access the intermediate design models that are created during the synthesis process and thus to change or influence decisions by the designer. The only models accessible to the designer are the behavioral input model, the structural output model, and design constraints of the design.

Some interactive synthesis approaches [14], [15] address the importance of user-interaction with synthesis system. However, they have a fixed design flow, that is, the designer has to perform a sequence of synthesis tasks in a predefined order. Furthermore, a cycle-accurate simulation model with complex components is not available for the intermediate stages.

Accellera [16] defines modeling semantics for simulatable and synthesizable intermediate models at different stages in the synthesis process. Description of partial design decisions are possible as well. However, multi-cycle and pipelined functional units are not supported [17].

III. RTL Design Environment

In this section, we will describe our RTL design environment integrated in a system-level design flow. The environment provides synthesis, refinement, and exploration for RTL design as shown in Fig. 1. It includes a GUI and a set of tools to automate the design flow and perform refinement steps. In our flow, designers or algorithms of automatic tools can make decisions such as clock period selection, allocation, scheduling, and binding. The GUI allows designers to input and change such design decisions. It also enables the designer to observe the effects of the decisions and to manually override the decisions at will. Furthermore, the designers can make partial decisions and then run automatic tools to take care of the rest of the decisions.

During preprocessing, the behavioral description of custom hardware in C/C++ will be refined into a super finite state machine with data (FSMD) model where each state is a basic block of the original description. Also, some presynthesis optimization techniques including constant propagation, dead code elimination, and common subexpression elimination are integrated. The generated FSMD will be the input model of the RTL synthesis.

A performance analysis tool is used to obtain characteristics of the initial design such as the number of operations, variables,

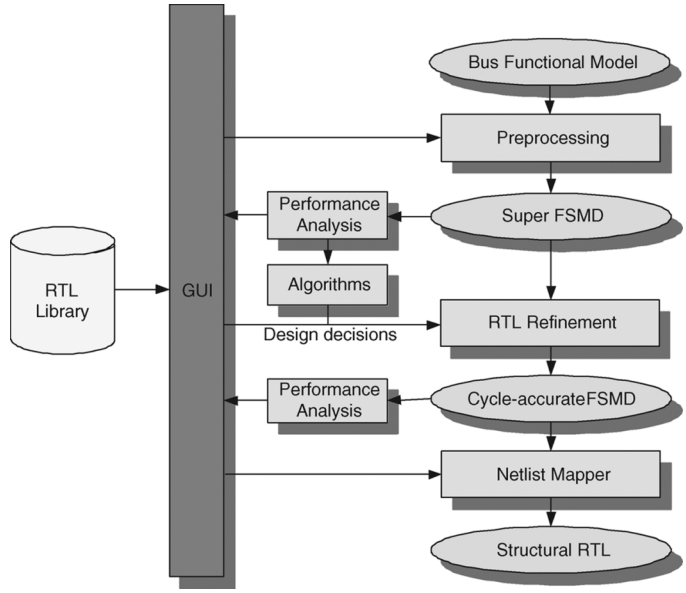

Fig. 1. RTL design flow.

and data transfers in each state, which serves as the basis for RTL design exploration. It also produces estimated quality metrics for RTL design such as the delay and power of each state and area of the design to help designer to make decisions on clock selection, allocation, scheduling, and binding.

Based on designer's decisions, the refinement tool will then automatically transform the super FSMD into a cycle-accurate FSMD description representing and implementing the designer decisions. Using the performance analysis tool, detailed quality metrics of the RTL design can be obtained from this model. With these quality metrics, designers can observe the effect of design decisions made and check whether or not design constraints are satisfied. In an iterative process, designers can then go back and change design decisions, rerun refinement, and obtain new metrics until constraints are met.

Finally, once a satisfying solution has been obtained, a final structural RTL model is produced by a netlist mapper, ready to feed into traditional design tools for logic synthesis.

A. Input Model

The input model of our RTL design is the bus functional model of the custom hardware PE as illustrated in Fig. 2. Such a model can be generated by system-level design tools [18], [19]. In this model, a hierarchy of sequential behavioral blocks inside the hardware PE describes its functionality. The hardware unit communicates through system busses with other PEs. The communication functionality is implemented by a protocol channel adapter, as shown in Fig. 2. The protocol channel adapter defines the timing-accurate implementation of each transaction over the system busses. It contains *read/write* function calls, which will be refined down to their RTL description.

B. RTL Models

Fig. 3 [20] illustrates the four major models encountered in our RTL design flow. The design flow starts with a behavioral specification in the form of program code, as shown in Fig. 3(a).

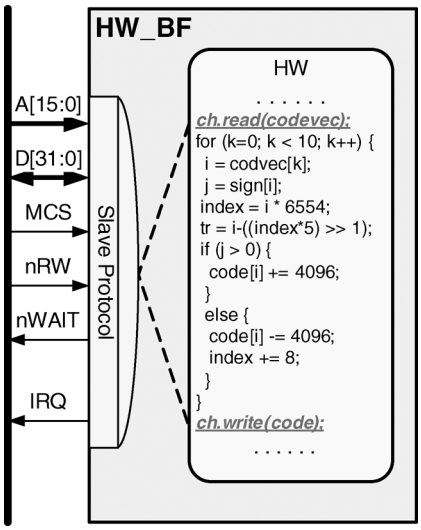

Fig. 2. Bus functional model for custom hardware.

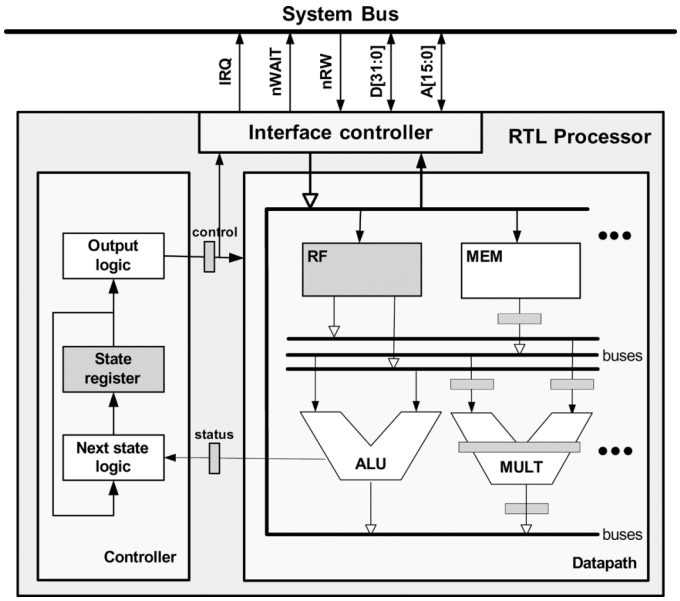

Fig. 4. RTL processor architecture.

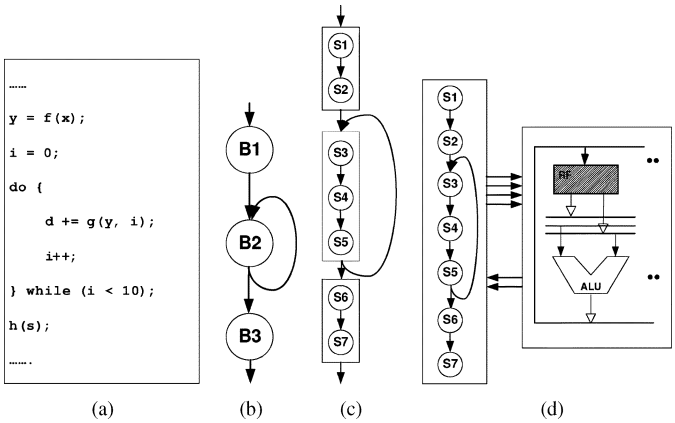

Fig. 3. RTL models during a typical behavioral synthesis flow. (a) Behavioral code. (b) SFSMD. (c) FSMD. (d) Structural model.

The behavioral model specifies the functionality of the intended design. This model is written as one process (a single thread of control) and usually contains no timing information.

As a first step towards implementation, the control flow in the design is then extracted and expressed in form of explicit state transitions, leading to the second model, a super-state finite-state machine with data (SFSMD), as shown in Fig. 3(b). Each node in the graph typically corresponds to one basic block in the previous model. As before, this model is written as a single untimed process.

Next, the core tasks of behavioral synthesis are performed, namely scheduling, allocation, and binding. During these tasks, RTL components and detailed timing are introduced in the model. Typically, an FSMD model is used to represent the design, as shown in Fig. 3(c). The previous super-states are now scheduled into actual clock cycles and represented by separate states. Thus, the FSMD model should now be clock cycle accurate and will exhibit the actual timing of the design when simulated.

We model an RTL design as an FSMD [16], which is an finite-state machine (FSM) model with assignment statements added to each state. The FSMD can completely specify the behavior of an arbitrary RTL design. The variables and functions in the FSMD may have different interpretations which in turn defines several different styles of RTL semantics [16], [20].

At the end of the RTL design flow, a netlist generator creates a structural model of the design, as shown in Fig. 3(d). In this structural model, the control unit drives a separate datapath consisting of registers, busses, and ALUs. All units in the design are connected by explicit wires and execute concurrently in separate processes. Due to the explicit concurrency and the exposed connectivity, a structural model is usually very complex and difficult to understand, maintain, and debug. Moreover, because of the many threads of control in the model, simulation is typically very slow.

C. Target Processor Architecture

Our target RTL processor architecture is shown in Fig. 4. It consists of a controller, a datapath, and an interface controller. The datapath consists of storage units such as registers, register files, and memories, and combinatorial units such as ALUs, multipliers, shifters, and comparators. These units and the input and output ports are connected by busses. During each state, the datapath takes the operands from storage units or input ports, performs the computation in the combinatorial units, and returns the results to storage units or output ports. The controller has a set of datapath control signals, status signals, and external signals. The datapath control signals select the operation for each components in the datapath. The status signals indicate when a particular value in the datapath is computed or when a particular relation between two data values stored in the datapath is satisfied. The external signals represent conditions in the external environment on which the model must respond, or signal to the environment that the model has reached a certain state or finished a particular computation.

In addition, the interface controller takes care of reading and writing data over the system bus. It is a state machine that drives and samples the actual bus wires according to the timing diagrams and constraints defined by the bus protocol. In our environment, the protocol implementation is taken out of the database and inserted as an interface to the RTL component. The main FSM controller then generates the control signals for the interface controller to perform bus transactions and read/write data from/to storage units in the datapath.

Each RTL processor follows this general architecture, although two RTL processors may differ in the number and type of control units and datapaths, the number of components and connections in the datapath, the number of states in the control unit, and the number and type of I/O ports.

An RTL processor may also be pipelined in several different ways: control pipelining, datapath pipelining, and unit pipelining.

1) *Control Pipelining*: Control can be pipelined by inserting latches or registers on control signals and/or status signals. Control registers are usually inserted in the last implementation stage, while a status register is frequently used from the beginning. However, the status register introduces at least one state delay in the control flow. In other words, the condition evaluation must be performed one state before it is used, since it is loaded into the status register in one state and used in the next. Similarly, the control register introduces one state delay in driving the datapath.
2) *Datapath Pipelining*: The datapath can be pipelined by inserting latches or registers on selected connections, such as before or after functional units. With datapath pipelining, the result of register transfers can be used only n states later, where n is the number of datapath stages.
3) *Function Unit Pipelining*: Each functional unit can be pipelined by dividing it into several stages and inserting latches between the stages. In the case of pipelined units, the result of the operation can be used only n states later, where n is the number of the latches in the functional unit.

IV. RTL Design

As shown in Fig. 1, we have three inputs to the RTL design tasks. The first input is the HW component's bus-functional description in a system level design language. The second input is an RTL component library that consists of a variety of RTL components including functional units, storage units, and busses. The final input is a set of synthesis decisions such as clock period, allocation, scheduling, and binding that define the RTL refinement task which will then be executed by the tool.

A. Preprocessing

In our interactive design environment, we attach the design decisions to the behavioral description of custom hardware in C/C++. We choose a three-address code representation to display the design and input the design decisions in our GUI. In addition, during the preprocessing step, the behavioral description of custom hardware is transformed to a SFSMD model. In the SFSMD, each state is a basic block of the original description where each state contains the basic block's data flow graph of operations in three-address form. In order to represent the SFSMD of the design, we have introduced a specific `fsmd` construct in SpecC [21]. Also, we introduced two constructs `piped` and `after` to represent pipelined and multicycle operations, respectively [20].

During the preprocessing step, some presynthesis optimization techniques including constant propagation, dead code elimination, and common subexpression elimination are also integrated.

B. RTL Component Library

The RTL component library [22] contains models of RTL units such as functional units, storage units, local busses, and protocol interface controllers for system interfaces. RTL units are also described in C/C++ or HDL form (Verilog/VHDL) for RTL netlist generation. RTL units will be used for RTL component allocation and the generation of the final RTL netlist.

Components generally have attributes and parameters. Component attributes describe characteristics or metrics for a component such as area, delay, cost, power, and so on. Attributes of a components are stored as an annotation attached to the component. All components in the RTL library can be parameterizable in bit width, size, etc. For a parameterized component, the designer selects values for each of the component's parameters during allocation. The value of attributes is also adjusted according to the selected parameters.

Generally, functional units can be pipelined, multicycled, and chained. Also, storage units can be pipelined or multi-cycled in our target architecture. The storage units can be composed of registers, register files, and memories with different latency and pipeline schemes.

C. Synthesis Decisions

Our refinement engine works on directions called the RTL synthesis decisions. The synthesis process can either be automated or interactive as per the designer's choice. However, the decisions must be provided to the refinement engine using a specific format. For the purpose of our implementation, we annotated the input model with the set of synthesis decisions. The refinement tool then detects and parses these annotations to perform the requisite model transformations. Based on these decisions, the refinement engine imports the required RTL components from the RTL component library and generates the cycle-accurate FSMD.

The synthesis decisions adjustable by the designer are the clock period, the component allocation (number and type of components), scheduling (control step for each operation) and binding (the functional unit for each operation, the source and target storage units for variables, and the used busses can be selected).

1) GUI for Interactive Decision-Making: The decisions can be made by designers interactively through GUIs and/or be made through automatic algorithms. The GUIs for interactive decision-making allows designers to 1) specify decisions; 2) override the decisions which are already made; or 3) partially assign decisions and automatic algorithms will fill in the rest of the assignments.

In order to help designers to make synthesis decisions interactively, we provide an *allocation window* and a *scheduling &*

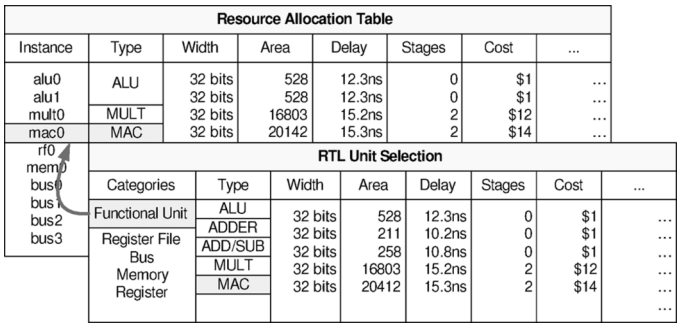

Fig. 5. Allocation window.

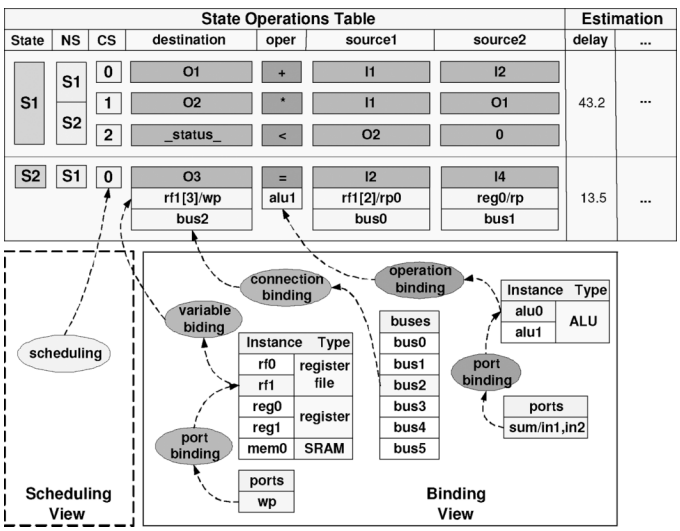

Fig. 6. Scheduling & binding window for an SFSMD.

binding window. In the allocation window, as shown in Fig. 5, the designer can see all RTL components in the RTL component library, select them, and set the parameters such as bit width, memory size, and so on [22].

The scheduling & binding window displays the SFSMD in *state-operations table* format which contains a series of states, each state containing a set of operations to be performed in the state, as shown in Fig. 6. The state-operations table displays the behavior of a design and all design decisions made in graphical format. The designer can modify all design decisions at any time in the design process in the state-operations table. In the table, *State* is the current state and *NS* is next state. *CS* is the control step of the expression which is relative to the start time of the state.

While not shown in Fig. 6, the table also shows statistics such as the lifetimes of all variables, occurrences of operations, the number of data transfers, and the critical path in a number of operations in each state. It also shows the as soon as possible (ASAP) and as late as possible (ALAP) control step for each expression in each state.

All expressions are scheduled at specified control steps in the scheduling view, which will be assigned to *CS* in the state-operations table. All operations are bound to functional units and their ports, which will be specified in the *oper* column. Also, all operand variables (*destination, source1, source2*) are mapped to storage units, read/write ports of the storage unit, and busses in the binding view. If the variables are mapped to memory, then the base address needs to be specified as well.

2) Automatic Decisions: Our design environment provides well-known scheduling and binding algorithms such as ASAP and ALAP, and list scheduling and register binding algorithms. It also provides the user with an interface to plug-in custom design decision algorithms. Algorithms take preassigned decisions together with estimates and fill in all remaining decisions. Often, efficient results are obtained when some critical decisions on scheduling and binding are partially made by the user and the rest of the decisions are made by the automatic plug-ins. After the algorithm has been run, the user can view and edit/override decisions made by the algorithm at will by use of the scheduling and binding windows.

D. RTL Refinement

RTL refinement generates different styles of RTL models following the Accellera [16] semantics.

During the refinement, each state in the SFSMD will be split down to several states depending on scheduling decisions. Variables and operations are replaced with storage units and functional units in the FSMD. The resulting FSMD model is then a cycle-accurate representation of the design.

During the netlist generation, the FSMD will be refined down to the structural RTL description of the design. In this step, storage and functional units are connected to busses in the datapath, and the signals to control the units in the datapath are introduced in the model [compare Fig. 3(d)].

E. Performance Analysis

Several synthesis metrics are implemented to help the designer or algorithms in deciding how to select the allocation and partition an SFSMD description into control steps. Two important metrics of design cost are operator occurrences and variable lifetimes. Operator occurrence metrics show the number of operations of each type used in each state. The maximum number of occurrences of a certain operator type over all states determines the required minimum number of functional units to perform that type of operation. Variable lifetime metrics identify states in which a variable holds a useful value. The maximum number of variables with overlapped lifetimes over all states determines the required minimum number of storage units.

After allocation, performance estimation calculates the delay and power consumption of each state and the area of the design within seconds. The designers can observe the effects of the decisions made and check whether or not design constraints are met.

V. RTL Synthesis Example

To illustrate the application of our methodology, we will walk through a simple design which computes the square-root approximation (SRA) [23] of two signed integers, a and b by the following formula:

$$\sqrt{a^2+b^2} \approx \max\left((0.875x+0.5y), x\right)$$

where $x = \max(|a|, |b|)$ and $y = \min(|a|, |b|)$.

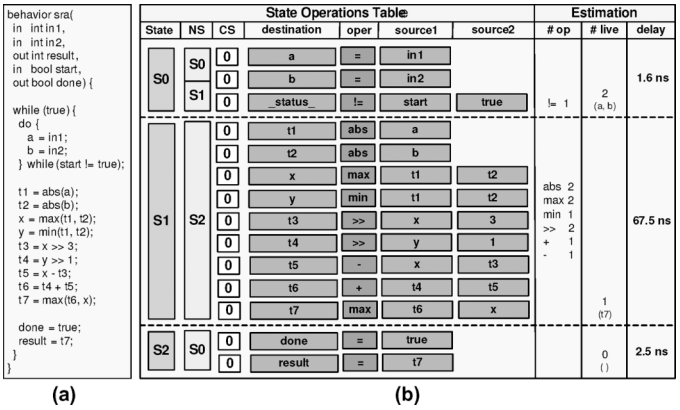

Fig. 7. Square root approximation example: (a) behavioral C/C++ code and (b) SFSMD.

TABLE I
RTL COMPONENT LIBRARY

component	functions	delay (ns)	area (# gates)
abs	abs	10.0	310
min	min	11.4	328
max	max	11.4	328
shift	<< / >>	9.0	798
add	+	10.5	307
sub	−	11.1	452
reg	register 1 rp/1 wp	1.6 (write) 2.5 (read)	332
mux	multiplexer	1.8	151
tbuf	tri-state buf	1.2	96

As shown in Fig. 7(a), this design has two input ports *in1* and *in2*, which are used to read integers a and b, and one output port *result*. The design reads the input ports and starts the computation whenever the input control signal *start* becomes equal to 1. After the computation is done, it makes the result available through the *result* port for one clock cycle. At the same time, it sets the control signal *done* to 1 in order to signal the environment that the data at the *result* port is valid.

A. Preprocessing

The first step for our methodology is to generate an SFSMD from the hardware part of the bus functional model. This step is done by an automatic tool which performs control/data flow analysis and splits C/C++ behavioral code into states based on basic blocks. The generated SFSMD model in state-operations form is shown in Fig. 7(b). Also shown in Fig. 7(b) are the quality metrics, number of operators (*# op*), number of live variables (*# live*), and state delay (*delay*).

B. Synthesis Decisions

The total execution time of a design can be defined as the product of the clock period used in the design and the maximum number of clock cycles. Hence, to optimize the performance of a design, it is important to select the clock period wisely, as well as to minimize the number of clock cycles [15].

Table I shows the component library that we use to implement our design. From the number of operators metric shown in

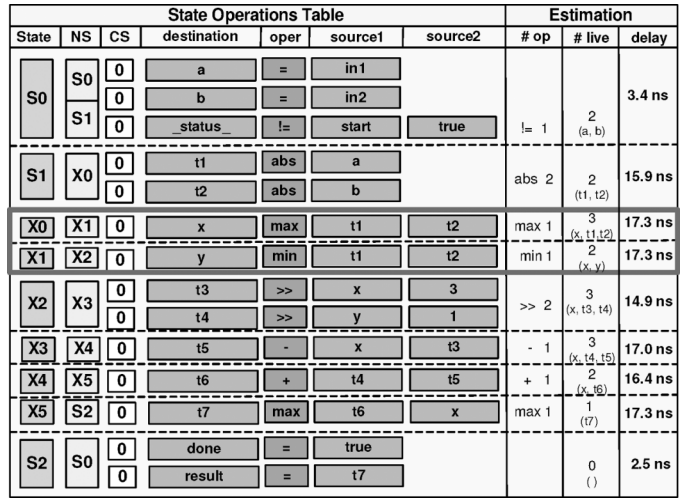

Fig. 8. SRA design after list scheduling.

Fig. 7(b), it is obvious that the current schedule requires at least two `abs` components[1] for the computation of an absolute value, two `min`/`max` components for the computation of the minima, maxima, and so on. $\#\ live$ shows the number of live variables at the end of each state, which is the minimum required number of storage units.

In total, unit area is estimated to be 4627 gates, which is the sum of the areas of all the required components including three I/O registers for variables a, b, and $t7$. At the same time, the state delay metric shows that the longest state delay is 67.5 ns. The total execution time would be $67.5 \times 3 = 202.5$ ns. After applying a list scheduling algorithm with resource constraints (four registers and four busses), the design needs nine cycles to execute, as shown in Fig. 8. The longest state delay is now 17.3 ns because multiplexers are introduced at the write port of the registers. The total execution time will be $17.3 \times 9 = 155.7$ ns. According to this result, the scheduling reduces the total execution time significantly. However, we have to introduce four registers for all variables in different states, which results in an increase of the area by one register and four multiplexers at the write port of four registers, a total of gates (936).

In order to reduce the area of the design, we can manually remove a `max` unit to perform two max operations in state X0 and state X5 using a single remaining `max` block.[2] However, we have to introduce multiplexers at the input ports of the `max` unit, and accept the additional delay of the multiplexers (1.8 ns). The longest state delay is then $17.3 + 1.8 = 19.1$ ns. Consequently, the area of the design reduces by 328 for the `max` unit, but increases by 151 for the multiplexer. In total, the area is now estimated to 5386.

As highlighted in Fig. 8, state `X0` and `X1` are split because the resource constraint list scheduling algorithm, which we implemented, does not check whether reading of two variables (`t1`, `t2`) can share the same busses. However, the two states can now be manually scheduled at the same control step, as shown in Fig. 9.

[1]Note that both `abs` operations are scheduled in state `S1`.

[2]This requires changing allocation, scheduling and binding by the designer, which takes less than a minute in our environment.

State Operations Table							Estimation		
State	NS	CS	destination	oper	source1	source2	# op	# live	delay
S0	S0	0	a	=	in1		!= 1	2 (a, b)	3.4 ns
		0	b	=	in2				
	S1	0	_status_	!=	start	true			
S1	X0	0	t1	abs	a		abs 2	2 (t1, t2)	15.9 ns
		0	t2	abs	b				
X0	X2	0	x	max	t1	t2	max 1 min 1	2 (x, y)	19.1 ns
		0	y	min	t1	t2			
X2	X3	0	t3	>>	x	3	>> 2	3 (x, t3, t4)	14.9 ns
		0	t4	>>	y	1			
X3	X4	0	t5	-	x	t3	- 1	2 (t4, t5)	17.0 ns
X4	X5	0	t6	+	t4	t5	+ 1	2 (x, t6)	16.4 ns
X5	S2	0	t7	max	t6	x	max 1	1 (t7)	17.3 ns
S2	S0	0	done	=	true			0 ( )	2.5 ns
		0	result	=	t7				

Fig. 9. SRA design after user decision.

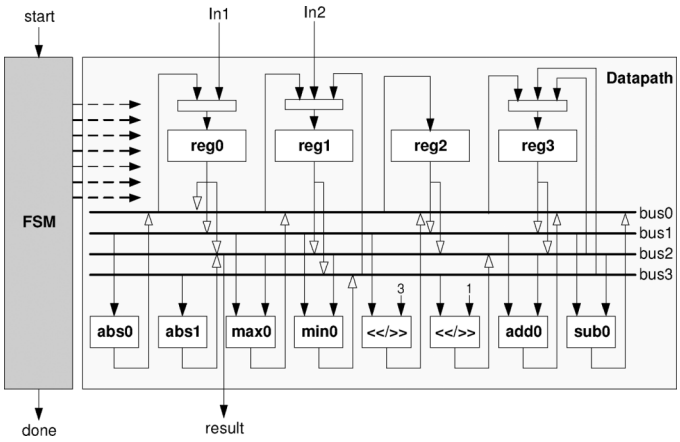

Fig. 10. Structural RTL of SRA.

Both post-optimization changes (to remove functional units and to merge states) were applied manually after running the list scheduling algorithm. Additionally, we optimized the required multiplexers by reassigning the connectivity manually. The resulting structural RTL representation is shown in Fig. 10. The maximum state delay of the design is $19.1 - 1.8 + 1.2 \times 2 =$ 19.7 ns because multiplexers are removed at the input ports of the `max` unit and tri-state buffers are introduced between the output ports of registers and functional units and the buses. The synthesis result for the example is shown in the first row of Table III. The total area of the design is 6798 where the area of the controller is included.

VI. Experimental Results

We have implemented our interactive RTL synthesis approach in C++ (algorithms, data structure) and Python (GUI). Fig. 11 shows the GUI for allocation to select the number of RTL components for the design. It shows various categories for the RTL components listed in the leftmost column, and the functional units and their parameters in the right-most column. The designer can browse the RTL components and select them and set parameters in the GUI.

Fig. 12 shows the GUI for scheduling and binding, which allows to set the control step of each operation and to map operations to functional units, variables to storage units, and connections to busses. The left-most columns in this GUI show the states and performance estimation (the number of operations, live variables, data transfers, delay, and power) for each state. The right-most columns show the behavior of the selected state S3 in three-address form. The designer can specify the control step in the `Cycle` column, and select binding information for each variable and operation through pull-down menus as shown in Fig. 12.

In order to evaluate the feasibility and benefits of our approach, we applied our RTL design environment to a variety of RTL design examples. As a first step, we used a set of simple, well-known examples that are part of a typical high-level synthesis benchmark suite. Benchmarks we used include square root approximation *(sra)*, greatest common divisor *(GCD)*, and differential equation solver *(DIFFEQ)*.

In addition, we applied our approach to a set of realistic, industrial-strength examples including different types of discrete cosine transformation (DCT) implementations: *DCT* (2-D DCT with matrix multiplication), *ChenDct* (2-D DCT implementing Chen algorithm [24]), *MP3Dct* (1-D DCT for MP3 Codec [25]), and *MP3ImDct* (1-D ImDCT for MP3 Codec [25]). Finally, the *Codebook* example is a design of a complete codebook search block that is part of a GSM Vocoder [26]. The GSM vocoder implements a standard for mobile communication employed worldwide for cellular phone networks. As such, our input model was based on the bit-exact reference implementation of the ETSI standard in ANSI C.

For `GCD`, `DIFFEQ`, `DCT`, `ChenDCT`, and `Codebook`, we used a simple handshake protocol to receive input data and send output data. The protocol is synthesized automatically. For the system interfaces of `MP3 DCT` and `MP3 IMDCT`, we implemented a simple bus interface protocol with address and data busses in the RTL database. Our tool inserts its implementation in the datapath and generates corresponding control words in the controller.

Table II lists the characteristics of the examples in terms of the number of operations (`#OPs`) in the input description (which is indicative of the data complexity of the design) and the number of basic blocks (`#BBs`) (indicative of its control complexity). Also, the types and quantities of each resource allocated to schedule and bind each design are given in Table II. The resources indicated in this table are shown as follows.

ALU	Arithmetic/logic unit used for arithmetic and logic ($+$, $-$, &, $\vert$, $\hat{}$, and $\sim$) or saturated arithmetic operations (in `Codebook` example).
ADD	Addition ($+$).
SUB	Subtraction ($-$).
ASU	Both addition and subtraction combined.
MULT	Multiplication ($\times$) in 1 stage pipeline fashion.
DIV	Division ($\div$) in five-stage pipeline fashion.
SQRT	Square root operation in five-stage pipeline fashion.
LU	Logic operations (&, $\vert$, $\hat{}$, and $\sim$).

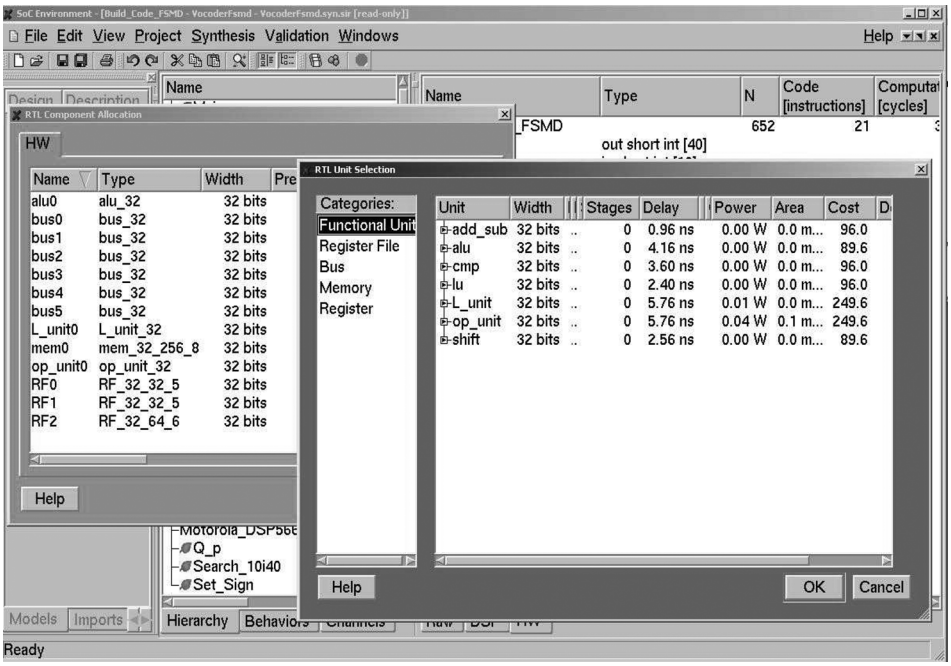

Fig. 11. GUI for allocation.

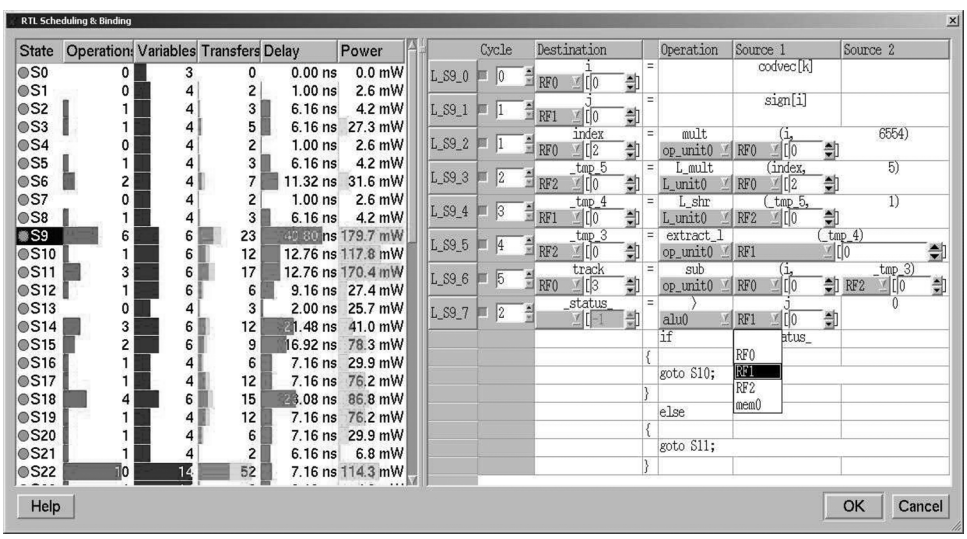

Fig. 12. GUI for scheduling and binding.

SHIFT Left/right shift operation ($\ll$, $\gg$).

CMP Comparison ($\geq$, $\leq$, , $<$, $==$, $!=$, !).

REG is a register with one read port and one write port, and *RF* is a register file with two read ports and one write port. *MEM* and *RAM* have one read port and one write port, and the *ROM* has one read port. The number in parenthesis indicate the sizes of register files and memories.

For our experiments, we implemented a heuristic based on a list scheduling algorithm [27] which performs scheduling and binding at the same time (`Automatic` in Table III). The heuristic considers the allocation of units and the number of ports of allocated units and busses. After applying the heuristic algorithm, we applied some manual optimization techniques to improve the performance of designs using our RTL design environment (`Automatic` + `Manual` in Table III). Finally, `Manual` shows the examples manually implemented by designers.

`MP3 DCT` and `MP3 IMDCT` were manually implemented to apply control pipelining by inserting pipeline registers on the output logic of the controller. This minimizes the clock period by reducing the delay from the output of output logic of the controller to its datapath. In addition, all control words were stored in program ROM. For both examples, we were able to achieve similar results by manually applying the same optimizations in our tool at the output of the heuristic algorithm.

For `Codebook`, the manual design inserts registers at the outputs of functional units to reduce the clock period and to implement data forwarding from the output of a functional unit to the input of other functions units [28]. In addition, special counters are introduced to calculate the index of arrays in the

TABLE II
CHARACTERISTICS OF SYNTHESIS EXAMPLES

Examples	# OPs	# BBs	Interface	resources
SRA	9 (3)	5	start/done	as shown in Figure 10
GCD	5 (3)	9	start/done	1 SUB, 1 CMP, 1 LU, 4 REGs
DIFFEQ	11 (10)	6	start/done	1 ASU, 1 CMP, 1 MULT, 16 REGs
DCT	21 (8)	22	handshake bus	1 ADD, 1 MULT, 3 COUNTs, 3 REGs, 3 RAM (64), 2 ROM (64)
ChenDCT	167 (42)	27	handshake bus	1 ADD, 1 SUB, 1 SHIFT, 1 MULT, 1 CMP, 16 REG, 1 RAM (128)
MP3DCT	330 (0)	3	handshake bus (DB)	1 ASU, 1 SHIFT, 1 MULT, 1 CMP, 2 RFs (64)
MP3IMDCT	195 (178)	49	handshake bus (DB)	1 ASU, 1 SHIFT, 1 MULT, 1 CMP, 2 RFs (64)
Codebook	604 (253)	197	handshake bus	1 ALU, 1 CMP, 1 MULT, 1 DIV, 1 MAC, 1 SQRT, 16 REGs, 1 RAM (2048)

TABLE III
SYNTHESIS RESULT

Examples	Automatic				Automatic + Manual					Manual				
	area	states	CP	cycles	area	states	CP	cycles	Perf.	area	states	CP	cycles	Perf.
SRA	6685	9	27.9	9	6798	8	27.1	8	13.7%	—	—	—	—	—
GCD	5154	9	35.7	34	5177	9	33.9	34	5.0%	—	—	—	—	—
DIFFEQ	17484	14	37.6	113	16314	12	31.7	93	30.6%	—	—	—	—	—
DCT	95408	44	60.1	6914	95173	29	50.9	5792	29.1%	114090	11	42.6	4225	56.7%
ChenDCT	32487	173	49.0	2469	31155	164	45.4	2129	20.1%	—	—	—	—	—
MP3DCT	85786	565	61.2	565	83780	308	54.9	308	51.1%	103357	308	47.7	308	57.5%
MP3IMDCT	93069	531	55.0	382	91669	356	50.9	209	49.4%	70239	356	44.3	209	55.9%
Codebook	991215	872	71.4	56492	991065	655	70.2	43195	24.8%	987162	315	55.5	33000	54.6%

memory (inside loops), which are accessed by row and column by two indices. In contrast, when running the example through our tool, we manually optimized critical hot spots in the output of the heuristic algorithm to reduce and merge states by sharing busses and ports.

We present the logic synthesis result obtained after synthesizing the RTL Verilog description generated by our Netlist mapper. Verilog output code was synthesized using the Synopsys *Design Compiler* logic synthesis tool. The `LSI − 10K` synthesis library is used for technology mapping, and components are allocated from the Synopsys DesignWare Foundation library.

The logic synthesis results are presented in terms of three metrics: the unit area (in terms of synthesis library used), the number of states (`states`) in the FSM controller, the critical path length (clock period, `CP` in nanoseconds) and the number of clock cycles (`cycles`) to execute the design. The critical path length is the length of the longest combinational path in the netlist as reported by the static timing analysis tool, and it dictates the clock period of the design.

The speedup (`Perf.`) of a design by `Automatic` + `Manual` and `Manual` is defined as follows:

$$\mathrm{Speedup}_x = \frac{\mathrm{CP}_{\mathrm{Auto.}} \times \mathrm{cycles}_{\mathrm{Auto.}} - \mathrm{CP}_x \times \mathrm{cycles}_x}{\mathrm{CP}_{\mathrm{Auto.}} \times \mathrm{cycles}_{\mathrm{Auto.}}} \times 100$$

where x is either `Automatic` + `Manual` or `Manual`.

For all experiments, the performace of the design using only the heuristic algorithm is the worst among all three implementations. After manually modifying the scheduling and binding result, the designs get about 30% improvement (40% degradation against manual design), especially in terms of clock cycles. This shows that even with some simple, straightforward post-synthesis modifications, results can be improved significantly. In some cases (*MP3DCT* and *MP3IMDCT*), applying our simple manual optimizations can even lead to results that come close to the fully manual implementations by human designers.

VII. SUMMARY AND CONCLUSION

In this paper, we proposed an interactive C-based RTL design environment targeting a generic RTL processor architecture. The environment is interactive and allows the designer to graphically view and edit allocation, scheduling and binding decisions. Automatic refinement tools take the decisions and implement them by generating corresponding output models at a variety of levels down to final synthesizable RTL code.

The environment supports automatic decision making through a plug-in mechanism. Users can select from a set of decision making algorithms to apply to all or parts of their design. Critical parts of the design, on the other hand, can be manually preassigned or post-optimized. Our environment is extensible in that new algorithms can be plugged in at any time, providing the designer with a synthesis tool box to choose from.

Both algorithms or designers can rely on a set of estimates to aid them in their decision making process. The environment includes profiling and analysis tools that return feedback about a variety of metrics like delay, power, variable lifetimes, or operator counts.

In summary, our environment provides a combination of automation and interactivity. In fact, the main contribution of our work is the interactivity of the synthesis process. Our environment takes full advantage of the designers' insight by allowing them to enter, modify or override all decisions at will. On the other hand, tedious and error prone tasks of code generation are automated. In addition, synthesis algorithms can be selectively applied to automatically implement or preassign noncritical parts of the design.

Results have shown the benefits of our approach. In all cases, including several industrial-strength examples, significant improvements could be obtained through simple interactive post-optimizations of synthesis algorithm outputs. As opposed to

push-button solutions, this proves the need for keeping the designer in the loop in order to leverage human insights for efficient design space exploration.

REFERENCES

[1] D. D. Gajski, N. Dutt, S. Y.-L. Lin, and A. Wu, *High Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.

[2] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[3] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*. Norwell, MA: Kluwer, 2000.

[4] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[5] K. Wakabayashi and T. Okamoto, "C-based SoC design flow and EDA tools: An ASIC and system vender perspective," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1507–1522, Dec. 2000.

[6] D. D. Gajski, *Silicon Compilation*. Reading, MA: Addison-Wesley, 1988.

[7] D. Ku and G. De Micheli, *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer, 1992.

[8] C.-M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: An interactive synthesis environment for high performance real time applications," in *Proc. Int. Conf. Comput. Des.*, 1989, pp. 432–435.

[9] B. Landwehr, P. Marwedel, and R. Dömer, "OSCAR: Optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proc. Euro. Des. Autom. Conf.*, 1994, pp. 90–95.

[10] S. Gupta, "SPARK: High-level synthesis using parallelizing compiler techniques," 2003 [Online]. Available: http://www.cecs.uci.edu/~spark/

[11] Synopsys Inc., Mountain View, CA, "Behavioral compiler," 2007 [Online]. Available: http://www.synopsys.com/

[12] Mentor Graphics Inc., Wilsonville, OR, "Catapult C synthesis," 2007 [Online]. Available: http://www.mentor.com/

[13] Forte Design Systems Inc., San Jose, CA, "Cynthesizer," 2007 [Online]. Available: http://www.forteds.com/

[14] A. A. Jerraya, I.-C. Park, and K. O'Brien, "AMICAL: An interactive high level synthesis environment," in *Proc. Euro. Des. Autom. Conf.*, 1993, pp. 58–62.

[15] H.-P. Juan, D. D. Gajski, and V. Chaiyakul, "Clock-driven performance optimization in interactive behavioral synthesis," in *Proc. Int. Conf. Comput.-Aided Des.*, 1996, pp. 154–157.

[16] Accellera C/C++ Working Group of the Architectural Language Committee, "RTL Semantics, draft specification," Feb. 2001 [Online]. Available: http://www.eda.org/alc-cwg/cwg-open.pdf

[17] S. Zhao and D. D. Gajski, "Defining an enhanced RTL semantics," in *Proc. Des. Autom. Test Conf. Euro.*, 2005, pp. 548–553.

[18] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski, *System Design: A Practical Guide With SpecC*. Norwell, MA: Kluwer, May 2001.

[19] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "System-on-chip environment (SCE Version 2.2.0 Beta): Tutorial," Center for Embedded Computer Systems, University of California, Irvine, Tech. Rep. CECS-TR-03-41, Jul. 2003.

[20] R. Dömer, A. Gerstlauer, and D. Shin, "Cycle-accurate RTL modeling with multi-cycled and pipelined components," in *Proc. Int. SoC Des. Conf.*, 2006, pp. 375–378.

[21] R. Dömer, A. Gerstlauer, and D. D. Gajski, "The SpecC Language Reference Manual, Version 2.0," presented at the SpecC Technol. Open Consortium, Japan, Dec. 2002.

[22] A. Gerstlauer, L. Cai, D. Shin, R. Dömer, and D. D. Gajski, "System-on-chip component models," Center for Embedded Computer Systems, University of California, Irvine, Tech. Rep. CECS-TR-03-26, Aug. 2003.

[23] D. D. Gajski, *Principles of Digital Design*. Englewood Cliffs, NJ: Prentice-Hall, 1997.

[24] W.-H. Chen, C. H. Smith, and S. C. Fralick, "A fast algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-25, no. 9, pp. 1004–1009, Sep. 1977.

[25] Underbit Technologies, Inc., San Diego, CA, "MAD: MPEG Audio Decoder, libmad-0.15.1b," [Online]. Available: http://www.underbit.com/products/mad/

[26] *Digital cellular telecommunication system: Enhanced full rate (EFR) speech transcoding*, GSM 06.60, European Telecommunication Standards Institute (ETRI), Nov. 1997.

[27] D. Shin and D. D. Gajski, "Scheduling in RTL design methodology," Center for Embedded Computer Systems, University of California, Irvine, Tech. Rep. CECS-TR-02-11, Apr. 2002.

[28] A. Gerstlauer, S. Zhao, D. Gajski, and A. Horak, "SpecC system-level design methodology applied to the design of a GSM vocoder," Information and Computer Science, University of California, Irvine, Tech. Rep. ICS-TR-99-11, Mar. 1999.

Dongwan Shin received the M.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1997, and the Ph.D. degree in information and computer science from University of California, Irvine (UCI), in 2004.

Since 2004, he has been an Assistant Project Scientist with the Center for Embedded Computer Systems, UCI. From 1997 to 1999, he was with LG Semincon, Co., Ltd., Seoul, Korea. His research interests include system-level design automation, high-level synthesis, and low-power system design.

Andreas Gerstlauer (M'04) received the Dipl.-Ing. degree in electrical engineering from the University of Stuttgart, Stuttgart, Germany, in 1997, and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine (UCI), in 1998 and 2004, respectively.

He is currently an Assistant Researcher with the Center for Embedded Computer Systems, UCI. His research interests include electronic system-level design languages, methodologies and tools, system modeling, and embedded hardware and software synthesis.

Rainer Dömer (M'03) received the Ph.D. degree in information and computer science from the University of Dortmund, Dortmund, Germany, in 2000.

He is currently an Assistant Professor with the Electrical Engineering and Computer Science Department, University of California, Irvine (UCI). He is also a member of the Center for Embedded Computer Systems, UCI. His research interests include system-level design and methodologies, embedded computer systems, specification and modeling languages, system-on-chip design, and embedded software.

Daniel D. Gajski (LF'08) received the Dipl.Ing. and M.S. degrees in electrical engineering from the University of Zagreb, Zagreb, Croatia, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.

After 10 years as a Professor with the University of Illinois, Chicago, he has joined the University of California, Irvine, where he presently holds The Henry Samueli Endowed Chair in Computer System Design. He directs the UCI Center for Embedded Computer Systems, with a research mission to incorporate embedded systems into automotive, communications, and medical applications. He has authored over 300 papers and numerous textbooks, including *Principles of Digital Design* (Prentice-Hall, 1997) that has been translated into several languages.