C-based Interactive RTL Design Methodology

Dongwan Shin, Andreas Gerstlauer, Rainer Dömer and Daniel Gajski

Technical Report CECS-03-42
December 1, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, doemer, gajski}@cecs.uci.edu

C-based Interactive RTL Design Methodology

Dongwan Shin, Andreas Gerstlauer, Rainer Dömer and Daniel Gajski

Technical Report CECS-03-42
December 1, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, doemer, gajski}@cecs.uci.edu

Abstract

Much effort in RTL design has been devoted to developing "push-button" types of tools. However, given the highly complex nature of RTL design, interactive design space exploration with assistance of tools and algorithms can be more effective. In this report, we propose an interactive RTL design environment, targeting a generic RTL processor architecture including pipelining, multicycling and chaining. Tasks in the RTL design process include clock definition, component allocation, scheduling, binding, and validation. In our interactive design 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 also provide a simultaneous scheduling and binding algorithm to automate RTL synthesis process. In the end, we present a set of experimental results that demonstrates the benefits of the proposed approach.

Contents

List of Figures

C-based Interactive RTL Design Methodology

Dongwan Shin, Andreas Gerstlauer, Rainer Dömer and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

Much effort in RTL design has been devoted to developing "push-button" types of tools. However, given the highly complex nature of RTL design, interactive design space exploration with assistance of tools and algorithms can be more effective. In this report, we propose an interactive RTL design environment, targeting a generic RTL processor architecture including pipelining, multicycling and chaining. Tasks in the RTL design process include clock definition, component allocation, scheduling, binding, and validation. In our interactive design 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 also provide simultaneous scheduling and binding algorithm to automate RTL synthesis process. In the end, we present a set of experimental results that demonstrates the benefits of the proposed.

1. 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 RTL design has been recognized as one of the major design steps [GDLW92].

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, or the results are suboptimal, 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).

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 and to verify the functional correctness of the design, even to refine the design into implementation [GZD+00] [GLMS02] [WO00]. 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 a HDL. This process is well known for being both time consuming and error prone. Moreover, it 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.

At the output of RTL design, many commercial and academic tools have been based on multiplexer-based architectures, where all data transfers among RT components are achieved through dedicated connections with multiplexers [Syn]. As the size of a design increases, the performance of the multiplexer-based architecture becomes slower than that of bus-based architecture. Our interactive RTL design environment targets a bus-based architecture, also known as RTL processor [Acc01]. The RTL processor is universal processor architecture which includes pipelining at different levels, multicycling and chaining.

The rest of the report is organized as follows: section 2 shows related work and section 3 introduces our RTL de-

sign environment and the program flow of the proposed RTL synthesis tool. In section 4, we will go over our design methodology with a simple example. In section 5, we will introduce a simultaneous scheduling and binding algorithm to accelerate synthesis process. Section 6 shows the experimental results. Section 7 concludes the report with a brief summary.

2. Related Work

In the recent years, a few projects have been looking at means to use C/C++ as an input to current design flows [GZD+00] [GLMS02] [WO00]. 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 register transfer level (RTL) [Mic99] [WO00] [SÓ1] [Gup03].

Wakabayashi et al. [WO00] has developed C-based high-level synthesis system, *Cyber*, that takes behavioral description language (BDL) which is extension of C language for hardware. *Cyber* is integrated their C-based SoC design environment and also provides cycle-accurate C++ model for validation of the system design.

SpC [SÓ1] addressed mapping of C code with pointers and `malloc/free` into hardware. In hardware, a pointer is not only the address of data in memory, but it may also reference data mapped to registers, ports or wires. Pointer analysis is used to find the set of locations each pointer may reference in a program at compile time. The values of the pointers are then encoded, and branching statements are used to dynamically access data referenced by pointers. Dynamic memory allocation and deallocation are supported by instantiating hardware memory allocators tailored to an application and a memory architecture. Several optimizations may also be performed. A heuristic algorithm is presented to efficiently encode the values of the pointers. Compiler techniques may also be used to reduce storage before loads and stores.

Gupta [Gup03] has developed C-based high-level synthesis framework, *SPARK*, that employs a set of parallelizing compiler techniques and synthesis transformations to improve the quality of high-level synthesis results. The compiler transformations have been re-instrumented for synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost model. *SPARK* takes behavioral ANSI-C code as input, schedules it using speculative code motions and loop transformations, runs an interconnect-minimizing resource binding pass and generates a finite state machine for the scheduled design graph. Finally, a backend code generation pass outputs synthesizable register-transfer level (RTL) VHDL. The *SPARK* methodology is particulary targeted to control-intensive microprocessor functional blocks and multimedia and image processing applications.

The methodology adopted in the *SPARK* system is based on a toolbox approach. Core transformations for code motion and loop transformations are implemented. Heuristics can then be designed which use these transformations either interactively with the help of the user, or based on some algorithm, and try to improve the scheduling and resource sharing results.

Some interactive synthesis approaches [JPO93] [JGC96] addressed the importance of user-interaction with synthesis system. The AMICAL [JPO93] allows the user to mix automatic and manual design. The user may start a design manually and ask the AMICAL to finish it. Alternatively, the user can execute the synthesis tasks step by step. At each step, the user has the choice to continue the synthesis automatically or manually. Yet, AMICAL has fixed design flow, that is, the user has to perform a sequence of synthesis tasks in the order of scheduling, chaining, allocation and the architecture generation.

The ISE [JGC96] has attempted to address physical design issues by allowing the user to start floorplanning early in the design process and generating feedbacks from the physical level to help the user making design decisions at behavioral and structural levels. It also provides automatic algorithms to either perform one design task for the user or to simply finish the design by completing the design tasks. The user can manually override the design decisions of synthesis algorithms.

3. RTL Design Environment

In this section, we will describe our RTL design environment integrated in a system-level design flow.

3.1. System Design Flow

Figure 1 shows the system level design flow [APY+03], in which the designer uses C/C++ not only as a specification or modeling language but also as the final implementation language, since we would like to avoid double coding for simulation and synthesis.

The system design process starts with a specification model written by the designer to specify the desired system functionality. During the architecture exploration, the designer selects a set of processing elements and maps the computation behavior of the specification onto PEs. Architecture exploration refines the specification model into the transaction level model. The transaction level model describes the PE structure of the system architecture and the mapping of the computation behaviors onto the PEs. The communication between PEs still is done by abstract message passing channels.

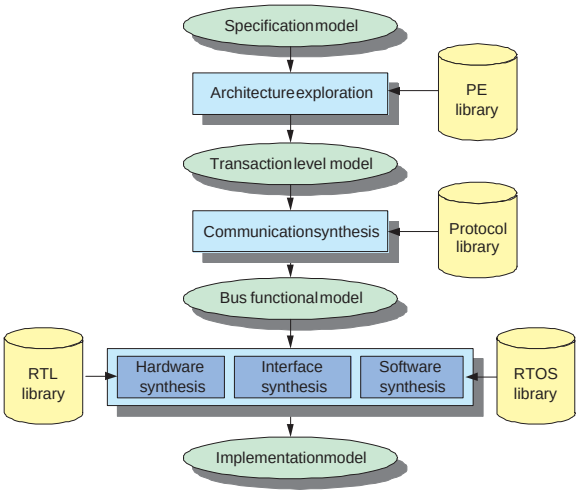

Figure 1. System design flow

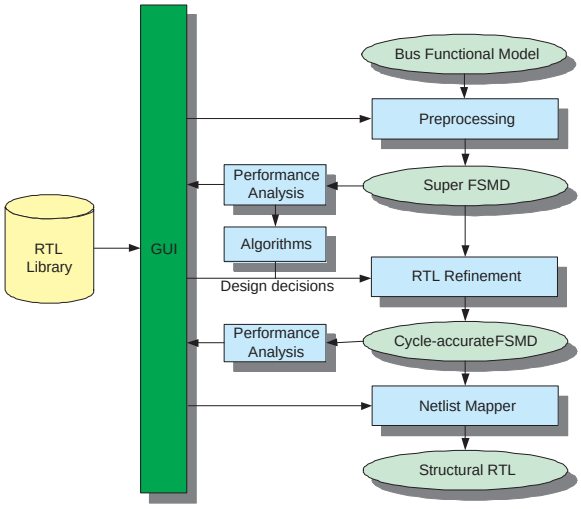

Figure 2. RTL design flow

Architecture exploration is followed by communication synthesis which selects a set of system busses and protocols, and maps the functionality of communication to the system busses. Communication synthesis creates the bus functional model which reflects the bus architecture of the system.

The bus functional model is the starting point for the three backend tasks: hardware synthesis, interface synthesis and software synthesis. Depending on the type of PE, software is compiled and assembled to instruction-set model, or hardware is synthesized to a structural RTL model. Either way a clock-cycle accurate model of the PE is obtained. The result of this backend process is the implementation model which is a cycle-accurate, structural description of the RTL/IS architecture of the whole system.

3.2. RTL Design Flow

The RTL design environment provides synthesis, refinement and exploration for RTL design as shown in Figure 2. It includes a graphical user interface (GUI) and a set of tools to facilitate 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. Further, 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 FSMD model. Also some presynthesis optimization techniques including constant propagation, dead code elimination, 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 and data transfers in each state, which serves as the basis for RTL design exploration. It also produces 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.

The refinement tool then automatically transforms the FSMD model based on relevant design decisions. Finally, the structural RTL model is produced by a netlist mapper, ready to feed into traditional design tools for logic synthesis, etc.

As shown in Figure 2, we have three inputs to the RTL design tasks. The first input is the HW components description in system level design language in bus-functional model which shows the communication architecture of a system. 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 clock period, allocation, scheduling and binding that define the RTL refinement task which will then be executed by the tool.

3.3. Preprocessing

During preprocessing, the behavioral description of custom hardware in C/C++ will be refined into a super FSMD model. The whole description can be considered to be a SFSMD with one state. But generally, the description can be divided into any number of the super state of any size.

The description consists of reading inputs (I), writing outputs (O) and executing expressions (EXP), which use

intermediate storage variables (V). Using these elements, SFSMD can be described syntactically in the same way as FSMD. The basic difference between the two models is that SFSMD does not restrict the size of the algorithm-parts assigned to a state, whereas FSMD does. This is, because SFSMD does not correspond to hardware at all, where as the states of FSMD correspond to clock cycles.

3.4. Finite State Machine with Data (FSMD)

For an exact description of FSMD, some sets have to be defined.

S:	set of *states*
I:	set of *inputs*
O:	set of *outputs*
V:	set of *storage variables*
EXP:	set of *expressions*: *functions* which give results depending on *storage variables* V and *operators* OP: $EXP = \{V, OP\} = \{f(x,y,z,\ldots)\|x,y,z \in V\}$
OP:	set of operators used in EXP
$STAT$:	set of *status expressions*: *logic relations* between two expressions from the set EXP: $STAT = \{Rel(a,b)\|a,b \in EXP\}$

Referring to the above definitions, the data processing (first item in above list) is described by function h:

$$h : S \times (I \cup STAT \cup EXP) \rightarrow (V \cup O)$$

There needs to be a set of initializing values $V0$ for the variables V when starting the FSMD, because the expressions in EXP read them. There also has to be an initial state $S0$. The next state is determined in a similar way:

$$f : S \times (I \cup STAT) \rightarrow S; V0, S0$$

$V0$ and $S0$ are needed again, because $STAT$ depends on EXP, and EXP depends on V.

With these definitions, an FSMD is described by:

$$< S, S0, (I \cup STAT \cup OP), V, V0, O, h, f >$$

The above definition of a FSMD can be given in tabular form with a state and output table as shown in Figure 5.

3.5. Input Model

The input model of our RTL design is the bus functional model of the custom hardware PE as illustrated in Figure 3, which was generated by system-level design tools. In this model, a hierarchy of sequential behavioral blocks inside the hardware PE describes its functionality. The hardware unit communicates data through system busses

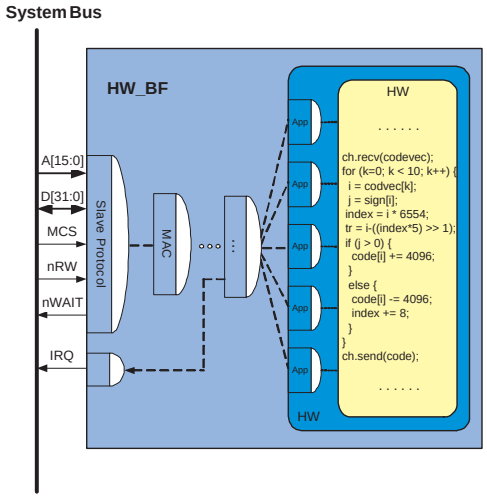

Figure 3. Bus functional model for custom hardware

with other PEs. The communication functionality is implemented by layers of the protocol stack including protocol layer, MAC layer, application layer as shown in Figure 3. These channels define the timing-accurate implementation of each transaction over the system busses. These channels are function calls which will essentially be inlined into the code during preprocessing [GCS+03].

3.6. RTL Component Library

The RTL component library [GCS+03] contains models of RTL units such as functional units, storage units and local busses. RTL units are also described in C/C++. 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 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 are pipelined or multicycled in our target architecture. The storage units can be composed of registers, register files and memories with different latency and pipeline schemes.

3.7. Synthesis Decisions

The 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 input 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 [GCS+03]. 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 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 (a) specify decisions (b) override the decisions which are already made by the designers or automatic algorithms (c) partially assign decisions and automatic algorithms will fill in the rest of decisions.

The GUI also allows automatic algorithms being plugged in. Thus it is easily extendable because designers can select an algorithm from a list of plug-in algorithms such as ASAP, list and force-directed scheduling and graph coloring for binding and so on.

3.7.1 GUI for Interactive Decision-making

In order to help designers to make synthesis decisions interactively, we provide an *allocation window* and a *scheduling & binding window*. In allocation window as shown in Figure 4, designer can see all RTL components in the RTL component library, select them and set the parameters such as bit width, size of array and so on [GCS+03]. The

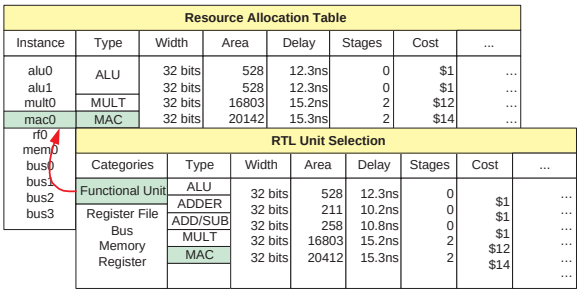

Figure 4. Allocation window

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, shown in Figure 5. The state-operations table displays the behavior of a design and all design decisions made in graphical format. This is, the designer can modify all design decisions at any time in the design process in the state-operations table.

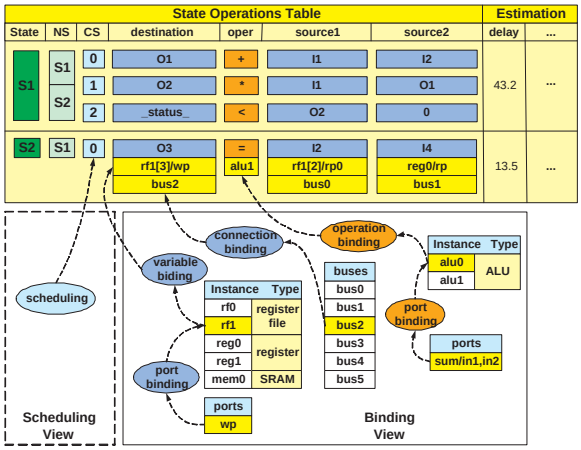

Figure 5. Scheduling & Binding window for an SFSMD

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.

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 number of operations in each state. It also shows the ASAP and 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.

Designers can input, modify all decisions and override decisions which algorithms made through automatic tools in scheduling & binding window. Furthermore, the designers can partially specify some of the decisions and then algorithms take care of the rest of decisions still meeting the specified designer's decisions. Furthermore some of the decisions can be omitted at all if the model is fed into traditional RTL design tools such as Design Compiler. For example, scheduling decision is made but binding is not. Then the traditional RTL tool will make decisions for binding. In order to do this, we generate a cycle accurate model in Verilog.

3.8. Performance Analysis

In our RTL design methodology, some synthesis metrics are implemented to help the desinger make synthesis decision. The synthesis metrics should be measured according to how much synthesis decision is made. Our performance analysis, therefore, is divided into two modes: pre-synthesis analysis and post-synthesis estimation.

Pre-synthesis analysis profiles an RTL design and collects statistics information to help the designer decide how to select allocation and partition a super FSMD description into control steps. Three important metrics of design cost are operator occurrences, variable lifetimes, data transfers. Operator occurrences metric shows 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 lifetimes metric identifies 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. Data transfer metric shows the number of data transfers to perform opertions. The maximum number of data transfers of operations over all operations determines the the required minimum number of buses.

Post-synthesis estimation reflects synthesis decisions to the RTL design and calculates delay and power consumption of each state and area of the design. After allocation, we can calculate these metrics through initial mapping of operations to units using the allocation information. The scheduling and binding decision will give more accurate estimation for the design.

3.9. Target Processor Architecture

Our architecture, RTL Processor, is shown in Figure 6. It consists of a controller, a datapath and an interface controller. The datapth 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. The datapath takes the operand from storage units or input ports, performs the computation in the combinatorial units, and returns the results to storage units or output ports during each state.

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 satisfied or when a particular relation between two data values stored in the datapath is satisfied. The external signals represent that conditions in the external environment on which the model must respond or identify to

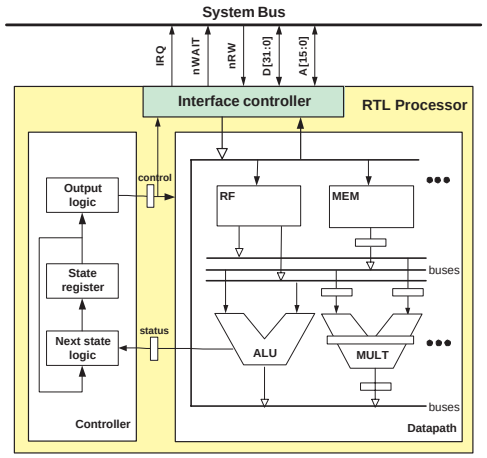

Figure 6. RTL processor architecture

the environment that the model has reached a certain state or finished a particular computation.

The selection of operands, operations and the destination of the result are controlled by the controller by setting proper values of datapath control signals. The datapath also indicates through status signals when a particular value is a particular storage unit or when a particular relation between two data values stored in the datapath is satisfied. The input ports can be connected directly to register or storage units or to any other component in the datapath including the output ports. The output ports could be used for possible connections to other RTL processors through outside busses or directly through point-to-point connection.

Similar to the datapath, the controller has a set of input and a set of output signals. There are two types of input control signals: external signals and status signals. External signals represent the conditions in the external environment on which the model must respond. There are also two types of output control signals: datapath control signals and external signals. The control signals select the operation for each components in the datapath, which the external signals identify to the environment that the model has reached a certain state or finished a particular computation. A controller consists of state register and next-state and output logic. Next-state logic generates the value for the state register in the next clock cycle while output logic generates the value of control and external signals.

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.

A RTL processor may also be pipelined in several different ways:

1. *Control pipelining*: By inserting the latches or regis-

ters on control signals and/or status signals. Control registers are usually inserted in the last implementation stage, while status register is frequently used from the beginning. However the status register introduces at least one state delay. 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 conditional evaluation.

2. *Datapath pipelining*: 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 pipeline stages in the functional unit.

4. Interactive RTL Synthesis Example

To illustrate the application of our methodology, we will walk through a simple design which computes the square-root approximation (SRA) [Gaj97] of two signed integers, a and b by the following formula:

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

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

4.1. Synthesis Decisions

The maximum execution time of a design can be defined as product of the maximum length of clock period used in the design and 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 [JGC96].

Table 1 shows the component library that will be used in implementing our design. From the operator occurrences

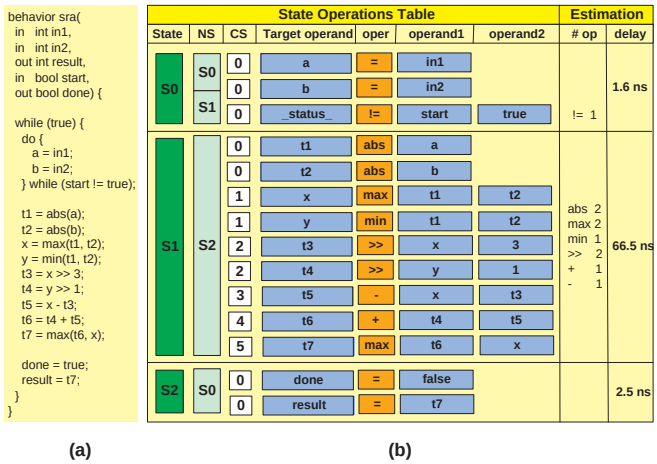

Figure 7. Square root approximation example (a) behavioral C/C++ code (b) SFSMD

Table 1. The RTL component library

component	functions	delay (ns)	area (# gates)
abs	abs	10.0	233
min	min	11.4	357
max	max	11.4	357
shift	$<<$ / $>>$	9.0	673
add	$+$	10.5	330
sub	$-$	11.1	591
add_sub	$+/-$	12.6	1056
reg	register 1 rp/1 wp	1.6 (write) 2.5 (read)	324
mux	multiplexer	1.8	151
tbuf	tri-state buf	1.2	96

metric shown in Figure 7 (b), it is obvious that the current schedule requires at least two components for the computation of an absolute value, two components for the computation of the maxima, and so on. Therefore unit area is estimated to be 4776 gates, which is the sum of the areas of all the required components including 3 registers for variables, a, b, $t7$. At the same time, the state delay metric shows that the longest state delay is 66.5 *ns*. The maximum execution time would be $66.5 \times 3 = 199.5$ *ns*. If we apply the ASAP scheduling, the design needs 8 cycles to execute as shown in Figure 8. The longest state delay is 15.5 *ns* and the maximum execution time will be $15.5 \times 8 = 124.0$ *ns*. According to this result, we can reduce the total execution time significantly. But we have to introduce 8 registers for all variables in different states, which results in increase of the area of the design by $324 \times 8 = 2592$.

In order to reduce the area of the design, we may use a max unit to perform two max operations in state X0 and

state X4. We have to introduce multiplexers in the input ports of the max unit and the additional delay of the multiplexers (1.8 *ns*). The the longest state delay would be $15.5 + 1.8 = 17.3$ *ns*. The area of the design reduces by $357 - 151 = 206$.

State Operations Table							Estimation	
State	NS	CS	Target operand	oper	operand1	operand2	# op	delay
S0	S0	0	a	=	in1			1.6 ns
		0	b	=	in2			
	S1	0	_status_	!=	start	true	!= 1	
S1	X0	0	t1	abs	a		abs 2	14.1 ns
		0	t2	abs	b			
X0	X1	0	x	max	t1	t2	max 1	15.5 ns
		0	y	min	t1	t2	min 1	
X1	X2	0	t3	>>	x	3	>> 2	13.1 ns
		0	t4	>>	y	1		
X2	X3	0	t5	-	x	t3	- 1	15.2 ns
X3	X4	0	t6	+	t4	t5	+ 1	14.6 ns
X4	S2	0	t7	max	t6	x	max 1	15.5 ns
S2	S0	0	done	=	false			2.5 ns
		0	result	=	t7			

Figure 8. SRA design after ASAP scheduling

Figure 9 shows a screenshot of our RTL design environment for the above simple example. We uses 4 registers for variables and 4 buses for data transfers between functional units and storage units. The binding is done by the designer. The maximum state delay of the design would be $17.3 + 1.8 + 1.2 \times 2 = 21.5$ *ns* because multiplexers are inserted at the write ports of the registers and tri-state buffers are introduced between the output ports of registers/functional units and the buses. The synthesis result for the example is shown in Table 4. The area of the design is 8874 because the area of the FSM of the design is included.

5. Synthesis Algorithms

It is tedious and error-prone job for the designers to make all decisions. In order to help the designers to make decisions, we provide automatic algorithms to complete design tasks. Also, we provide APIs for algorithm developers to implement algorithms in our design environment.

5.1. Internal Data Structure

In our RTL design methodology, input description is FSMD representation, in which each state has computation with assignments and control constructs and information on next states. Therefore, a state in FSMD can be represented by a control/data flow graph for computation.

The CDFG has been used for the internal representation of RTL synthesis tool since mid-1980s and has many variations. Basically, a CDFG has data flow information to describe the operations and their dependencies and has control flow information which is related to branching and iteration constructs. It can be hierarchical or non-hierarchical, polar or non-polar, and cyclic or acyclic.

Our CDFG structure for RTL design methodology is hierarchical and acyclic polar graph, which is shown in Figure 11. The acyclic graph makes it easy to implement the graph algorithm, because it has no loop. The polar graph has the single-entry and single exit property using no-operation (source node/sink node in our graph) and makes it easy to build hierarchical graph. The node *S* in Figure 11 represents the no-operation node. The top *S* is the source node and the bottom *S* is the sink node. In the CDFG, the

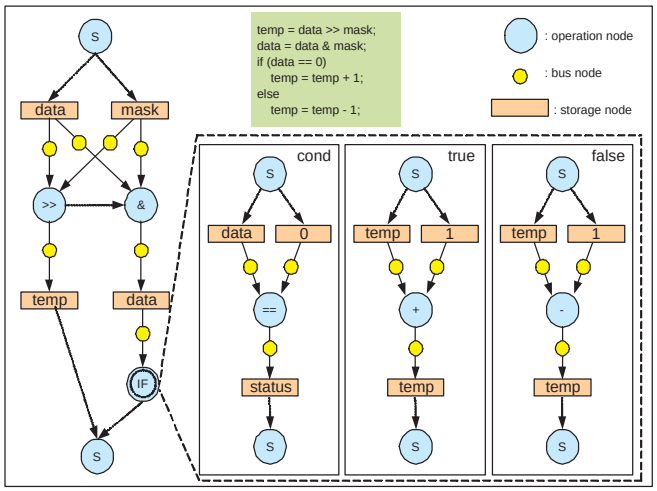

Figure 11. CDFG for unmapped (style 1) RTL description)

edge has the dependency information between nodes such as control dependency and data dependency. The node has all information except the flow information. The node is divided into the non-hierarchical node and the hierarchical node. The non-hierarchical node has the datapath operation information such as operation node to perform arithmetic/logic operation, storage node to store the data, bus node to transfer the data between functional unit and storage unit, control node to generate the status information of datapath, and state transition node to store state transition information in finite state machine. In Figure 11 shows the operation node which is the white circle node, storage node which is the shaded rectangular node, bus node which is the small shaded circle node between operation node and storage node. The hierarchical node is divided to the module node to represent the structural hierarchy in the RTL description, branch node to represent branching information and loop node to represent the iteration information.

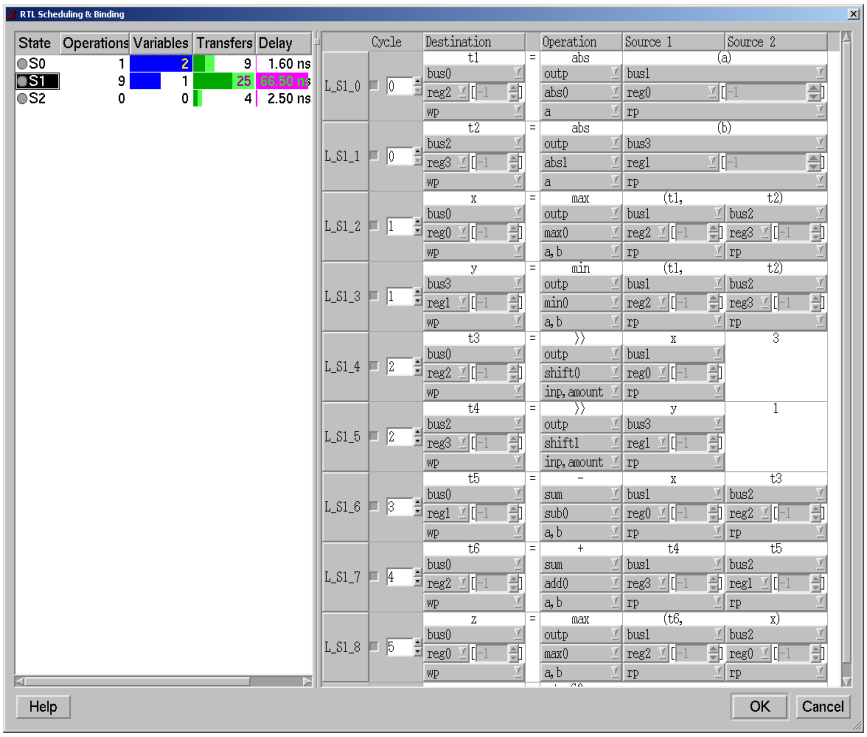

Figure 9. A scheduling and binding result of SRA

The branch node (`if` node) and loop node (`for` node) are shown in Figure 11.

5.2. Clock Selection and resource allocation

The maximum execution time of a design can be defined as product of the clock period used in design and the maximum number of clock period used in the design and 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 [JGC96]. Moreover, the number of clock cycles required to finish all operations in a design depends on the clock period. Therefore a bad choice of the clock period could severely affect the performance of the design. In our methodology, the clock selection is done by user.

Resource allocation is also important step in RTL synthesis. The number of resources can be determined by automatic tool or user [GDLW92]. In our RTL design methodology, resource allocation is performed by user.

5.3. Scheduling and binding algorithm

we describe simultaneous scheduling and binding algorithm which solves scheduling and binding problems together. This algorithm is greedy but simple and easy to implement. However, our methodology are independent of scheduling and binding algorithms and can use any other algorithms such as force-directed heuristic as well.

5.3.1 Problem Definition

Given:

1. A behavior represented by state transition graph, *STG(S, T)*, where *S* is state in FSMD and *T* is state transition among states.

2. Each state *S* contains hierarchical control/data flow graph, *CDFG(V, E)*, where *V* is a set of vertices representing operations, storages, buses, and hierarchical nodes such as branch and loop, and *E* is dependency between nodes.

3. A component library containing functional units, storage units and buses characterized by type, area, delay, pipeline states and so on. In addition, storage units have the number of read/write ports.

4. clock period and resource allocation, such as number of functional units, storage units, buses and read/write ports of storage units.

Determine:

1. control step of each node in a behavior

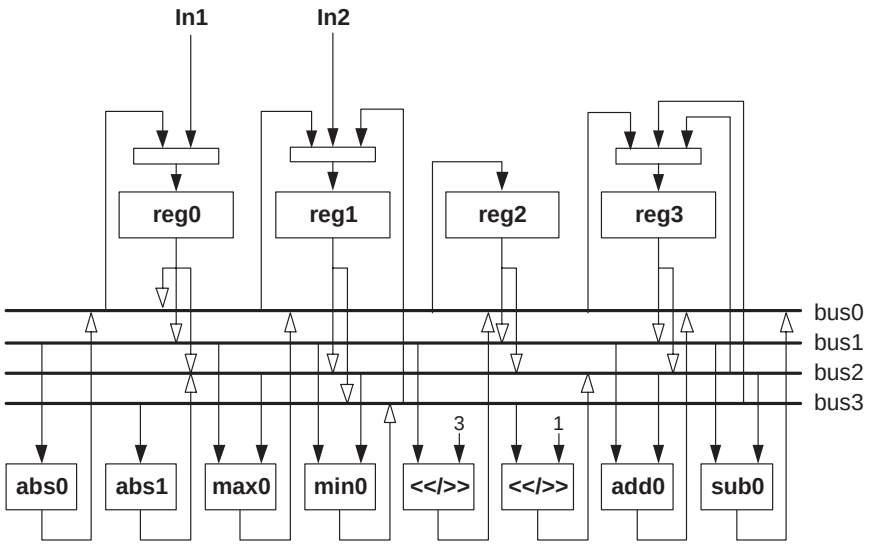

Figure 10. A datapath for SRA

2. resource selection for each node but hierarchical node

Such that:

1. the number of control steps is minimized.

2. the resource allocation constraint is satisfied.

In the proposed RTL design methodology, the scheduling plays a major role in refining from behavioral RTL to exposed-control RTL by re-scheduling each state in FSMD in the behavioral RTL description. The scheduling algorithm is divided into two layers: one is scheduling of states in FSMD and the other is scheduling and binding in CDFG.

5.3.2 State Scheduling

The scheduling of states determines the order of each state in FSMD and reflects resource reservation tables of the already scheduled states to next states which are affected by scheduling result of predecessor states. For example, if the states have multicyle operations and pipeline operations with more than 1 cycles delay, the next states will be affected by these previous states.

The state scheduling algorithm is shown in Algorithm 12. The state scheduling algorithm performs breadth-first search to find next state to be scheduled in FSMD. During RTL synthesis, we maintain the resource reservation table, which contains all resources such as number of RTL units, number of ports of each unit, number of buses and address space for storage units available for scheduling and binding. During state scheduling, resource reservation table for each state is updated by considering the resource reservation table of scheduled predecessor states.

Figure 12. State scheduling(STG, R_o): state scheduling algorithm

```
1:  // get reset state from STG
2:  Get reset state s0 ∈ STG;
3:  Sr = Sr + s0;
4:
5:  while (Sr is not empty) do
6:      Get front state s in Sr;
7:      Sr = Sr − s;
8:      sold = s;
9:      Rs = GetResourceTable(Ro, predecessors of s));
10:     snew = Sched_Bind(CDFGs, Rs);
11:
12:     // scheduling or binding of the state is changed
13:     if (snew != sold) then
14:         Append successors of s to Sr;
15:     end if
16: end while
17:
```

Each state calls the scheduling algorithm (Sched_Bind(G_s, R_s) to schedule nodes in CDFG. In state scheduling algorithm, we use candidate list and resource reservation table as follows.

- resource reservation table R_o, R_s: are original resource utilization table based on resource allocation and resource reservation table for state s, respectively.

- candidate states S_r: are those states which need to be re-scheduled because scheduling result of their prede-

cessors is changed.

In this scheduling algorithm, S_0 is reset node which is first executed after reset is deasserted.

5.4. Simultaneous scheduling and binding algorithm

The simultaneous scheduling and binding is to schedule and bind operations in each state. Because resource allocation like number of FUs, the ports of storage units and buses, is given by the designer, the resource-constrained scheduling and binding should be done. The aim of this task is reduce the number of states at minimal hardware cost. Our algorithm allows for resources to be shared amongst multiple operations, while component selection allows a mixture of fast and slow components to be used in the design. The components are selected such that the fast and expensive components are used for critical operations, and the slower ones are used for non-critical operations.

The algorithm traverses all states in FSMD and also the control-data flow graph of each state. It schedules one 3-address expression(operands and an operation) at a time. The states are ordered and scheduled by breadth-first search.

The heuristic for simultaneous scheduling and binding a basic block `Sched_Bind (`$CDFG(V,E)$`, `R_s`)` is listed in Figure 13.

This heuristic takes as input CDFG ($G(V,E)$) and resource reservation table R_s. The control step *cs* is maintained and is relative value to the first control step in each state.

The heuristic starts by collecting a list of available or ready expressions, V_r. Available expressions are a set of expressions whose data dependencies are satisfied and can be scheduled in the current cycle. Our heuristic has to maintain the unfinished expressions U_{cs}, which contains expressions started at earlier cycles and whose execution is not finished at control step *cs*. If the execution delay of an operation is 1 or less, the operation should not be included in the set of unfinished operations.

Method `FindAvailableResource (`v`, `cs`, `R_s`)` gets a resource which expression v can be bound to in resource reservation table (R_s) at control step *cs* still meeting resource constraint. It has to look ahead control step to check available resources for the expression because functional unit and storage unit can be multicycled or pipelined. When the functional unit of an operation is selected, the storage unit of the target operand variable which is output of the expression to write value, is also determined.

In the proposed algorithm, the number of ports of storage units and buses which are used in the specified control step, will be determined when the expression is scheduled, because the functional unit will use the ports and the buses in order to read data at the start time and to write data at the end time of the operation. In other word, the data transfer will occur at the start and the end of execution of the operation. The read time of the operand variable will be changed according to the start time of execution of the operation, which will read data from the storage unit. The write time of the storage unit is the same as the end of the execution of the operation, which will write data to the storage unit.

Figure 13. Sched_Bind ($CDFG(V,E)$, R_s): simultaneous scheduling and binding algorithm

```
1:  V_r ← all available expressions in CDFG
2:  while (V_r is not empty) do
3:    V_old = V_r;
4:    for (each node v ∈ V_r in highest priority) do
5:      // find available resource for node in resource table
6:      r_v = FindAvailableResource (v, cs, R_s);
7:      if (r_v exists) then
8:        Schedule v at cs and bind it with r_v.
9:        Update resource reservation table R_s at cs with
          r_v;
10:       Update ready list V_r and unfinished list U_cs with
          v;
11:       V_r = V_r − v;
12:     end if
13:   end for
14:   // no nodes are bound or
15:   // resource table in cs is exhausted
16:   if (V_old == V_r or R_s in cs has no available resource)
      then
17:     cs = cs + 1;
18:     Update resource reservation table R_s at cs;
19:     Update ready list V_r and unfinished list U_cs;
20:   end if
21: end while
```

5.4.1 Priority function

The list scheduling algorithms are classified according to the selection step. A priority list of the operations is used in choosing among the operations, based on some heuristic measure. Our proposed algorithm has two priority functions. One is for node selection among ready node list. The other is for resource selection from library.

1. node selection: the ready node list is sorted by the priority: urgency, mobility, number of successors in decreasing order, to select the node among the ready node list.

2. resource selection: to select resource of operation/operands, cost function in library is utilized. The

designer selects cost function according to the latency of unit, size of unit, whether or not it's pipelined.

The ready node list is sorted by the priority list for node selection and the resource reservation table is sorted by the priority list for resource selection.

5.4.2 Scheduling and binding Process by an Example

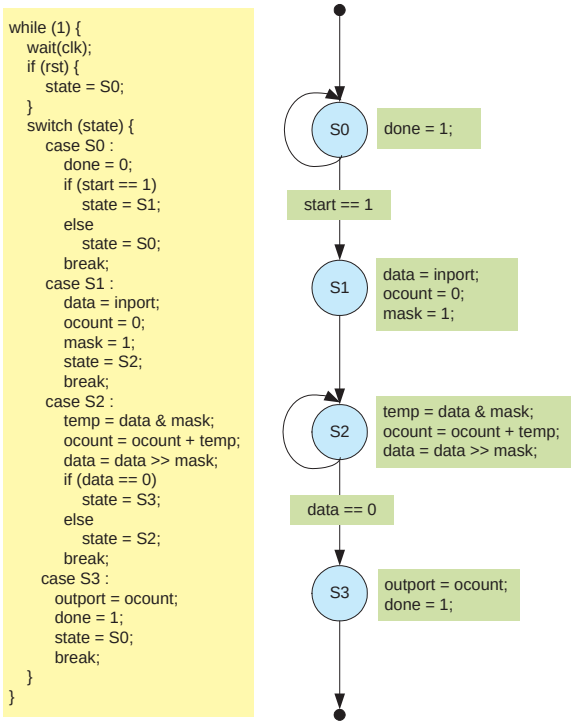

Figure 14. FSMD for one's counter

To explain the proposed algorithm, we will use one's counter as an example, shown in Figure 14, which calculates the number of ones in given number. It takes one input variables and generates one output result. The left side of this figure shows SpecC code for one's counter example and the corresponding FSMD is shown in the left side of this figure. Before scheduling, this FSMD consists of 4 states. State `S0` is reset state, and if reset is asserted, FSMD will enter this state first. State S2 has self loop to calculate number of one's in `data` variable until `data` is equal to 0. The Figure 15 shows the target datapath organization for the one's counter, which consists of two 2-stage pipelined ALUs (ALU0 and ALU1) and one register file and 3 buses. The functional unit ALU0 can perform bitwise and operation and addition operation in 2 cycles, and ALU1 can perform left/right shift and comparison operation in 2 cycles. The regiser file with two read ports and one write port is neither pipelined nor latched. Figure 16 shows the CDFG of

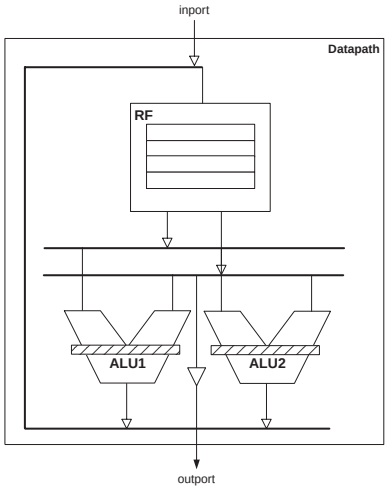

Figure 15. Target datapath organization for ones's counter

state `S2` which is generated from FSMD in Figure 14. The CDFG has 4 ALU operations and 8 storage nodes and 12 bus nodes. Figure 2 shows the control step according to the proposed algorithm. The 1st column represents the control steps. The next 2 columns represent the ready operations for each type of functional unit. The next 4 columns represent the resource reservation table, which has the number of resources used in each control step. In BUS column, the left value shows the number of buses which is used to transfer the result of functional units to storage units, the right value shows the number of buses which is used to transfer the input data for functional units from storage units. In RF column, the 1st value represents the number of the used write ports, and the other value shows the number of the used read ports in register file. The last two columns represents the scheduled operations and unfinished operation in current control step. According to the proposed algorithm, all nodes in CDFG are scheduled using this table. The latency of the scheduled CDFG is 5 control steps. The scheduled CDFG is shown in Figure 17. If we change 2-stage pipelined ALU0 to 3-stage pipelined ALU0, the scheduling result will be changed as shown in Figure 18 and in Figure 3. In Figure 18, operation + should be executed in 3 cycles but the control step of the state `S2` should be finished in `cs4`. The `S2` has self loop, then the remaining two cycles of the operation + will be performed in `cs1` and `cs2` in the state `S2`.

6. Experimental Results

Based on the described methodology and algorithms, we have developed a RTL design environment and refinement

Table 2. Scheduling process for one's counter

	ready		resource reservation table				scheduled	unfinished
	ALU0	ALU1	ALU0 (1)	ALU1 (1)	RF (1/2)	bus (1/2)		
cs1	&&	>>	0	0	0/0	0/0		
		>>	1	0	0/2	0/2	&&	
cs2		>>	0	0	1/0	1/0		&&
			0	1	1/2	1/2	>>	
cs3	+		0	0	1/0	1/0		>>
			1	0	1/2	1/2	+	
cs4		==	0	0	1/0	1/0		+
			0	1	1/2	1/2	==	
cs5			0	0	0/0	0/0		==
			0	0	0/0	0/0		

Table 3. Scheduling process for one's counter (II)

	ready		resource reservation table				scheduled	unfinished
	ALU0	ALU1	ALU0 (1)	ALU1 (1)	RF (1/2)	bus (1/2)		
cs1	&&	>>	0	0	0/0	0/0		
			1	1	0/2	0/2	&& >>	
cs2			0	0	1/0	1/0		&& >>
			0	0	1/0	1/0		
cs3		==	0	0	1/0	1/0		&&
			0	1	1/2	1/2	==	
cs4	+		1	0	0/0	0/0		==
			0	1	0/2	0/2	+	
cs5			0	0	0/0	0/0		+
			0	0	0/0	0/0		
cs6			0	0	1/0	1/0		+
			0	0	1/0	1/0		

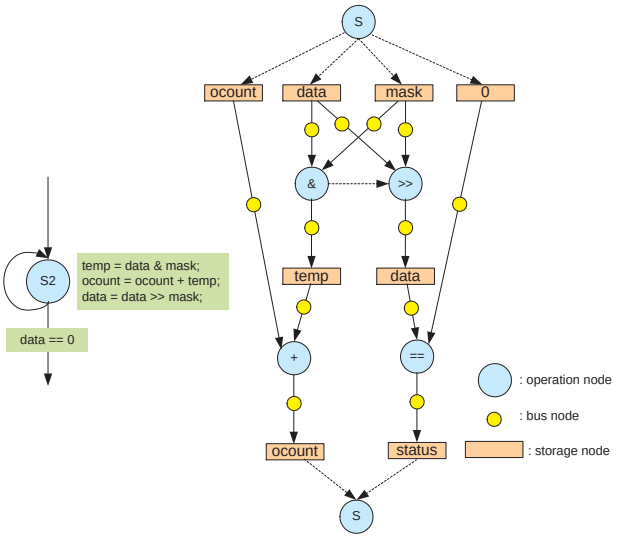

Figure 16. CDFG for state S2 in one's counter

tool and decision-making tools in the SoC design environment [APY+03]. The examples (*q_p*, *set_sign*, *buildcode*, *search codebook*) have been chosen from the GSM Vocoder which is employed worldwide for cellular phone networks. The model was based on the bit-exact reference implementation of the ETSI standard in ANSI C. The example *sra* is the SRA which was explained in the previous section.

Table 4 lists the characteristics of the designs used in terms of the number of basic blocks and the number of operation in the input description. The number of basic blocks is indicative of the control complexity of the design. Also, given in this table are the type and quantity of each resource allocated to schedule and bind this design for all the experiments. The resources indicated in this table are: *alu* performs arithmetic and logic operations, *sat* does saturated arithmetic operations and *rf* is register file with 3 read ports and 1 write port and *mem* has 1 read port and 1 write port. *Alu* executes in 1 cycle and *sat* unit is a two stage pipelined unit. The number in parenthesis indicates the size of a register file and a memory.

We present the logic synthesis result obtained after synthesizing the RTL Verilog generated by Netlist mapper 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 number of states in FSM controller, the critical path length (in nanoseconds) and the unit area (in terms of synthesis library used) through the design. The critical path length is the length of the longest combinational path in the netlist as reported by static timing analysis tool and it dictates the clock period of the design.

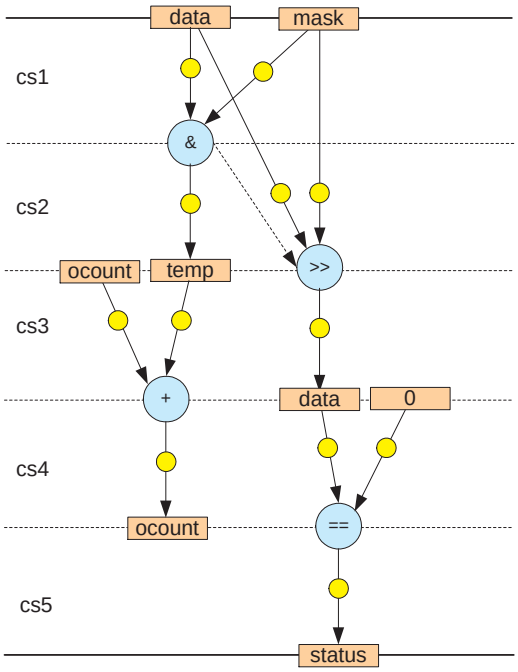

Figure 17. Scheduled CDFG for state S2 in one's counter

7. Conclusion and Future Work

In this paper, we proposed an interactive C-based RTL design environment targeting a generic RTL processor architecture. Our environment takes full advantage of the designer's insight by allowing to enter, modify, override all decisions at will. A tool has been developed and experiments were performed to validate this environment. This allows designers to evaluate several design points during fast exploration. Future work in this direction will involve the scheduling of bus protocols under timing constraint in clock cycles. We present the logic synthesis result obtained after synthesizing the RTL

References

[Acc01] Accellera C/C++ Working Group of the Architectural Language Committee. RTL Semantics, Draft Specification. Technical report, Accellera, February 2001. available at http://www.eda.org/alc-cwg/cwg-open.pdf.

[APY+03] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-chip Environ-

Table 4. Synthesis result for examples

benchmark	# BB	# OPs	# resources	# states	critical path (ns)	unit area
sra	3	8	1 min, 1 max, 1 abs, 2 shift, 1 add, 1 sub, 4 reg, 4 bus	8	21.5	8874
q_p	7	7	1 alu, 5 reg, 4 bus	12	29.2	9816
set_sign	28	38	1 alu, 2 sat, 1 rf (16), 1 mem (256), 6 bus	61	43.4	72434
buildcode	51	78	1 alu, 2 sat, 2 rf (16), 1 mem (256), 6 bus	117	48.5	87088
search	59	371	1 alu, 2 sat, 3 rf (32), 1 mem (2048), 8 bus	293	55.5	151334
codebook	180	559	2 alu, 2 sat, 3 rf (32), 1 mem (2048), 8 bus	543	58.2	188217

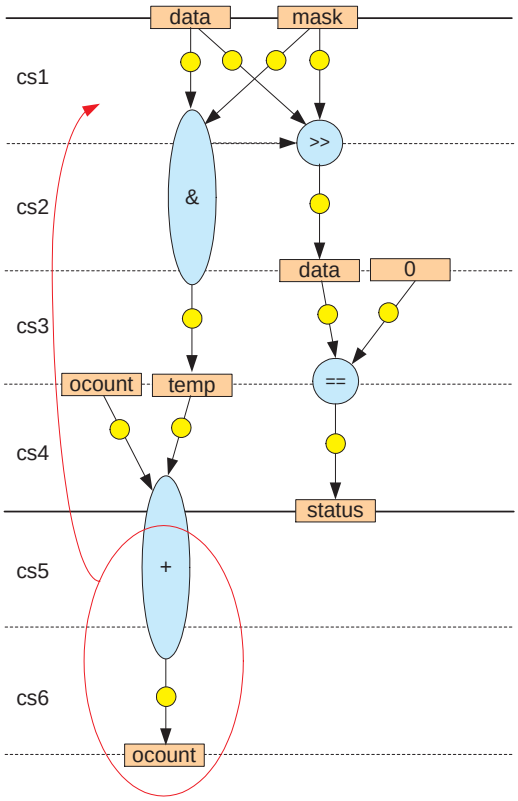

Figure 18. Scheduling process for one's counter (II)

ment (SCE Version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-18, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[Gaj97] Daniel D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[GCS+03] Andreas Gerstlauer, Lucai Cai, Dongwan Shin, Rainer Dömer, and Daniel D. Gajski. System-on-chip Component Models. Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, August 2003.

[GDLW92] Daniel D. Gajski, Nikil Dutt, Steve Y-L. Lin, and Allen Wu. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, March 2002.

[Gup03] Sumit Gupta. *Coordinated Coarse-Grain and Fine-Grain Optimizations for High-Level Synthesis*. PhD thesis, University of California, Irvine, School of Information and Computer Science, June 2003. available at http://www.cecs.uci.edu/~spark/.

[GZD+00] Daniel D. Gajski, Jiwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Suqing Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, January 2000.

[JGC96] Hsiao-Ping Juan, Daniel D. Gajski, and Viraphol Chaiyakul. Clock-driven performance optimization in interactive behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–157, November 1996.

[JPO93] Ahmed A. Jerraya, In-Cheol Park, and K. O'Brien. AMICAL: An interactive high level synthesis environment. In *Proceedings of the European Design Automation Conference*, pages 58–62, February 1993.

[Mic99] Giovanni De Micheli. Hardware synthesis from C/C++ models. In *Proceedings of the Design Automation and Test Conference in Europe*, pages 382–383, March 1999.

[SÓ1] Luc Séméria. *Applying Pointer Analysis to the Synthesis of Hardware from C*. PhD thesis, Stanford University, Electrical Engineering, June 2001. available at http://chronos.stanford.edu/users/lucs/.

[Syn] Behavioral compiler, synopsys inc. available at http://www.synopsys.com/.

[WO00] Kazutoshi Wakabayashi and Takumi Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vender perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.