ISCAS 2000 - IEEE International Symposium on Circuits and Systems, May 28-31, 2000, Geneva, Switzerland

A PARAMETRIC MODULE DESIGN FRAMEWORK AND ITS APPLICATION TO GATE-LEVEL DATAPATH/DSP MODULE SYNTHESIS

Ming-Luen Liou and Tzi-Dar Chiueh

Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan R.O.C
chiueh@cc.ee.ntu.edu.tw

ABSTRACT

The paper presents a parametric module design framework that is suitable for datapath/DSP soft-IP design. This framework is based on the integration of various frequently used parametric module generators. Under this design framework, system or circuit designers specify the structural information of the modules in C++, and then compile and co-simulate with any C/C++ programs/algorithms. Furthermore, they can manually adjust the simulation model whenever necessary. Once the system design is completed, an efficient gate-level Verilog code can soon be generated automatically. By examining the system functionality using high-level language and automatically translating the design entries into gate-level description, we can easily keep our design effort at system level while maintaining a tight consistency between different levels of abstraction. Therefore the proposed framework yields a fast, robust, and cost-effective solution to high-complexity datapath/DSP module design.

1. INTRODUCTION

Rapid advancement in VLSI technologies today makes the system on chip(SoC) design more and more feasible. Many EDA tools, such as integrated design environments and enhanced design methodologies, have been proposed to handle the chip design/verification routines with very high complexity as SoC [1]. In general, the guidelines of improvement are: integrating tools and design entries between different levels of abstraction, improving the design reusability, and reducing human-design portion in a design flow in order to shorten the design cycle. Figure 1(a) depicts a conventional IC design flow. With a target specification, designers divide the whole system into several modules. They decide the types of implementation (hardware/software), describe each module using high-level language and verify the system functionalities. When the system architecture is determined, designers manually re-write or synthesize the RTL/gate-level code, design the chip layout by tools or by hand, perform the layout parasitic extraction, and verify the timing and power specification. Since designers need to repeat design phases till the required specifications are met, these procedures which have not been automated inevitably become hurdles in the design process [2].

2. DESIGN BASED ON C++

Recently several companies have focused on bridging the design gap between C++ and RTL descriptions [2, 3]. They provide C++

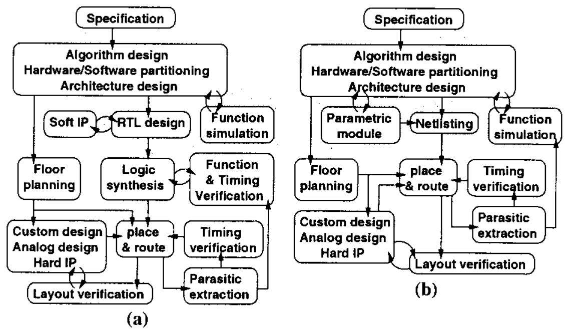


Figure 1: Chip design flows: (a) traditional, and (b) proposed.

classes, templates, and operators which complement the hardware-related features missing in C/C++. Using this extension, hardware modules can be designed, compiled, and linked with any C/C++ routines. Comparing with building a new hardware-oriented language for module design, this approach does make sense because many available programs and algorithms are written in C. On the other hand, they provide C-to-RTL translators which converts C-algorithms into RTL modules. These tools facilitate RTL code generation, thus the designer can focus on design in C programming. As shown in Figure 1(a), the design phases enclosed in the shadow block form a C design framework. Once the RTL code (the output of this design framework) is generated, the designer can synthesize the gate-level code using various commercialized logic synthesis tools/technology mappers.

The viewpoint mentioned above is still based on the assumption that universal logic synthesis tools are available. However, general-purpose synthesis tools are practically more suitable for random-logic synthesis. Datapath synthesis, which exhibits much more regularity, requires specific module generators designed case-by-case (according to their functionalities, target technologies, and other architectural concern [4, 5, 6, 7]). Consequently, we focus on integrating the various parametric module generators into a C++ design framework. That is, each module generator is declared as a new class which encapsulates functional simulation/verification, parameter estimation, and netlisting routines. Most routines are inherited from a base class and need not be designed again. A parametric module can be easily assembled by describing the structural information (ownership, connectivity) of the module in C++, according to its parameters. Under this circumstance, **a new design**

0-7803-5482-6/99/$10.00 ©2000 IEEE

is naturally a parametric module, and can be added in to a parametric module library for design-reuse. After the structural information are properly specified, we can perform function simulation, parameter estimation, or netlist generation. The proposed design flow is shown in Figure 1(b). Note that in the new design flow, the RTL level is absent.

Table 1: The parametric module library structure.

Top level, high complexity

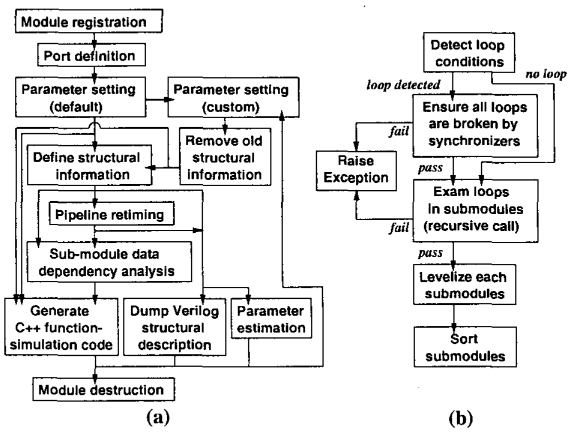
Catagory		Parametric modules	Architecture
Macro blocks (frequently used datapath / DSP modules)		Complex multiplier / MAC Correlator Digital interpolator FIR/IIR filter NCO/CORDIC ALU	
Datapath components (basic datapath building blocks)	Arithmetic	Binary adder	CPA / CLA / CSA
		Multiple input adder (w/wo sign extension)	WTA
		Signed/unsigned multiplier Constant multiplier	Parallel (Tree/Array) Serial (Booth's Enc.)
	Misc.	Shifter	Barrel type Multiplexer type
		Counter	Synchronous Asynchronous
		FIFO/Delay line	
Basic modules (array-type standatd cells)	Simple Gate	NAND/NOR/AND/OR OAI/AOI XOR/XNOR Multiplexer/Demultiplexer	Static CMOS Transmission Gate SFPL (high fan-in gate)
	Arithmetic Cell	Half Adder/Full Adder 4-2 Compressor Carry/Carry-lookahead	Transmission Gate CPL Hybrid
	Buffer	INV/BUF (w/wo enable) Line driver	Static CMOS
	Flip-flop Memory	PET/NET/DET DFF SRAM/ROM/PLA	Transmission Gate

Buttom level, low complexity

3. PARAMETRIC MODULES AND HARDWARE RELATED DATA CLASSES

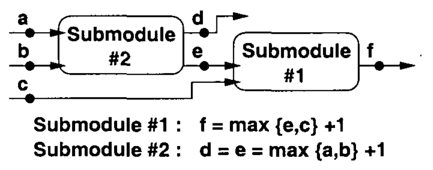
Table 1 lists the proposed C++ parametric datapath module library. The basic modules, which are just arrays of standard cells/primitives, lie in the bottom level of the hierarchy. The datapath components, which are composed by basic modules and other datapath components, lie in the middle level of the hierarchy. The macro blocks that are frequently used in large datapath or DSP core design belong to the top level of the hierarchy. All modules inherit the base module class, *BaseModule*, which encapsulates the netlisting functions (for the Verilog structural description/C++ function simulation code generation), parameter estimating functions(for area, latency, and power estimation), and several maintenance routines. Most of them are executed in a hierarchical manner, i.e., they have a similar behavior that recursively call the homonymous functions in their submodules.

Figure 2(a) depicts the work phases of a module. Each work phase corresponds to a set of member functions defined in *BaseModule*. In module registration and port defining phases, the system performs memory allocation, port mapping and object registration routines. In structural information defining phase, the module allocates its submodules, internal nets, and specifies submodule connectivity according to I/O port widths and parameters given in parameter setting phase (The I/O port widths are implicitly defined in net data structure). Then, the submodules can be re-timed or sorted according to the data dependency. The functions in the submodule data-dependency analysis block are depicted in Figure 2(b). Basically, they perform a well-known levelization process that computes the "level" of each module, and sort the modules by their levels. An example that levelizes two modules and their



Module registration
Port definition
Parameter setting (default)
Parameter setting (custom)
Define structural information
Remove old structural information
Pipeline retiming
Sub-module data dependency analysis
Generate C++ function-simulation code
Dump Verilog structural description
Parameter estimation
Module destruction
(a)
Detect loop conditions
loop detected
no loop
Ensure all loops are broken by synchronizers
fail
Raise Exception
pass
Exam loops in submodules (recursive call)
fail
pass
Levelize each submodules
Sort submodules
(b)

Figure 2: (a) The generic life-cycle of a module, and (b) the submodule data-dependency analysis block.



a
b
c
Submodule #2
d
e
Submodule #1
f
Submodule #1 : f = max {e,c} +1
Submodule #2 : d = e = max {a,b} +1

Figure 3: The levelization procedure.

output nets is shown in Figure 3. Because the data dependency is unknown, the process should be repeated until the level of each node stop changing. Therefore, the running time of the levelization process is a quadratic function of the module size, which is acceptable because this process is executed only once.

The hardware-related data classes are used to describe the data exchange or connection between modules. In Verilog gate-level representation, we declare *wire* variables to connect modules. This type of variable is not capable of storing values. (The storage elements are implicitly declared in primitives/UDP). However, to easily pass parameters/values between C++ modules, we declare variables that can store values in each modules. To support various data types and operations existing in HDL, we introduce several C++ classes with the following features:

- Provide variable and constant fixed-point types with nearly unlimited precision (number of bits). Each type is automatically truncated or sign-extended if necessary.
- Each type possesses basic functions/operators for arithmetic, compare, logical, assignment, and type converting operations.
- Support bit-level addressing and arbitrary data/wire concatenations.

4. FUNCTIONS FOR SIMULATION, PARAMETER ESTIMATION, AND NETLISTING

In our C++ simulation methodology, the timing model is chosen to be cycle-based. Function simulation can be performed by repeatedly changing the main module inputs, and calling the simulation routine to evaluate the module outputs. The C++ code is generated by the module according to either the structural information or user-specified functions. The default module simulation routine evaluates submodule outputs (invokes simulation routines of submodules) according to submodule data dependency analyzed in the submodule data-dependency analysis phase (Figure 2).

The C++ language shows a remarkable advantage over C or HDL because the designers can overload the simulation routine to verify the module functionality or to speed up functional simulations. For example, the default simulation routine of the main module (e.g. tree multiplier) calls its submodule simulation routines, namely, the simulations routines of the AND-plane / the Booth's encoders, the Wallace tree adder, and the high-speed binary adder. In addition, the behavior of each module can be modeled by a pure function followed by a delay element with variable number of cycles. Hence programmers can also design a simulation routine with a binary multiplication and time delays. The latter simulation routine is more abstract and irrelevant to the detail structure of the tree multiplier, and is suitable when only functional correctness is of concern. Module designers can compare the outputs of the two routines to uncover design flaws in the module structural information defining phase. Moreover, to speed up simulations, designers can overload the default simulation routine using the simple routine once the module has been verified. Furthermore, if the default simulation routine has been overloaded, we can just perform simulation without specifying the module structural information (refer to Figure 2(a)). This does agree with the concept of high-level simulation.

Since parametric module can support many parameter configurations, designers need a fast way to choose a proper configuration to implement. The parameter estimating functions encapsulated in a module are designed to evaluate several characteristics/parameters (e.g. gate-count, area, delay or power consumption) for design-space explorations from a system perspective. By calling these functions after defining structural information, designers can acquire the knowledge about module characteristics, and try another parameter configuration if necessary. The default parameter estimating functions provided by the *BaseModule* class are briefly described below:

- The gate-count estimating function returns the sum of submodule gate-count.
- The area estimating function returns the sum of submodule area plus some area for routing.
- The power estimating function estimates the data activity of each net declared in this module, and then calculates the dynamic power according to the data activity and the capacitance (fan-in, fan-out) information stored in each net. The function returns the power dissipation of these nets, plus the power dissipation of submodules.
- The propagation delay estimation is done using a static timing analysis algorithm similar to the levelization process mentioned above.

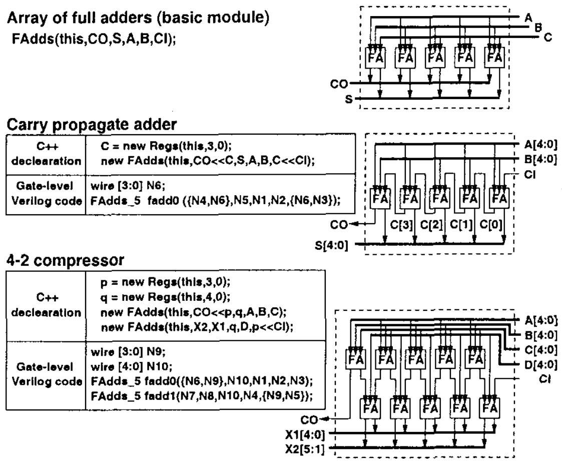
The above parameter estimating functions are designed to quickly estimate the module characteristic and hence, are very simple. Module designers can overload the functions with more accurate ones.

The netlisting function dumps the Verilog description of the module. In this function, the system traverses all the submodules, and dumps the nets and submodule connection recorded in the module data structure. Since C++ code should be compiled before execution, a naming mechanism is required. Here we simply assign an object name using an alphabetical prefix concatenated with the object serial number. The parametric module names, on the other hand, are assigned according to the parameter values.

5. MODULE DESIGN EXAMPLES

To design a parametric module, we just declare a new class that inherit the *BaseModule*, and overload the module *organization* function (define structural informations). This function allocates submodules/internal nets (*Regs* objects) and specifies the module connection using the internal nets or module ports. Moreover, the structured statements in C/C++ (*for, while, repeat, if*) facilitate the parametric description for allocation and connection of module instances.

When a module has been designed, users can allocate an module instance and execute its organization and data dependency analyzing functions. Next, the C++ code of the module can be generated automatically such that users can perform simulations. For example, users can assign a nested for-loop to verify the module functionality for all input conditions. On the other hand, users can call the parameter estimating functions to roughly check the module characteristics. Once the input parameters and port widths are determined, users can execute the module netlisting function to generate the gate-level Verilog code. Figure 4 demonstrates two simple design examples. The C++ statements in module organization functions and the corresponding Verilog netlist are listed together for comparison. Obviously, there is a strong analogy between the two representations.



Array of full adders (basic module)

```
FAdds(this,CO,S,A,B,CI);
```

Carry propagate adder

C++ declearation	`C = new Regs(this,3,0);` `new FAdds(this,CO<<C,S,A,B,C<<CI);`
Gate-level Verilog code	`wire [3:0] N6;` `FAdds_5 fadd0 ({N4,N6},N5,N1,N2,{N6,N3});`

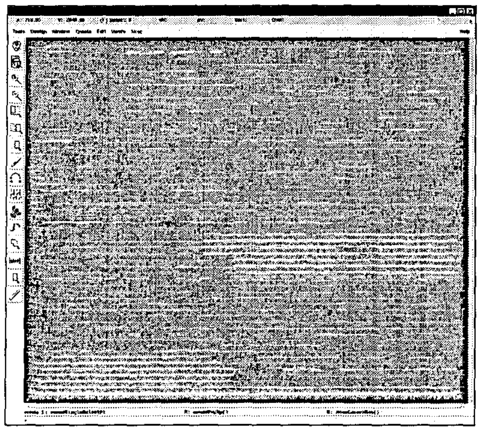
4-2 compressor

C++ declearation	`p = new Regs(this,3,0);` `q = new Regs(this,4,0);` `new FAdds(this,CO<<p,q,A,B,C);` `new FAdds(this,X2,X1,q,D,p<<CI);`
Gate-level Verilog code	`wire [3:0] N9;` `wire [4:0] N10;` `FAdds_5 fadd0({N6,N9},N10,N1,N2,N3);` `FAdds_5 fadd1(N7,N8,N10,N4,{N9,N5});`

Figure 4: Modules constructed by basic modules.

In the C++ statements, the left-shift operators "≪" and "≪=" are overloaded for wire concatenation. For the carry propagate adder shown in Figure 4, we concatenate the current module I/O ports *CO, CI* with the net *C*, and pass them to the submodule(*FAdds*). Similarly, in a usual statement likes "a[3]=b[4]+1;", the bucket operator "[ ]" is overloaded for bit addressing operations. These operators utilize several features of C++ (Reference, Automatic type conversion, Copy constructor, etc.) for passing structural information and are transparent to module designers.

In the next example shown in Figure 5, we allocate a module instance of a tree-type parallel multiplier that is connected with two 54-bit input ports. We call the organization and pipeline functions to generate a pipelined 54x54 tree multiplier. Then, we call the netlisting function to generate the Verilog structural description and C++ function simulation code. The multiplier gate count reported by the system is 30273, which is approximately 100K transistors. The Verilog description is then placed and routed. TSMC 0.6μm SPTM technology is selected and the layout size is 2802μm x 2311μm. Consequently, an module layout is soon delivered. Since the designer does not need to manually design the gate-level code, the design gap between the C++ module design and layout design (place and route) is now bridged and a design can be completed smoothly and efficiently.


(a)

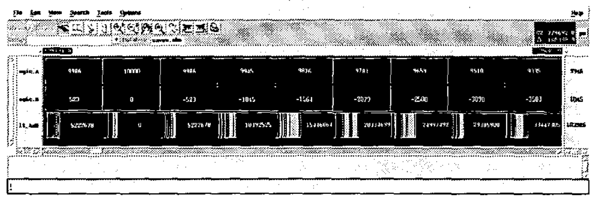

(b)

Figure 5: Mapping designs onto silicon : (a) The layout of a 54x54-bit tree multiplier, and (b) its Verilog gate-level simulation result.

6. CONCLUSION

In this paper we demonstrate a parametric datapath module library design based on the proposed C++ module design framework that encapsulates simulation, parameter estimation, and netlisting routines into one C++ class. Under this framework, we can design hardware modules using existing ones and perform simulations with other algorithms/programs written in C/C++. Once the system design is completed, efficient gate-level Verilog code can soon be delivered. This approach drives a hardware design directly from C++, and keeps a tight consistency between different levels of abstraction. A feature summary and comparison between different system design methodologies are listed in Table 2. The proposed design framework listed at the right-most column seems simpler and superior to the other methodologies in datapath/DSP design. We believe this design framework is a fast, robust, and cost-effective solution to high-complexity datapath/DSP module design.

Table 2: A comparison for various design methodologies

	Traditional (manually rewrite RTL code)	Design framwork proposed by [2,3]	This work
Design level of abstraction	low	high	high
Design time	long	short	short
Required tools for function simulation and netlisting	C++ compiler logic synthesizer	C++ compiler C-to-RTL translator logic synthesizer	C++ compiler
Perform simulation using compiled code	depend on tools	yes	yes
Module description	describe either the behavior or the *structural information* of a module.	*describe the behavior* of a module.	*describe the structural* information of a module. *The behavior description* is optional.
Used C++ terminologies	—	class, template, macro and preprocessor	class
Functions encapsulated in C++ modules	—	simulation	Dump C++/Verilog code parameter estimation simulation
Simulation model	event-driven	cycle-based	cycle-based
Simulation speed	slow	fast	fast
Output	—	RTL Verilog/VHDL	gate-level Verilog C++ code for function simulation
Target technology	depend on logic synthesis tools	depend on logic synthesis tools	depend on each modules generator
Usage	depend on logic synthesis tools	depend on logic synthesis tools	*datapath / DSP core* design
Design reuseability	low	high	high

7. REFERENCES

[1] L. Geppert, "Design tools for analog and digital ICs," *IEEE Spectrum,* Apr. 1999, pp. 41-48.

[2] J. Sanguinetti, "Bridging the design gap with Cynthesis," CynApps Company web page: http://www.cynapps.com/, 1999.

[3] Frontier Design Press, "C spans floating to fixed-point gap," Fron-
tier Design Company web page: http://www.frontierd.com/, Jun. 1999.

[4] H.M.A.M. Arts, J.T.J. van Eijndhoven, and L. Stok, "Flexible block-multiplier generation". *International Conference on Computer-Aided Design. Digest of Technical Papers.* Santa Clara, CA, Nov. 1991, pp. 106-109.

[5] J. Hsu and O. Bair, "A compiler for optimal adder design," *Proceedings of the IEEE Custom Integrated Circuits Conference,* Boston, May 1992, pp. 25.6.1-25.6.4.

[6] J. Nurmi, "Portability Methods in Parametrized DSP Module Generators," *Proceedings of IEEE Workshop on VLSI Signal Processing,* New York, Oct. 1993, pp. 260-268.

[7] J. Pal Singh, A. Kumar, and S. Kumar, "A multiplier generator for Xilinx FPGAs," *Proceedings of the 9th International Conference on VLSI Design,* Bangalore, India, Jan. 1996, pp. 322-323.