A Macro Generator for Arithmetic Cores

D. Bakalis[1,2], M. Bellos[1], H. T. Vergos[1,2], D. Nikolos[1,2] & G. Alexiou[1,2]

[1]Computer Engineering and Informatics Dept., University of Patras, 26 500, Rio, Greece
[2]Computer Technology Institute, 3, Kolokotroni Str., 26 261 Patras, Greece

e-mail : {bakalis, vergos, nikolosd, alexiou}@cti.gr, bellos@ceid.upatras.gr

Abstract

In this paper we present a tool for macro generation of soft cores performing arithmetic operations for a wide variety of operand sizes and architectures. The tool produces structural Verilog descriptions. Hence, any commercial synthesis tool can be used to map the produced designs to a specific technology. The generator covers all four basic operations: addition, subtraction, multiplication and division. Therefore, applications requiring arithmetic cores, as for example digital signal processing and multimedia applications, can be completed faster and with less effort.

1. Introduction

System-on-a-Chip (SoC) design challenges engineers with a large scale task to integrate and provide a rich set of features, often with high performance, while achieving time-to-market goals on progressively shorter product life cycles. Design automation has increased productivity and reduced cycle time, while a reuse strategy has become a means of amortizing the development effort and investment across multiple projects [1, 2].

The current trend in hardware design is code reusability and core-based design. The use of *pre-verified, pre-tested macros* leads to reduced time-to-market through faster design cycles and higher productivity of the design teams. Moreover this development approach minimizes risk and enables cost-effective solutions. In contrast with hard cores, that are firm design blocks already targeted to a specific implementation technology, cores that are provided as a Hardware Description Language (HDL) description at the RTL level provide the designer with a greater freedom. They can be mapped to any ASIC or FGPA implementation technology by any logic synthesis tool that the designer is acquainted to. Note that the substitution of the core with actual RTL synthesizable code is done during synthesis. For simulation purposes, the core author also provides the user with a behavioral model that cuts down the simulation time significantly.

Arithmetic modules are common to almost every system design. Coding of algorithms for arithmetic operations, or of building blocks for such algorithms, in a technology independent HDL and proved capability of synthesizing such a description in a variety of industrial products, offer a system designer with efficient and versatile building blocks for developing complex System-on-a-Chip solutions. Most available synthesis tools offer some kind of block generators for arithmetic modules. Common limitation is the lack of a variety of architectures for all possible implementation technologies.

Several arithmetic module generators have been presented in the last years. In [3] a module generator capable of generating complex arithmetic structures is presented. Rather than using distinct adder or multiplier modules, this generator combines the arithmetic expressions that supports into a single module. The disadvantage of this approach is that restricts the designer from examining different alternatives to meet his design goals. [4] presents a macro-cell generator for floating-point division and square root operation. A module generator for logic-emulation applications, which is able to generate macro cells of arbitrarily complex functions described in HDL is proposed in [5]. This module generator accepts Verilog description and produces partitioned CLB netlists for decomposing the design into multiple FPGA chips.

In this work we present a tool for macro generation of cores performing arithmetic operations. The generator produces a structural Verilog description of an arithmetic module given the type of the requested operation and the size of the operands. Since a completely structural description is produced, reusability is assured. Our generator covers all four basic operations: addition, subtraction, multiplication and division. For each type of operation it is capable of producing more than one cores using different algorithms for their implementation.

Our generator produces each description within a very small amount of time enabling the designer to synthesize two or more distinct architectures to a target technology and then select the one that best suits his needs. If the same technology and core is used several times, only one synthesis run is required, if the results are stored in a core database.

Therefore, the generator can both alleviate the design effort and help towards the reduction of time to market.

The remaining of the paper is organized as follows: In Section 2 we discuss about the requirements that such a generator must fulfill. In Section 3 we give details about the implementation of the macro generator for arithmetic cores. Experimental results obtained by our generator are presented in Section 4. Section 5 describes how the design of a floating-point multiplier unit was simplified by the use of our generator.

2. Requirements and Specifications

In order to be generic and easy-to-use, a tool for macro generation of arithmetic cores has to confront to the most of the following requirements:

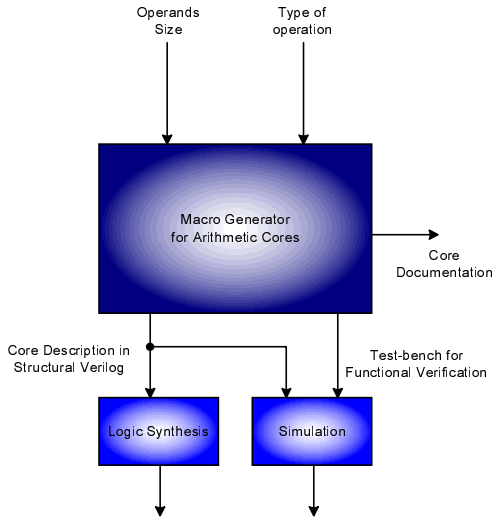

Fig. 1. A block diagram of the macro generator for arithmetic cores

- *Technology Independence*: It should not restrict to specific gates or technology libraries. Rather it must provide the core description in a more abstract level (eg. in HDL).
- *Easy Integration*: the designer must be able to integrate, without difficulty, the modules generated by the tool with the rest of his design.
- *Many Alternatives*: the tool must be able to produce cores for all four basic operations (addition, subtraction, multiplication and division). For each type of operation it has to give the designer the opportunity to select, between several different architectures performing a specific operation, the one that best suits his needs in terms of circuit area, delay, power and testability. Moreover all these alternatives should be available independently of the targeted technology.
- *No restrictions on the operand size*: The arithmetic macro generator must have the smallest possible limitations about the operands size of the cores that can produce.
- *Applicability*: The generator must be able to produce, at least, the most frequently used arithmetic modules.
- *Easy-to-Use*: The tool has to provide the designer with a simple and easy-to-use interface.
- *Expandability*: It has to be implemented with an open interface that ensures that the addition of new capabilities or architectures for arithmetic operations is straightforward.

Having all these requirements in mind, we have made the following decisions regarding the implementation of the arithmetic macro generator.

In order for the core to be technology independent, the macro generator produces its description in structural Verilog language. Note that this gives the ability to the designer to easily integrate a module with the rest of a design and map it to a specific technology library through the use of any commercially available logic synthesis tool. Moreover it does not disturb the requirement for a behavioral simulation model. The requested core, that the tool generates, will be described as a interconnection of basic cells (eg. full adders, half adders, controlled adder/subtracters). For this reason the designer has the ability to define his own gate-level implementations of the basic cells even if he has no knowledge of the algorithm implementing the arithmetic operation.

Furthermore, the tool is able to produce several different modules for each of the four basic arithmetic operations (addition, subtraction, multiplication and division). For example the tool can implement both ripple-carry and carry look-ahead adders, both Baugh-Wooley and Booth-recoded multipliers etc. This feature provides the designer with a volume of design alternatives for a specific type of arithmetic operation and the ability to choose the one that best meets his needs regarding area, delay, power and testability.

Finally, we decided to implement the macro generator as a software tool with both a command-line and a simple menu-driven user interface so as to help the designer interact with it and make upgrades and improvements an easy task.

The block diagram of the macro generator is given in Figure 1. Once the designer defines the type of the operation and the size of the operands, the tool produces the requested core described in structural Verilog and the core documentation. Then the core can either be combined with the rest of a design and mapped to a specific technology with a logic synthesis tool or can be simulated for functional verification using a test-bench module that is produced by the generator together with the core.

3. The Macro Generator for Arithmetic Cores

This section consists of two subsections. In the first one we present the types of cores that the macro generator can produce and the algorithms that are used for this purpose. In the second one we discuss several implementation issues.

3.1 Implemented Algorithms for arithmetic operations

We now present the types of cores that the generator can produce. Subtraction is usually combined with addition and therefore we consider these two types of arithmetic cores as one.

In the case of adder/subtracter cores the generator can produce:

(a) adders/subtracters based on *Ripple-Carry* propagation (RCAS) composed of full adders,

(b) *group Carry Look-Ahead adders/subtracters* (GCLAS) composed of 4-bit Carry Look-Ahead units and ripple carry propagation between groups,

(c) *Carry Look-Ahead adders* (MLCLA) with multiple levels of Carry Look-Ahead generation logic [6],

(d) *Sklansky parallel-prefix* adders (SKA) [7], and

(e) *Kogge-Stone parallel-prefix* adders (KSA) [8].

In the case of parallel multiplier cores we distinguish the case of unsigned operands and the case of operands in 2's complement arithmetic. In the former case, the macro generator can produce:

(a) *Carry-Propagate Array Multipliers* (CPAM) composed of a series of ripple-carry adders for adding the partial products and

(b) *Carry Save Array Multipliers* (CSAM) [6] that use carry-save adders to sum the partial product bits and a ripple-carry adder to produce the final product.

In the case of 2's complement multipliers, the macro generator can produce cores implementing:

(a) the *TriSection Pezaris* algorithm (TPAM) [9] consisting of 3 different "adder" basic cells,

(b) the *Baugh-Wooley* algorithm (BWAM) [10], and

(c) multiplier cores based on 2-bit *Booth* recoding (MBM) [11]. In this case, carry-save adders are used to sum the partial products and an adder is used to produce the final product.

Furthermore, in the case of serial-parallel multipliers the macro generator can produce serial-parallel multiplier cores [12] for 2's complement numbers.

In the case of dividers, the generator can produce:

(a) *Non-Restoring Cellular Array Dividers* (NRCAD) composed of 1-bit controlled adders/subtracters and

(b) *Restoring Cellular Array Dividers* (RCAD) composed of 1-bit controlled subtracters [6].

3.2 Macro Generator Implementation Issues

The tool for macro generation of arithmetic cores has been implemented using ANSI C programming language. This language was chosen because it is widely used and is supported by many different operating systems.

The implementation of the above algorithms for arithmetic operations imposes some restrictions on the size of the operands that can be handled. For example, since we use 2-bit recoding for Booth multipliers, the operand size must be a multiple of 2. Furthermore, in the current version of the tool, we support only modules with equal sized operands (in the case of dividers the size of the one operand is twice the size of the other operand).

Table 1. Modules and operand sizes

	Minimum size	**Step**	**Maximum Size**
Adder/Subtracter Modules			
Ripple Carry Adder/Subtracter	2	1	1024
Group Carry Look-Ahead Adder/Subtracter	4	4	1024
Multi-Level Carry Look-Ahead Adder	4	4	256
Sklansky Parallel-Prefix Adder	2	1	1024
Kogge-Stone Parallel-Prefix Adder	2	1	1024
Multiplier Modules			
Carry Propagate Array Multiplier	2	1	256
Carry Save Array Multiplier	2	1	256
TriSection Pezaris Array Multiplier	2	1	256
Baugh-Wooley Array Multiplier	2	1	256
Modified Booth Multiplier	2	2	256
Serial-Parallel Multiplier	2	1	256
Divider Modules			
Restoring Cellular Array Dividers	8	power of 2	64
Non-Restoring Cellular Array Dividers	8	power of 2	64

The supported modules and operand sizes are shown in Table 1. The second column indicates the smallest operand size supported while the fourth column indicates the largest operand size supported. The third column indicates the step between two supported operand sizes. For example, in the case of multi-level carry look-ahead adders, the tool can produce all cores with operand size a multiple of 4, e.g. 4, 8, 12, 16, ... , 256.

The output of the macro generator is the structural Verilog description of the requested arithmetic core. This description consists of 4 distinct parts:

- a *core information* part. This part provides information about the type of the arithmetic module as well as its primary inputs and outputs,
- the *basic cell definition* part which consists of the description of the basic cells that are required to create the requested core, e.g. half adders, full adders, 1-bit controlled adders/subtracters, etc,
- the *core description* part which is the structural description of the interconnections of the basic cells to create the requested core, and
- a *test-bench* for functional verification. This module can provide input vectors to the arithmetic core giving the designer the ability to check its correct functionality by a logic simulation tool.

The macro generator is designed as a software tool with both a command-line and a menu-driven user interface. Figure 2 shows the menu-driven user interface.

4. Experimental Results

The macro generator that was presented in the previous section can be used in many design applications. Since it provides a library with general-purpose arithmetic cores, it leads to rapid system prototyping. The descriptions of the modules in structural Verilog offer the ability to the designer to quickly and easily integrate these modules with his design and proceed to logic synthesis. He also has the ability to decide, among several alternatives, the one that is suitable for the specific design that he is interested in, regarding area, delay, power and testability.

We have used the macro generator to create several arithmetic cores. The test-bench that accompanies each core was used to verify the correct operation of it. For example, in Figure 3 we show some results of simulating a 14-bit Baugh-Wooley multiplier.

Furthermore, because of the structural description, the cores that the macro generator produces can be synthesized with any commercial logic synthesis tool. We have performed logic synthesis on several arithmetic cores produced by the macro generator. We have used two commercial logic synthesis tools: (a) Leonardo Spectrum by Mentor Graphics with its sample ASIC XCL05U library and (b) Design Analyzer by Synopsys driven by the AMS CUB library. Table 2 presents the results of the logic synthesis in terms of area and delay. Note that the two logic synthesis tools produce their area estimation results using two different metrics, namely square mils and equivalent gates. This is because the Synopsys tools also include a routing area estimate in their results.

The time the generator needs to produce the requested cores is very small. The descriptions of the arithmetic cores presented in Table 2 were produced in less than one second. The time needed to synthesize a core depends on the hardware platform and the specific logic synthesis tool that is used as well as the size of the core. The logic synthesis

```
(N) -> Define Circuit Name
(X) -> Define Circuit Size

Adders/Subtracters
 (R) -> Create Ripple Carry Adder/Subtracter
 (G) -> Create Group Carry Look-Ahead Adder/Subtracter
 (C) -> Create Multi-Level Carry Look-Ahead Adder
 (Y) -> Create Sklansky Parallel Prefix Adder
 (K) -> Create Kogge-Stone Parallel Prefix Adder
Multipliers
 (P) -> Create Carry Propagate Array Multiplier
 (S) -> Create Carry Save Array Multiplier
 (M) -> Create Trisection Pezaris Array Multiplier
 (U) -> Create Baugh-Wooley Array Multiplier
 (B) -> Create Modified Booth Multiplier
 (L) -> Create Serial-Parallel Multiplier
Dividers
 (D) -> Create Non-Restoring Cellular Array Divider
 (A) -> Create Restoring Cellular Array Divider

(E) -> Exit

Choice :
```

Fig. 2. The menu-driven user interface of the generator

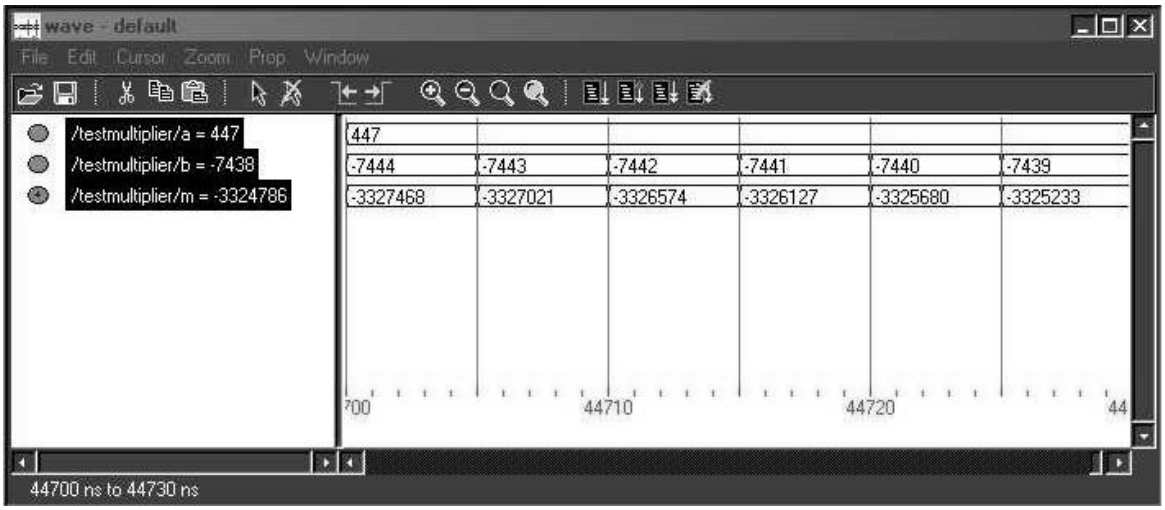

Fig. 3. Simulating a 14-bit Baugh-Wooley multiplier core using ModelSim by Mentor Graphics

process, using Leonardo Spectrum, of the largest core of Table 2 (14-bit Baugh-Wooley multiplier) completed in less than 35 seconds on a system with one Intel Pentium-III 500MHz processor, 128MBs of main memory and 100MHz system bus speed. The time needed to synthesize the other cores of Table 2 was less than 30 seconds. Note that synthesis needs to be performed only once for a specific core and implementation technology. Synthesis results can be stored in a database, allowing then the designer to have immediate access to all possible forms of a given arithmetic core in order to make the choice that best suits his needs very early in the design cycle.

5. An application example: A floating-point multiplier

The described generator for arithmetic cores was used for the design of a floating-point, double precision, IEEE standard, multiplier [13]. The multiplier's functionality was described in behavioral HDL and simulated against a software model.

Several design alternatives had to be examined to reach an implementation that balances area and performance requirements. The floating-point multiplier design makes use of three arithmetic cores:

- an 53-bit unsigned multiplier for the mantissas.
- two 11-bit integer adders/subtracters for the exponents. The first one is used to subtract the bias from the first exponent whereas the second adds the second exponent.
- an 53-bit incrementer for the rounding procedure.

The maximum delay of the floating-point multiplier depends on the delay of the unsigned multiplier for the mantissas. To this end, several implementations of this fixed-point multiplier have been taken into account. Table 3 presents attained performance and area results when alternative architectures, provided by our generator, were used for the mantissas multiplication. We have used Group Carry Look-Ahead cores for the 11-bit adder/subtracters and the 53-bit incrementer. The Leonardo Spectrum synthesis tool with the sample ASIC XCL05U library was used to obtain the results.

Due to the availability of the presented macro generator, the design and evaluation of alternative designs for floating-point multiplication was taken place in minimum time and design effort.

Table 2. Synthesized arithmetic cores with Synopsys and Mentor Graphics tools

	Synopsys Tools (AMS CUB)		**Mentor Graphics Tools (sample XCL05U)**	
	Area (sq. mils)	*Delay (ns)*	*Area (eq. gates)*	*Delay (ns)*
32-bit Ripple Carry Adder/Subtracter	295	16,3	857	16,4
32-bit Multi-Level Carry Look-Ahead Adder	272	15,1	619	14,2
14-bit Carry Propagate Multiplier	1574	34,5	4590	33,3
14-bit Carry Save Multiplier	1579	24,3	4621	22,6
14-bit Baugh-Wooley Multiplier	1531	26,1	4632	29,0
14-bit Booth-recoded Multiplier	1341	19,1	4303	16,9
14-bit TriSection Pezaris Multiplier	1653	28,1	4432	29,9
16/8-bit Restoring Array Divider	640	78,7	1461	42,6
16/8-bit Non-Restoring Array Divider	556	50,3	1159	39,6

Table 3. Floating Point Multiplier's Area and Delay results for 3 different unsigned multiplier architectures

Unsigned Multiplier Architecture	Floating Point Multiplier	
	Area (eq . gates)	***Delay (ns)***
54-bit Baugh-Wooley	399 927	175,8
53-bit Carry Save Array	394 650	169,2
54-bit Booth Recoded	402 969	107,9

6. Conclusion

We have presented a tool for macro generation of cores performing arithmetic operations. Given the type of the arithmetic operation and the size of the operands, the tool produces the requested module in structural Verilog. The generator covers all four basic operations: addition, subtraction, multiplication and division and is capable of producing more than one different architectures for any type of operation. The use of the tool gives the ability to the designer to select, in the smallest possible time, between alternative designs the one that best meets his design goals. The use of the presented generator minimizes the design effort and reduces time-to-market.

We are currently extending the generator for producing more complex cores (for example, digital filters) as well as for producing its descriptions in VHDL along with the currently supported Verilog language.

References

[1] M. Keating & P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*, Kluwer Academic Publishers, 1998.

[2] G. Dare, D. Linzmeier & B. Deitrich, "Circuit Generation for Creating Architecture-Based Virtual Components", Proc. of Design Automation and Test in Europe (User Forum), pp. 79-83, 2000.

[3] D. Kumar & B. Erickson, "ASOP: Arithmetic Sum-of-Products Generator", Proc. of International Conference on Computer Design, pp. 522-526, 1994.

[4] M. Aberbour, A. Houelle, H. Mahrez, N. Vaucher & A. Guyot, "On Portable Macrocell FPU Generators for Division and Square Root Operators Complying to the Full IEEE-754 Standard", IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 114-121, March 1998.

[5] W. Fang, A. Wu & D. Chen, "EmGen - A Module Generator for Logic Emulation Applications", IEEE Transactions on VLSI Systems, Vol. 7, No. 4, pp. 488-492, December 1999.

[6] K. Hwang, *Computer Arithmetic Principles, Architecture and Design*, John Wiley & Sons, 1979.

[7] J. Sklansky, "Conditional Sum Addition Logic", IRE Transactions on Electronic Computers, EC-9 (6) pp. 226-231, June 1960.

[8] P. Kogge & H. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Transactions on Computers, Vol. 22 (8), pp.783-791, August 1973.

[9] D. Pezaris, "A 40ns 17-bit by 17-bit Array Multiplier", IEEE Transactions on Computers, Vol. C-20, No. 4, pp. 442-447, April 1971.

[10] R. Baugh, A. Wooley, "A Two's Complement Parallel Array Multiplication Algorithm", IEEE Transactions on Computers, Vol. C-22, No. 1-2, pp.1045-1047, December 1973

[11] M. Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, 1986

[12] G. Alexiou & N. Kanopoulos, "A New Serial/Parallel Two's Complement Multiplier for VLSI Digital Signal Processing", International Journal of Circuit Theory & Applications, Vol. 20, pp. 209-214, 1992

[13] ANSI/IEEE, "IEEE standard for binary floating-point arithmetic", ANSI/IEEE Trans. Std. 754-1985.