Journal of Engineering Science and Technology Review 9 (5) (2016) 171 - 175

JOURNAL OF
Engineering Science and Technology Review

www.jestr.org

Research Article

Online Generation of Constant Mulitplication Accelerators

M. Dasygenis and I. Petrousov

Department of Informatics and Telecommunication Engineering, University of Western Macedonia, Kozani – 50100, Greece.

Received 30 June 2015; Accepted 25 January 2016

Abstract

The rising complexity of embedded digital applications and the growing importance of time-to-market require EDA tools to automate the design and implementation process of various IP blocks. One very important class of EDA tools is the generation of hardware descriptions for popular IP blocks. The multiplication by an integer constant is a special type of problem that is required in a plethora of situations. Here, we present an online tool that can generate HDL descriptions constant multiplication intellectual property blocks, using only elementary operations, like shifting and addition. Our synthesized circuits on Xilinx Virtex 6 FPGA XC6VLX760, operate up to 589 Mhz.

Keywords: Constant multiplication , EDA tool, HDL, IP

1. Introduction

More and more components are implemented as IP blocks on the same silicon, raising the number of transistors on the same die to billions. The integration of all these IP blocks is a difficult task that can intimidate even the best design teams, especially when there is lack of IP block support. Furthermore, some researchers and organizations are investigating and investing on the implementation of some algorithms in FPGA accelerators [1], a daunting task that requires many EDA tools to support and alleviate the hardware complexities.

Multiplication by an integer constant is a fundamental operation in algorithms that require some kind of matrix calculations, like Karatsuba algorithm on large integer multiplication, or the fast approximate computation of consecutive values of a polynomial. Furthermore, in case that a design space exploration is required, many circuits descriptions are needed and thus the efficiency of the team plummets. This places a lot of pressure on these teams to develop quickly with the traditional methods of edit, compile, simulate and verify. If only a tool could quickly generate parametrized, verified and accurate HDL models for such circuits, the team could reap considerable benefits not only in development time and productivity, but also in code maintainability and readability.

We noticed this shortcoming and decided to create a tool that will be able to create custom constant multiplication IP blocks, with or without pipeline to be used for custom architectures. Thus, our major contribution of our work is that we present a public web accessible tool that can create very fast syntactically correct register-transfer-level VHDL description of a constant IP block multiplications[1].

The rest of this paper is structured as follows. In the next section (Section 2) we present the importance of constant multiplication, while in Section 3 we discuss some related work. We present our algorithm in Section 4 and our tool in Section 5. The output of our tool is discussed in Section 6. Finally, we present some experimental results in Section 7.

2. Establishment of the multiplication function

Constant multiplication is a function much more complicated than addition. In the early days of microprocessors it was established that it was necessary to create a specialized circuit that will perform this task [2]. Contemporary digital circuits that perform digital signal processing (DSP), error correction codes (ECC), fast fourier transformations (FFT) all implement this function [3], [6]. Specifically, FFT processing is one of the most critical components in the orthogonal frequency division multiplexing (OFDM) [4]. OFDM itself is used in technologies including WiMax, WAN and LTE just to mention a few.

Perhaps the most important inception in improving the speed of multiplication was made by Wallace [5], who used full adders (FA) to add more than three numbers without carry propagation. This design became the base for modern multipliers. Many researches focus on improving upon this design [7], [8], [9], [10]. One of the improvements is the reduction of components used in the circuit, in this case full adders. This can achieved by reducing the number of '1' in the coefficients. Another great inception which improved the speed of the circuit was Booth's multiplication algorithm, which enabled the multiplication using shifting and allowed the use of negative numbers. Booth's technique led to many modifications of the original algorithm [11], [12] (and many more), which greatly improved the design of the multiplier in terms of speed and number of used components.

The term constant multiplication is used to denote the operation of consecutive additions of a variable x. The number of additions is determined by a constant number α. Our tool is able to produce such circuits in HDL, which can perform this operation in parallel.

* E-mail address: mdasyg@ieee.org; petrousov@gmail.com

[1] http://arch.icte.uowm.gr/hdl/constant_multiplier.php

3. Related work

The process of generation of hardware description language (HDL) code from a higher level language is not new. We selected to present only a few tools that we found to be more relevant to ours. MyHDL [13] is a framework where the multipurpose programming language python is used for the construction of structures which are translated to VHDL. The SPARK project [14] is a similar tool which accepts specifications written in C and produces system on chip (SoC) designs in VHDL. The aforementioned systems work only offline and require their installation on the local machine. This process requires root privileges and knowledge on the Linux system administration and the existence of other tools and libraries such as the gcc compiler. Furthermore, this process may differ from one operating system to another. We believe that when it comes to circuit design and space exploration, a properly configured and ready to use EDA tool able to produce syntactically correct circuit designs is a key essence.

Generators are tools which given a set of parameters are able to produce HDL code. The given parameters determine the behavior and possibilities of the produced circuit. The code must be syntactically correct and synthesizable. The flopoco project [15] is a tool which falls under the category of such generators. Specifically it is a generator of arithmetic functions. This work is not online and must be downloaded and compiled. Also, their integer multiplication units utilize binary compressors specific for the DSP blocks of an FPGA, and they are not architectural neutral, like our implementation.

In the scientific electronics libraries we have also located a few references [18], [19] which mention a tool named DiaHDL. This tool is presented to be a web-based EDA generator able to produce circuits and their testbenches in VHDL. According to [19], this tool can be run from any web browser with Java Runtime Environment and it is available to students at any place in the world. Despite our best efforts, we were unable to locate this tool and provide comparison results.

One last generator to mention is the SPIRAL tool [21] which is accessible online. This tool is able to handle only fractional numbers and does not provide a testbench to verify the multiplication results. Also our tool is able to calculate the number of components and transistors, two metrics which can be fairly useful when developing digital circuits. One more difference worth mentioning is the fact that our output descriptions are in VHDL while SPIRAL produced Verilog files.

4. The multiplication algorithm

From hardware point of view, it's always a waste of space and time to implement a generic constant multiplier [6]. Considering this we have realized our multiplier design using the simple functions of shifting and addition. The algorithm can be better understood from an example shown in Eq. (1).

$$f(x) = x \cdot 6 \qquad (1)$$

Here we want to implement the function which given a number *x* will multiply it by 6. To do this, first we need to find the binary representation of the number 6, which is 110. The number of '1's in the binary constant determines the number of coefficients which will have be added in the end. Each coefficient is a product of left shifts of the original number *x*. Here we have two '1' and thus we have two coefficients. The number of left shifts of the two coefficients is determined by the position of each '1' in the binary constant starting from the least significant bit (LSB). In our example the two '1's are located in the positions 1 and 2. Considering this, the first coefficient is a single left shift of the number x and the second is the two times left shift of the number *x*. After the shifts we add the two coefficients to get the result. The realization of the example is summed in equation (2).

$$f(x) = (x \ll 1) + (x \ll 2) \qquad (2)$$

Here we can clearly see the two coefficients which are essentially left shifts of the original number.

We present the generic algorithm used in Figure 1.

Algorithm 1 Constant multiplication
```
1: procedure ConstantMultiplication
2:     $binary\_constant \leftarrow$ binary array of $constant\_number$
3:     $result \leftarrow 0$
4:     for position of each digit in $binary\_constant$ do
5:         if $binary\_constant[position] == 1$ then
6:             result $+= x << position$
```
Fig. 1. Constant multiplication algorithm

This algorithm can be scaled to any constant and variable number because we can easily compute the number of '1's and their positions in the constant.

5. Our online EDA tool

Having faced the task of circuit design and space exploration ourselves, we have created a tool[2], which automatically generates syntactically correct VHDL code. Knowing how time consuming this task might be, we decided to share our work with the scientific community. In the scope of this paper we have designed a new function able to generate VHDL code for constant multipliers. Unlike some previously mentioned works, our tool requires no installation, is online and publicly accessible by anyone through a web browser. We utilize a number of technologies (PHP, Python, JSON) in order to deliver a syntactically correct and synthesizable VHDL description. Our tool is partitioned in two different departments, according to their function: the front end and the back end. These modules exchange information using the Javascript object notation (JSON) format [16].

The front end is a web based form, where the user inputs parameters for the circuit. These parameters include the bitwidth of the variable, the constant number that will multiply the input, the option to pipeline the circuit or not, the number of random generated vectors to be created and the option for these vector to be unique (requiring more time to be created). Validation of the inputs occurs upon submission.

The back end provides the analysis and construction modules for the multipliers. It consists of three modules: (i) the Multiplier design module, which analyzes the user inputs

[2] http://arch.icte.uowm.gr/hdl/

and creates the specific design description in a special netlist format called α-HDL [20], (ii) the HDL Generator module, which takes as input this netlist format and creates signals, networks, assignments, and connections, resulting in the output description in VHDL, and (iii) the VHDL Test bench creator, which takes as input the constructed data structures of the previous module, and generates a full VHDL test bench, with handles for automatic design validation.

A. Constant multiplier design module

The multiplier design module creates a netlist in an internal format developed at our laboratory, which we call α-HDL format, and operates in three stages: (a) locate all the '1', (b) carry save addition, (c) ripple carry addition. This module can be used to create multiplication units of unsigned vector for arbitrary bit lengths. Due to the fact that we use the Python language, there is no restriction as to the bitwidth of the input vector to be multiplied. For example the Xilinx Core generator can only create multiplication units up to 64×64 bits. Our tool has been used to create multiplication units with input vectors up to 512 bits. Such large vectors are usually found in cryptographic applications [17]. This module operates in three stages.

The first stage computes the network of shift wires. The outcomes of the first stage are two: (a) the α-HDL structure and (b) a two dimensional array that specifies for every column the bits that should be taken into consideration.

The second stage, accepts as input the array created in the previous stage and performs an optimized addition, using carry save adders. We have named this stage with the term reduction stage. This stage consists of many iterations. In every iteration i the reduction stage, scans all columns j starting from the least significant column, locates the columns that have more than one bit and places full adders (FA) or half adders (HA). The placement of adders is done in the best efficient way, in order to minimize the total number of FAs or HAs. This is achieved by delaying the placement of an FA or HA in favor of a better placement in a future iteration.

The third stage of the multiplier design module, is the final addition using a ripple carry adder. This stage, which is also optimized, places the best number and types of adders.

B. HDL generator module

The netlist created in the previous stage is given as input to the HDL Generator Module. This is a general purpose VHDL generator library that can be easily connected to many different generators. This module accepts as input a special and compact netlist format, which we name it abstracted HDL α-HDL. This netlist format, as well as the HDL Generator Module have already been presented in other works [20] and do not belong to the scope of this paper, and thus we will not describe them further.

C. The VHDL Test bench creator

We consider this module as of out most importance, as it produces testbenches which verify the generated VHDL designs. Our tool accepts as input the number of input cases to create, and generates the test bench in a VHDL file. To do this, first it creates an empty entity declaration, then it instantiates the top level component and creates signals for every input and output port. Furthermore, it creates a clock process and a function that is used to convert bits to integer. The next step is to create the requested number of input test cases.

For the number of input test cases, the module performs a loop in which a random number ranging from 0 to the maximum bitwidth is produced. This number is converted to binary and extended to the full bitwidth of the constant. Then the multiplication of the random input and the constant is precomputed and a VHDL assert clause is written on the testbench file to check the precomputed output, with the output that will be computed by the circuit. A 'wait' clause is used in order to keep the correct timing. The latency has been reported by the HDL generator, and is known in this tool.

All the test bench vectors are created randomly and automatically, according to the requested number of tests. As mentioned before, the user can also include the generation of unique testvectors. If this option is selected, the generated random numbers will all be unique and non repeating. This module also predicts the case where the requested number of testbenches supersedes the maximum random number able to be generated from the provided bitwidth. The outcome depends on the option for the unique testvectors. If it is selected, our function will generate the greatest possible number of unique tests for the given bitwidth. If not, the function will produce the requested number of tests and include repeating numbers. This process can be better understood from an example where the designer requests 10 tests to be generated for the bitwidth of 2 and constant of 2. According to equation (1) the possible results for the maximum generatable number 4 (2^2) are 0, 2, 4 and 8. If the option for the uniqueness is included, the outcome will be 4 tests (0, 2, 4, 8). Otherwise, there will be produced exactly 10 tests with repeating results (for example 4, 2, 0, 2, 4, 4 and so on). Also, all the checks are done automatically, which means that the designer can load the test bench file into his HDL Synthesis and Simulator tool, and can execute it without any other intervention.

Although our designs are syntactically correct and we perform many random test to verify the correctness of the produced circuits results. With this module we give the designer the ability to create his own custom testbenches and verify his designs.

6. Output

The output of the tool includes a library with the used components, the circuit for the multiplier and a summary report of that circuit. The first two are downloadable as VHDL files while the third is directly presented to the user. The produced VHDL descriptions are vendor neutral and can be synthesized in FPGA or ASIC circuits. The library can also be included in other designs and be repurposed accordingly. The summary presents information about the produced circuit including the number of components used, such as D flip-flops (FF), full adders (FA), half adders (HA) and more. Also, the tool is able to calculate the number of transistors used for the design.

7. Experimental results

In order to evaluate the efficiency of our web tool, we generated a large number of VHDL descriptions for different design parameters. Even though, both Xilinx and Altera provide a tool to create a parametrized multiplier for two input, the outcome is not a VHDL file and has a binary encrypted implementation netlist, which can be used only in a project targeting a specific FPGA board family. In contrast with these two vendors, our tool creates generic VHDL code that is vendor neutral and can be freely synthesized either in FPGA or in ASIC, and creates single-vector multiplications.

Another remark is, that even though there are few offline tools to create multiplication cores for only two input vectors, all these cores do not use carry save adders to compute the product, but they use special structures that make use of fast DSP blocks found on modern FPGA boards. Thus, on the one hand we cannot provide measurements with other CSA multipliers, and on the other hand, comparing our CSA multiplier scheme, with other schemes of multiplication can be used only to extract some general conclusions and not to determine the efficiency of our circuits. Some of our designs are summarized in Table 1.

Table 1. Automatically generated design results (non pipelined)

#bits	constant	transistors	FA	HA
16	46	1274	43	5
32	57	2604	90	6
64	78	5292	186	6
128	92	10682	379	5
256	121	28574	1017	7

In this table we can see some metrics that were calculated automatically by our tool and presented in the logfile. The respective pipelined versions of the designs are shown in Table 2 (DFF is the number of pipeline FF).

Table 2. Automatically generated design results (pipelined)

#bits	constant	transistors	DFF	stages
16	46	7104	453	21
32	57	22680	1608	36
64	78	85032	6516	70
128	92	314208	25037	133
256	121	1230588	99484	261

Additionally, we synthesized the generated VHDL codes with Leonardo Spectrum, Xilinx Vivado 2013.2, Altera Quartus II 12.0. The synthesis results (Table 3) from Xilinx Vivado (Virtex6, speed grade -2) show that for small input bitwidths (16 bits), the occupied slices for the pipeline version (denoted with the letter 'p') are low.

Table 3. Synthesis results for the Virtex 6 FPGA family

#bits	constant	slices	Freq(MHz)	power(W)
16	46	13	157.480	4.447
16 (p)	46	70	589.970	4.447
32	57	29	103.584	4.447
32 (p)	57	335	589.970	3.441
64	78	73	57.198	4.447
64 (p)	78	631	538.213	4.447
128 (p)	92	1387	538.213	4.473
256 (p)	121	2989	538.213	3.422

8. Conclusions

Complicated system-on-chip designs require EDA tools to perform various tasks, one of which is IP block generation. One important IP block is constant multiplication. Our tool can generate valid and verified VHDL description of parametrized constant multiplication blocks, that can operate up to 589Mhz on a Xilinx Virtex 6. As our tool is web-based, no local installation is required. Therefore, it ensures easy access to anyone in the world. There are many tools which can generate HDL code, but they are not online and some of them are commercial and expensive. Except the HDL description, our tools is able to supply a schematic and a random created testbench file to the user. The tool is public accessible from our webserver.

This paper was presented at Pan-Hellenic Conference on Electronics and Telecommunications - PACET, that took place May 8-9 2015, at Ioannina Greece.

References

1. A. Putnam, A. M. Caulfield et al., "A reconfigurable fabric for accelerating large-scale datacenter services," SIGARCH Comput. Archit. News, vol. 42, no. 3, pp. 13–24, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665678
2. A.C. Davies and Y.T. Fung. "Interfacing a hardware multiplier to a general-purpose microprocessor," Journal on Microprocessors, vol. 1, no. 7, pp. 425-432, October 1977.
3. A. H. Malini, C.Srimathi, "Low complexity digital serial FIR filter by multiple constant multiplication algorithms", IJRET 2014, vol. 3. no. 4, pp. 222-226, [Online]. Available: http://ijret.org/Volumes/V03/I04/IJRET_110304040.pdf
4. T. Chen, H. Liu, B. Zhang, "A Scalable, Fixed-Shuffling, Parallel FFT Butterfly Processing Architecture for SDR Environment," Journal Electronic Express, vol. 11, no. 2, pp. 20130905, December, 2013 [Online]. Available: http://dx.doi.org/10.1587/elex.10.20130905
5. C. S. Wallace. A Suggestion for a Fast Multiplier. IEEE Transactions on Electronic Computers, EC-13:14–17, February 1964.
6. F. de Dinechin, V. Lefèvre, "Constant Multipliers for FPGAs", International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, Las Vegas, Nevada, USA, June 2000.
7. K. Namba and H. Ito, "*Redundant Design for Wallace Multiplier",* IEICE - Transactions on Information and Systems*,* Volume E89-D Issue 9, pp. 2512-2524, September 2006.
8. S. Abed, B. J. Mohd, Z. Al-bayati and S. Alouneh, "Low power Wallace multiplier design based on wide counters", journal of circuit theory and applications, vol. 40 Issue 11, pp. 1175-1185, November 2012.
9. R. S. Waters and E. E. Swartzlander, "A Reduced Complexity Wallace Multiplier Reduction", IEEE Transactions on Computers, vol. 59, no. 8, pp. 1134-1137, 2010.
10. S. Rajaram, K. Vanithamani, "Improvement of Wallace multipliers using parallel prefix adders", ICSCCN-Signal Processing, Communication, Computing and Networking Technologies, Thuckafay, July 2011.
11. C. Efstathiou, N. Moshopoulos, N. Axelos, K. Pekmestzi, "*Efficient modulo 2n+1 multiply and multiply-add units based on modified Booth encoding*", Integration, vol. 47 no. 1, pp. 140-147, January 2014.
12. K. N. Vijeyakumar, V. Sumathy, S. Elango, "VLSI Implementation of Area-Efficient Truncated Modified Booth Multiplier for Signal Processing Applications", Arabian Journal for Science and Engineering, vol. 39, no. 11, pp 7795-7806, November 2014.
13. myhdl. [Online]. Available: http://www.myhdl.org/info.html
14. spark. [Online]. Available: http://mesl.ucsd.edu/spark/
15. F. de Dinechin. (2011) flopoco. [Online]. Available: http://flopoco.gforge.inria.fr/
16. I. E. T. Force, "Introducing JSON," September 2013. [Online]. Available: http://www.json.org/
17. L. Hars, "Long modular multiplication for cryptographic applications," in Cryptographic Hardware and Embedded Systems - CHES 2004, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds. Springer Berlin Heidelberg, 2004, vol. 3156, pp.

45–61. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_4

18. C. Widiasmoro, T. Schumann, "DiaHDL: A web-based VHDL code generator,"in Proc. of the European Conference on the Use of Modern Information and Communication Technologies (ECUMICT 2010), Gent, Belgium, ISBN 9-78908082-553-6, 2010.
19. A. R. D. Susanti, W. Thoyib, T. Schumann, "Development of a reliable GUI for DiaHDL: A web-based VHDL code generator,"in Proc. of IEEE International Conference of Electrical Engineering and Informatics (ICEEI 2011), Bandung, Indonesia, 2011.
20. M. Dasygenis, "A web EDA tool for the automatic generation of synthesizable VHDL architectures for a rapid design space exploration," in International Conference on Design and Test of Integrated Systems in Nanoscale Technology (DTIS). IEEE, 2014.
21. SPIRAL. [Online]. Available: spiral.ece.cmu.edu/mcm/gen.html