Generating custom bitwidth 4:2 compressors in hardware description language from an online tool

Ioannis Petrousov
Department of Informatics
and Telecommunication Engineering
University of Western Macedonia
Kozani, 50100, Greece
petrousov@gmail.com

Minas Dasygenis
Department of Informatics
and Telecommunication Engineering
University of Western Macedonia
Kozani, 50100, Greece
mdasyg@ieee.org

***Abstract*—The extensive implementation of sensors to measure and quantify various sizes makes digital signal processing a repetitive and widely used task. Moreover the need to keep the size of the information small has produced various redundant arithmetic number systems and compressors which use less hardware and consume less power. Especially in the real-time computing where the results must be produced as soon as possible with the less delay the implementation of such techniques becomes critical. Considering this problem, we have created a web accessible tool able to generate custom bitwidth 4 to 2 compressors in hardware description language. The produced compressors are syntesizable and use carry-save arithmetic for the input and output. The circuits have been synthesized for Xilinx Virtex 6 FPGA and operate up to 533MHZ.**

I. Introduction

Widely used devices such as smartphones, household appliances such as washing machines and medical devices, contain digital circuits which perform the task of digital signal processing (DSP). This repetitive task can become computationally intensive as the size of the input data grows, thus increasing the time needed to produce the results. This delay plays a crucial role especially in the occasion of monitoring or real time systems where actions are based on the results and must be performed as soon as possible. Moreover, the classic representation of data in binary arithmetic has proven to be inadequate, as it requires more time and more hardware to produce and store the results. Alternative numbering systems were created to tackle the aforementioned parameters of time and data size.

Redundant arithmetic systems are used extensively to speed up arithmetic operations [1] [2], [3] [4]. In fact, in [1] a computer aided design (CAD) tool has been created which recognizes patterns of operations where borrow-save arithmetic can be used. The properties of these arithmetic systems enable certain arithmetic functions, such as addition/subtraction, multiplication and more, to be performed faster and require less hardware.

Compressors are the building blocks of multipliers. Multipliers themselves are important components that dictate the overall performance of arithmetic circuits. Compressors allow the performance of addition or subtraction with minimal carry propagation. This ability constitutes a major speed enhancement technique used in modern digital circuits. A m:n compressor performs the addition of m numbers and reduces them to n, while keeping the carries and sum separate. For example, a 4:2 compressor accepts 4 numbers and reduces them to 2. This way any amount of numbers can be added together without carry propagation and only the recombination of the final carry and save part requires carry propagating addition.

In this work we have created a generator able to produce 4:2 number compressors for any bitwidth in hardware description language (HDL), specifically in VHDL. The input signals of the produced compressors are in CS representation, a redundant system which has proven be more suited for the field of DSP [5] [2] [6]. The generator is framed as a functionality to our previously developed online tool[1] [7] [8] and it's available for anyone to use[2].

The rest of this paper is organized as follows. In Section II, we present some related bibliography, in Section III, we extensively explain the carry save arithmetic and present some indicative examples. Section IV covers the architecture of our tool and the options that it offers for the creation of the compressor module which is described in Section V. Following, we present some metrics from our experimental results (Section VI) and conclude our research in Section VII.

II. Related work

Since the inception of 4:2 compressor [9] using full adders (FA), the module has been actively researched [10] [11] [12]. One of the most significant redesigns of the original module was probably the introduction of the Wallace tree [12] which reduced the critical path. Other researches focused on gate decomposition [13] in order to consume less area. More research is being done as the CMOS technology shrinks [14] [15] [16]. In [11] a novel full adder design based on the linear threshold gate (LTG) is proposed.

All the aforementioned researches are indeed improvements of the original design. Many of them use expensive and commercial programs (such as the tools from synopsys [17]) to design, verify and simulate their claims. None of the above mentions the creation of a free web based tool able to automatically generate custom IP cores. Moreover, to our best knowledge there is no online tool that can produce custom 4:2 compressors with the ability to download them as an IP core.

[1] http://arch.icte.uowm.gr/hdl/
[2] http://arch.icte.uowm.gr/hdl/compressor42.php

The process of generation of HDL code from a higher level language is not new and a plethora of tools [18], [19], [20], [21] which use different programming languages to produce circuits in HDL exists. In [18] the python programming language is used to produce hardware description language designs. The SPARK project [19] is a high level synthesis framework which uses descriptions in C to produced VHDL. [22] uses GEZEL designs to produce synthesizable VHDL code. In [21], an online tool offers a collection of generators able to produce application specific hardware descriptions such as Discrete Fourier Transform (DFT).

Not all of the above tools offer the ability to work online and produce custom circuits when needed. Those that do offer this, are limited to produce certain modules for specific purposes. The authors of [23] after examining and comparing such tools, conclude that there is a lack of EDA automation.

III. Basic carry-save arithmetic

The carry-save (CS) arithmetic is part of a group of numbering systems named redundant arithmetic numbering systems. In contrast to the binary system, in CS a decimal number requires twice as many bits for it's representation. A CS number consists of two parts, namely carry and save [5].

$$\begin{gathered} A^* = A_c + A_s \\ \text{carry } A_c \\ \text{save } A_s \end{gathered} \tag{1}$$

Considering this, a decimal number can have many CS representations, where a CS number has only one decimal representation. The possible values of CS digits are $\{0,1,2\}$, thus making it a radix-2 system. This way, when adding two numbers, the addition of each column of bits is independent and does not propagate carry to the next.

$$\begin{array}{rr} & 101110 \\ + & 110011 \\ \hline & 211121 \end{array} \tag{2}$$

Equation 2 illustrates this property where 1+1 in column 1 and 5 does not propagate a carry to the next column. The number of the radix can be increased if needed in order to maintain it's property of carry-free nature. This technique is usually used where the addition of three numbers or more is required. For example, if we want to add three numbers and at one point we need to add 3 bits 1+1+1, the result can be 3 to avoid carry propagation to the next digit.

In order to convert a CS number to binary, starting from the least significant bit (LSB), rewrite every 2 as 0 and propagate the carry to the next column.

$$\begin{array}{r} 112020 \\ \hline 112100 \\ \hline 1000100 \end{array} \tag{3}$$

Equation (3) demonstrates this conversion where carries are expanded and propagated. In the scope of this paper and to show this transformation we performed the operation serially, as we cannot predict the possible carries from each column.

IV. The online generator and compiler

After considering the importance of the hardware compressor, we decided to create a tool which can automatically generate custom IP cores of this module. 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 [24].

The front end is a web based form, where the user inputs parameters of the circuit. These parameters include the bitwidth and the type the compressor, the option to pipeline the circuit with D flip-flops, the number of random generated vectors to be created and the option for these vectors to be unique (requiring more time to be created). Validation of the inputs occurs upon submission. Optionally, our tool can generate a schematic and dot file of the design.

The input bitwidth number determines the range of the input numbers the module is able to process. This also determines the range of random numbers which are going to be produced by the testbench generator module. As seen from the architecture of the module in figure 1, the bitwidth determines the number of FAs and other components that will be used.

We offer the ability to select between two types of compressors. The first type includes only the 4:2 compressor which can handle only positive numbers and produces positive results. The second type can handle both positive and negative numbers with the usage of two's complement. This architecture is implemented with the inclusion of a 2:1 multiplexer which selects the positive or the negative number. The second type requires more components from the first one.

The two optional parameters are the schematic and dot file graph generation. The schematic is exported as a picture in PNG format and shows the components and their interconnections. The dot file is a graph written in dot language and can be used by humans or a computer program.

The back end provides the analysis and construction modules for the compressors. It consists of three modules: (i) the Compressor design module, which analyzes the user inputs and creates the specific design description in a special netlist format called a-HDL [8], (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. These modules are written in the Python programming language. The more computationally intensive functions are written in Cython which is translated to C, which is in order compiled to static object (*.so) files. For this reason the modules have no restrictions for the input bitwidth and our tool can generated compressors which can handle numbers up to 256 bits.

A. Compressor design module

The compressor design module creates a netlist in an internal format developed at our laboratory, which we call a-HDL and operates in two stages: (a) locate all the 1 and (b) carry save addition. This module operates in three stages. The first stage computes the network AND gates. The outcomes of the first stage are two: (a) the a-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.

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 abstracted HDL. This netlist format, as well as the HDL Generator Module have already been presented in other works [8] and do not belong to the scope of this paper, and thus we will not describe them further.

C. HDL Test bench Generator

This module, is of out most importance, because it creates multiple vectors of testbenches, which can be used to test the correctness of the design in an HDL simulator. As it is evident, the compressor design module is a very complicated process, which should be tested thoroughly. Our tool accepts as input the number of input cases to create, and generates the test bench file in VHDL. 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.

The algorithm here takes into account the type of the compressor and produces signals which fit each design. If the compressor can handle negative numbers, the next steps are followed. For the number of input test cases, the module performs a loop in which two random number ranging from 0 to the maximum bitwidth and a random selected signal from 0 to 1 are produced. The numbers are converted to binary and extended to the full bitwidth. Then the compression of the random inputs 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 from 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. The same process is followed for the simple compressor without the generation of the signal s.

The output consists of the VHDL codes for the multiplier, the library with the components and the testbench. These codes are vendor neutral and synthesizable in ASIC or FPGA. Furthermore, two optional outputs, the circuit schematic and dot file are available for download. The process of generation as well as the metrics for the components are presented to the user as default.

V. Architecture of the produced 4:2 compressors

As mentioned earlier, in CS, the addition of three binary numbers results to two binary numbers, this process is sometimes referred to as 3-2 compression. The same applies to the addition of more numbers, in our case 2 numbers, each consisting of save and carry parts, are reduced to one. The bitwidth of the input numbers is unrestricted and determined by the designer. The compressor is realized with full adders.

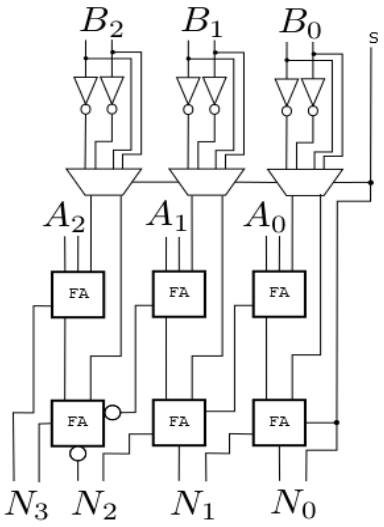

Fig. 1. 4:2 compressor with 2:1 mux for negative numbers

Figure 1 shows the architecture of the produced circuit. The input numbers A^* and B^* and the output number N^* are always in CS. The layout allows the component to perform the operations of addition or subtraction. The type of operation to be performed is controlled by the select bit S.

If the signal S is 0, the chosen operation is the CS addition $A^* + B^*$. The 2:1 multiplexer allows the number B^* to pass without further alteration to it. The addition is performed in parallel without propagating the possible carries from each column. The resulted number N^* is in CS format, which means that the possible digits of this number are $\{0,1,2\}$.

On the other hand, if the signal S is 1, the dictated operation is the subtraction $A^* - B^*$. This operation requires the signal B^* to be in two's complement, as the only action the module is able to perform is the addition. Figure 1 shows that the 2:1 multiplexer accepts the signals B^* and the inverted signal B^*_{1s}. Where B^*_{1s} is actually the one's complement of the original number and it is composed from the carry and save parts. Transformation of both of these parts to two's complement is required. For this reason the select bit S becomes the carry-in in the two LSB places. This translates to the addition of 1 to the already one's complement numbers, which transforms

them to two's complement. Equation 4 shows the subtraction $A^* - B^*$.

$$\begin{aligned} N^* &= A^* + B^*_{1s} = \\ &= A^* + B^s_{1s} + B^c_{1s} + 1 + 1 = \\ &= A^* + B^s_{2s} + B^c_{2s} = \\ &= A^* + B^*_{2s} = A^* - B^* \end{aligned} \tag{4}$$

Where A^* is the CS number, B^s_{1s} and B^c_{1s} are the one's complement of the save and carry parts, respectively B^s_{2s} and B^c_{2s} are the two's complements and B^*_{2s} is the two's complement of the CS number B^*. In the second line the addition of the two aces constitutes the addition of the signal S.

VI. Experiments and metrics

To present a few metrics and experimental results and evaluate the produced designs, we have generated and synthesized a large number of compressors. It is worth mentioning that there is no online tool able to generate custom compressors in HDL. For this reason we cannot provide comparison results from other authors.

A few representative samples are shown on Table 1. The synthesis was realized in Xilinx ISE 13.3 for the Virtex 6 FPGA family (XC6VLXl760, speed grade -1). The columns which show the number of transistors and components were calculated by our tool. Our non-pipelined design for 8 bit numbers was clocked at over 500 MHz and uses only 15 slices.

TABLE I. Automatically generated compressors for various bitwidths.

#bits	#transistors	#components
8	356	16
16	708	32
32	1412	64
64	2820	128
128	5636	256

TABLE II. Synthesis results from Xilinx ISE.

#bits	#slices	freq(MHZ)	power(Watt)
8	15	533.902	4.447
16	27	544.365	4.447
32	63	500.751	4.447
64	116	540.540	4.447
128	219	532.481	4.447

VII. Conclusions

Our contribution to the electronic design automation (EDA) domain, is the creation of a generator, accessible through the web, able to produce custom bitwidth 4:2 compressor designs in VHDL. The compressors utilize the benefits of CS arithmetic, a redundant numbering system, which greatly speeds up the performance of arithmetic operations. The produced designs are syntactically correct and can be synthesized and verified to confirm our results. Moreover, they can be implemented in FPGA or ASIC circuits and are vendor neutral. The output of our tool includes the VHDL codes for the compressor, the library with the used components, the block schematic and some calculated metrics such as the number of transistors. No other online tool which offers the same functionality exists to date.

References

[1] R. Chotin-Avon, S. Belloeil and H. Mehrez, "Arithmetic data path optimization using borrow-save representation," in *Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual*, 2008.

[2] Y. Dumonteix and H. Mehrez, "A family of redundant multipliers dedicated to fast computation for signal processing," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva.*, 2000.

[3] D. Neuhuser and E. Zehendner, "Reduced redundant arithmetic applied on low power multiply-accumulate units," in *11th WSEAS international conference on Electronics, Hardware, Wireless and Optical Communications*.

[4] N. Homma, T. Aoki, T. Higuchi, "A systematic approach for designing redundant arithmetic adders based on counter tree diagrams," *IEEE Transactions on Computers*, vol. 57, 2008.

[5] T. G. Noll, "Carry-save architectures for high-speed digital signal processing," *Journal on VLSI Signal Processing*, vol. 3, pp. 121–140, 1991.

[6] Y. Kim and T. Kim, "Accurate exploration of timing and area tradeoffs in arithmetic optimization using carry-save-adders," in *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, 2001.

[7] M. Dasygenis, "A distributed vhdl compiler and simulator accessible from the web," in *PATMOS 2014 Conference: Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on, At Mallorca*, 2014.

[8] M. Dasygenis, "A web eda tool for the automatic generation of synthesizable vhdl architectures for a rapid design space exploration," in *DTIS 2014 Conference: International Conference on Design and Test of Integrated Systems in Nanoscale Technology 2014, At Santorini, Greece*, 2014.

[9] A. Weinberger, "4:2 carry save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, 1981.

[10] M. Ghasemzadeh, A. Akbari, K. Hadidi, A. Khoei "A novel fast glitchless 4-2 compressor with a new structure," in *Mixed Design of Integrated Circuits and Systems (MIXDES), 2014 Proceedings of the 21st International Conference*, 2014.

[11] D. Bahrepour, M.J. Sharifi, "A novel high speed full adder based on linear threshold gate and its application to a 4-2 compressor," *Arabian Journal for Science and Engineering*, vol. 38, pp. 3041–3050, 2013.

[12] R. Hussin, A. Y. Shakaff, N. Idris, Z. Sauli, R .Che Ismail, A. Kamarudin, "Redesign the 4: 2 compressor for partial product reduction," in *Conference: ACST07: Proceedings of the 3rd IASTED Conference on Advances in Computer Science and Technology*, 2007.

[13] A. Pishvaie, G. Jaberipur, A. Jahanian, "Redesigned cmos (4; 2) compressor for fast binary multipliers," *Journal on Electrical and Computer Engineering*, vol. 36, pp. 111 – 115, 2013.

[14] A. Pishvaie, G. Jaberipur, A. Jahanian, "Improved cmos (4; 2) compressor designs for parallel multipliers," *Journal on Electrical and Computer Engineering*, vol. 38, 2012.

[15] A. Fathi, S. Azizian, K. Hadidi, A. Khoei, A. Chegeni, "Cmos implementation of a fast 4-2 compressor for parallel accumulations," in *Circuits and Systems (ISCAS), 2012 IEEE International Symposium*, 2012.

[16] A. Fathi, S. Azizian, K. Hadidi, A. Khoei, "A novel and very fast 4-2 compressor for high speed arithmetic operations," *IEICE Transactions on Electronics*, vol. E95.C, 2012.

[17] synopsys. [Online]. Available: http://www.synopsys.com/Tools/Pages/default.aspx

[18] myhdl. [Online]. Available: http://www.myhdl.org/info.html

[19] SPARK. [Online]. Available: http://mesl.ucsd.edu/spark/

[20] F. de Dinechin. (2011) flopoco. [Online]. Available: http://flopoco.gforge.inria.fr/

[21] spiral. [Online]. Available: http://www.spiral.net/hardware.html

[22] P. Schaumont. (2010) gezel. [Online]. Available: http://rijndael.ece.vt.edu/gezel2/codegeneration.html

[23] Y. Yankova, K. Bertels, S. Vassiliadis, R. Meeuws, A. Virginia, "Automated hdl generation: Comparative evaluation."

[24] D. Crockford. (2006) The application/json media type for javascript object notation (json). [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt