A generator tool for Carry Look-ahead Adders (CLA)

Dimitris Ziouzios[1,*], *Dimitris* Tsiktsiris[1,**], *Nikolaos* Baras[1,***], *Stamatia* Bibi[1,****], and *Minas* Dasygenis[1,†]

[1]University of Western Macedonia, Department of Electrical and Computer Engineering, Kozani, Greece

Abstract. A carry look-ahead adder (CLA) is a digital circuit which is used widely used in any electronic computational device to improve speed calculation by reducing the time required to define carry bits. Despite the fact that CLA is used massively in modern digital systems, there is no online tool to automatically generate the HDL description. For this reason we developed a cloud based tool to automate the design of optimized CLA and provide custom testbenches to verify their correctness for singed and unsigned numbers. It is also can be used by the students to create and understand deeply the way CLA works.

1 Introduction

An adder is a digital circuit which is capable of adding numbers and it is used for adding data in the processor[8]. A carry-lookahead adder (CLA) improves speed calculation by reducing the time required to define carry bits [1]. CLA is a very important building block for any digital circuits and it is a very complex binary adder[6]. The design automation and test processes (DAT) is very important for our digital era in which we are living. One of the advantages of DAT is the fast parameterized generation of bit accurate models and their test vectors, in a hardware description language (HDL). With DAT the designers are able to perform a very fast design space exploration and can pick the best implementation. Also, for almost every digital system the HDL generators for constructing circuits are very important[7]. Also, is the most important basic operation on digital systems [9]. Carry Look-ahead Adder (CLA) is one of the best choices to perform addition with multiple vectors. [8].

Although CLA design is used massively as an arithmetic module and in fact that is so important in circuits design, none until now presents a tool which is able for automation design and testing of parametrized CLA. Here, we present a web based tool, which is accessible from any web browser or even from a mobile device. It is easy to use and with reliable results. A custom CLA module can be generated using our tool, which is very important because the engineers increase their productivity and save time. Major EDA companies, like Xilinx or Altera, do not provide a tool like ours which is designed with IP-blocks generators, is cloud

[*]e-mail: dziouzios@uowm.gr
[**]e-mail: dtsiktsiris@uowm.gr
[***]e-mail: nbaras@uowm.gr
[****]e-mail: bibi@uowm.gr
[†]e-mail: mdasygenis@uowm.gr

based and free so anybody has access to use it. Moreover, students can easily and freely use this tool to create and understand in depth the way the CLA is working.

The programming environment for implementing Carry look-ahead adder is based on Python language which generates VHDL code that allows different levels of abstraction to be mixed in the same model, so, we can define a harware model in terms of gates, switches, RTL or behavioral code [2].

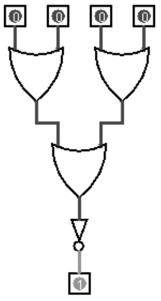


Figure 1. The implementation steps of the design using VHDL and Spartan-3 FPGA board.

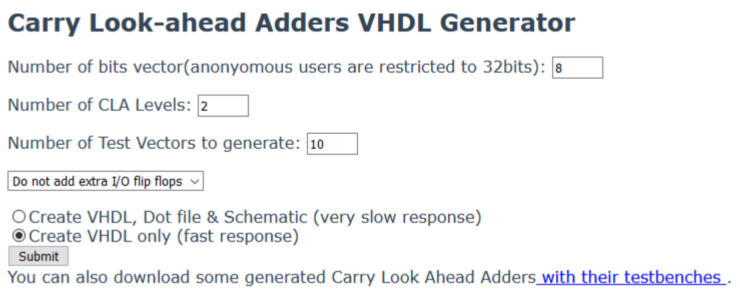
2 Related Work

According to a recent research [10] the last decade there are many research projects on automating HDL generation, but the results are not encouraging. Due to lack of automation and optimization, the automated hardware generation can't become a widely adopted industrial practice. Many complex systems, reconfigurable compuring systems, System-on-Chip. embedded systmes and more, have many demanding design requirements, which claim a thorough design exploration. For all these reasons, we developed our cloud based tool to support the designers with the EDA automation process. The leading EDA companies to this section, such as Xilinx, Altera, Mathworks, support customized IP blocks for different functions. For example, Xilinx provides the "Core Generator" and Altera the "Mega Wizard Generator". Of course, these tools can be used to create HDL descriptions of typical adders but for a certain range of bitwith and vectors. Daveau, Marchioro, Valderrama and Jerraya [3] present a way from SDL models to VHDL generation, but their work focuses only on the specification and there is no EDA tool. Lily Kanoriya, Aparna Gupta, Dr. Soni Changlani give a method of designing an IC layout using 90nm technology but they don't design any tool for this.

All the automating high level HDL generators accept a high level description of the code, commonly in C, and create an HDL description of it. The main disadvantage of high level synthesis is that the input code demands a lot of information, such as perfectly nested loops, regular memory and more while all the tools have a limited application domain. For example, the Streams-C project [5] and DWARV [11] support a very limited application domain. All the major EDA companies (Vivado, Triton, Altium etc.) provide a high level synthesis solution, but they support only typical design requirements and doesn't give the choice for parametrized CLA. Razvan Nane et al. represents a broad evaluation of several HLS tools where we can see from the results that no single tool produced the best results for all benchmarks.

3 The web CLA hardware compiler

To design our tool, we used various technologies such as Python, PHP, JSON, to attain the correct syntax and a synthesizable VHDL description. Our tool is free to be used by anyone on our public server[1]. Generally, our tool consists of two different partitions: the front-end and the back-end which exchange information with each other using the javascript object notation (JSON) format [4]. Figure 2 depicts the front-end of our tool and the multiple configuration options.


Carry Look-ahead Adders VHDL Generator

Number of bits vector(anonyomous users are restricted to 32bits): 8

Number of CLA Levels: 2

Number of Test Vectors to generate: 10

Do not add extra I/O flip flops

○ Create VHDL, Dot file & Schematic (very slow response)
● Create VHDL only (fast response)

You can also download some generated Carry Look Ahead Adders with their testbenches .

Figure 2. Users can configure a variety of settings when using our CLA tool.

The front-end partition is the form where the user inputs the desire data. In the first field the user defines the number of bits, up to 32bits for anonymous users, but if someone needs bigger number can contact us for full access. The next field is where the user puts the number of test vectors that wish to generate. The third field is about the number of CLA Levels and the user can choose to add or not extra input/output flip flops. The last one choice is to define to create VHDL only, which has fast response, or to create VHDL, Dot file and Schematic, which has very slow response. Finally, the user submits the form and a logfile of our tool appears and at the bottom of the page the user can download the results. Both partitions are crucial; the back-end includes all the basic functions therefrom we will concentrate on it. In the execution flow of the tool, we can identify three major subtools: (i) a tool that designs the RTL structure in our HDL netlist according to our defined rools (we call this tool CLA designer), (ii) our HDL compiler tool that can generate syntactically correct register transfer HDL, given an HDL netlist, and (iii) the test vector generator module, that can emmit a number of random generated test vectors together with their assert commands in VHDL to automatically verify the robustnness and functionallity of our structures.

3.1 CLA Designer

The whole idea behind the carry lookahead adder (CLA) is that we dont wait for the carries to ripple through the circuit. CLA uses the concepts of Generating (G) and Propagating (P) carries. To do this we connect full-adders to make a multi-bit carry-propagate adder, right-most adder adds least-significant bits. Carry-out is passed to next adder, which adds it to the next-most significant bits and so on. We can extend this to any number of bits. By pre-computing the major part of each carry equation, we can make a much faster adder. We

[1]The tool is available at http://arch.ece.uowm.gr/hdl/cla.php

start by computing the partial results for each bit:

$$P_i = A_i + B_i \text{ (called "propagate term")} \tag{1}$$

$$G_i = A_i B_i \text{ (called "generate term")} \tag{2}$$

Then:

$$C_i{+}_1 = G_i + P_i C_i \text{ (carry-out from bit i)} \tag{3}$$

We can compute these for each bit independently (no ripple) Each of the carries can now be expressed in terms of P and G:

$$C_i{+}_2 = G_i +_1 + P_i +_1 C_i +_1 \text{ (carry-out from bit i+1)} \tag{4}$$

$$C_i{+}_3 = G_i +_2 + P_i +_2 C_i +_2 \text{ (carry-out from bit i+2)} \tag{5}$$

$$C_i{+}_4 = G_i +_3 + P_i +_3 C_i +_3 \text{ (carry-out from bit i+3)} \tag{6}$$

And at the end, by forward substitution

$$C_i{+}_1 = G_i + P_i c_i \tag{7}$$

$$C_i{+}_2 = G_i +_1 + P_i +_1 G_i + P_i +_1 P_i C_i \tag{8}$$

$$C_i{+}_3 = G_i +_2 + P_i +_2 G_i +_1 + P_i +_2 P_i +_1 G_i + P_i +_2 P_i +_1 P_i C_i \tag{9}$$

$$C_i{+}_4 = G_i{+}_3 + P_i{+}_3 G_i{+}_2 + P_i{+}_3 P_i{+}_2 G_i{+}_1 + P_i{+}_3 P_i{+}_2 P_i{+}_1 G_i + P_i{+}_3 P_i{+}_2 P_i{+}_1 P_i C_i \tag{10}$$

So we can express each carry as a function of generate and propagate signals; in a two level AND-OR circuit and without any ripple effect. We can see in Table I, which is a truth table, when a carry generates. Our module works by the same way. In our laboratory we developed a netlist with the name HDL, which is create by the CLA design tool as an internal format in three steps. The first step accepts as input the number of bits vector, the number of CLA Levels and the number of Test Vectors to generate. Right-most adder adds least-significant bits. Carryout is passed to next adder, which adds it to the next-most significant bits until the end of bits. On the second stage it starts by computing the partial results (P and

Table 1. Truth table for carry

A	B	Ci	Ci+1	Condition
0	0	0	0	No carry generate
0	0	1	0	No carry generate
0	1	0	0	No carry generate
0	1	1	1	No carry propagate
1	0	0	0	No carry propagate
1	0	1	1	No carry propagate
1	1	0	1	Carry generate
1	1	1	1	Carry Generate

Table 2. Overflow flag on addition sign bits

num1sign	num2sign	operation	sumsign	flag	comments
0	0	addition	1	ON	should be positive
1	1	addition	0	ON	should be negative
0	1	subtraction	1	ON	should be positive
1	0	subtraction	0	ON	should be negative
The flag is off to all other cases					

G) and finally by forward substitution we can express each carry as a function of generate and propagate signals; in a two level AND-OR circuit.

Furthermore, we augment the functionality of this tool, adding automatically (*i*) an overflow flag, (*ii*) a zero flag, (*iii*) a signed flag and (*iv*) a carry flag when is necessary. All these flags are very important to modern ALU units, because are used for conditional execution branches. In particular, the overflow flag used when we have signed numbers [14] and we want to do an addition or a subtraction. There are many methods to detect overflow errors in 2's complement. In our tool the logic to detect overflow is to pay attention only to the sign bit which is the most significant bit. As we can see on Table II, there are four cases where we have to use the overflow flag (two cases for additional and two for subtraction). So, when we add/subtract two positive numbers and got a negative, or add/subtract two negative numbers and got a positive the result in both cases is wrong. To solve this problem we use the overflow flag to mention the problem. For example, if we have to subtract two positive numbers $num1sign = 0100$ and $num2sign = 0100$ the output result is negative $sumsign = 1000$ which is wrong. In this case, we use the overflow flag. The zero flag is very important to check the equality of two numbers. Technically the equality is an indirect abstraction and we can see the schematic equation of two 2bits numbers in Fig. 1. As we can see it is a bitwise reduction in parallel operation which is faster than the cascade method. Finally, we have the signed flag for the most significant bit and the carry flag which is created by the adder.

3.2 HDL compiler tool

Our HDL compiler tool has a big advantage that easily can be connected to different generators and accepts as input our compact netlist HDL format [15]. We can describe this special netlist as a hypercube, as shown in Fig. 2 where every knot corresponds to a component and carries the vectors that define the input connections. The netlist format endorsed in a single javascript object notation format (JSON).

Table 3. Estimated implementation metrics

nBits	nSlice LuTs	Delay	Frequency
4	8	7,406ns	135Mhz
8	22	7,641ns	130Mhz
16	93	7,505ns	133Mhz
32	272	7,437ns	134Mhz

We can see how our HDL compiler tool works in five steps: (*i*) authentication, (*ii*) top-level input output analysis, (*iii*) port mapping, (*iv*) generate HDL and (*v*) finally the generation of schematic.

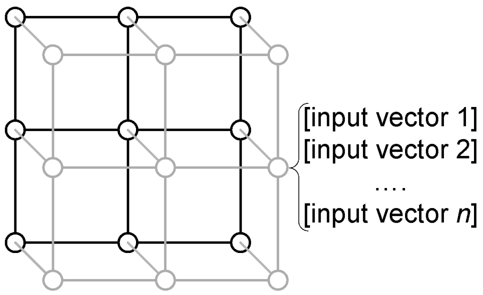
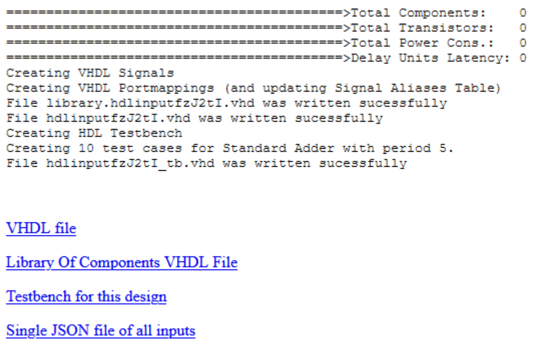


Figure 3. The visualization of the a-HDL netlist format forms a hypercube

On the first step, the HDL format is checked for nonconformities that may exist. Furthermore, in this step, all the components which used in the netlist is checked with the HDL library of the tool to confirm the compatibility. Continuing to the second step, the toplevel input and output analysis is executed. At this point, the tool examines every vector and registers the input vectors that point as origin an input port of the design. On the third step, port mapping, signals are created and connected to specific port numbers and types. In this step the input vectors of every component are examined to find a connection between the source and the destination component. For example if a signal name exists with the same attributes (i.e. type, bitwidth etc.), is registered as it is to the portmap structure, in other case a new signal name is created with the new attributes.

The HDL generator is the fourth step. We get to outputs VHDL files: i) a VHDL file that carries the entities and the architectural descriptions o all primitive components which used in the design, and ii) a VHDL file that describes the design derived from the netlist. At this point, the tool knowing the type of signals that used and puts the appropriate library declarations. With the use of input and output ports data structure, the tool creates the entity definition of the design. After that, using the appropriate data structure, portmapping declarations strictly adhering to the VHDL syntax. The last fifth step, includes the visual representation. A graph is constructed and rendered as png picture format, using the DOT visualization language. At this step, it is necessary to use the portmapping input and output port data structures, to complete all connections.We have to mention that our web tool does not synthesize the circuit because it is not made for this purpose but generates optimized VHDL codes which can be used by the engineer to their own synthesis tool. Figure 3 shows the results, after the CLA tool has finished running. Users can choose to download the VHDL file, download the library of components VHDL file, testbench for the specified design or download a single JSON file for all the inputs.



```
==========================================>Total Components:    0
==========================================>Total Transistors:   0
==========================================>Total Power Cons.:   0
==========================================>Delay Units Latency: 0
Creating VHDL Signals
Creating VHDL Portmappings (and updating Signal Aliases Table)
File library.hdlinputfzJ2tI.vhd was written sucessfully
File hdlinputfzJ2tI.vhd was written sucessfully
Creating HDL Testbench
Creating 10 test cases for Standard Adder with period 5.
File hdlinputfzJ2tI_tb.vhd was written sucessfully
```

VHDL file
Library Of Components VHDL File
Testbench for this design
Single JSON file of all inputs

Figure 4. Final screen of the CLA tool showing all the available options.

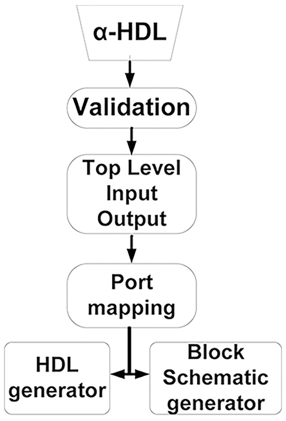


Figure 5. The HDL Generator accepts our compact netlist and outputs the VHDL files and the schematic.

3.3 Test vector generator module

The last but overly important for our tool, is the HDL Test Vector Generator Module, because it creates multiple vectors of testbenches which are very useful to be used to test the correctness of the design in an HDL simulator. It is obvious the complexity of the process of the CLA design and should be tested carefully at every stage. For doing this, firstly it creates an unfilled entity declaration, after that it instantiates the top level component and creates signals for every input and output port. Moreover, converts bits to an integer by creating a clock process and a function. Next, creates the requested number of input cases. At the end, in the HDL simulator, a comparison is made between the output value of the design with the value that has been computed. If are equal then the test passes and a success message is printed, differently a fail message printed with all the details. However, until this period of time all the CLA designs which are created by our tool pass all the tests. According to the requested number of tests by the user, all the testbench vectors are created randomly. All the checks which are needed, are done automatically, so the designer can load the testbench file in this HDL Synthesis and Simulator tool without any other mediation. If there is a problem with a vector, a message will be printed and the designer may review it at anytime, which is improbable to happen because of the plenty tests we have done. Using our tool, the designer quickly and reliable can test different large vectors.

4 Experimental Results

In order to check the effectiveness of our cloud based tool and the accuracy of the results, we generated plenty of VHDL descriptions with different parameters. We have to mention that we cannot compare our output results with other similar tools, because they don't exist at least at this time. That's why we present some indicative results which are easy to be verified by our tool. Also, on the bottom of our tool's webpage anyone can download over 150 automatic generated and tested Carry Look- ahead Adders with their testbenches.

Conclusions

For the designers it is important to use a tool for automation designing and fast circuit verification which helps to save time and be more productive. Carry look-ahead adder is a very important component to plenty digital system due to the speed. Our web tool for designing custom CLA devices, having no restrictions on the number or size of bits, it is reliable and anybody can have access to it. The outputs of this tool are a VHDLS file with synthesizable description, automated testbench, block schematic, Dot file and other metrics. The generated code can be synthesized in FPGA or in ASIC projects.

References

[1] F. C. Cheng et al. "Delay-Insensitive Carry-lookahead Adders". In: *VLSI Design, 1997. Proceedings*. 1997, pp. 322–328.

[2] Sandeep Dahiya and Rajender Kumar. "Performance Analysis of Different Bit Carry Look Ahead Adder Using VHDL Environment". In: *International Journal of Engineering Science and Innovative Technology (IJESIT)* 2 (July 2013). Issue 4.

[3] Jean-Marc Daveau et al. "VHDL generation from SDL specifications". In: Springer US, 1997. Chap. Part of the IFIP — The International Federation for Information Processing book series (IFIPAICT).

[4] I. E. T. Force. *Introducing JSON*. Sept. 2013. URL: http://www.json.org.

[5] M. Gokhale et al. "Streamoriented FPGA computing in the Streams-C high level language". In: *IEEE Field-Programmable Custom Computing Machines*. 2000, pp. 49–56.

[6] M. M. Mano and C. R. Kime. *Logic and computer design fundamentals*. Prentice-Hall, 2001.

[7] M. Nicolaidis. “Carry checking/parity prediction adders and ALUs”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11(1) (Mar. 2003), pp. 121–128.

[8] J. M. Rabey. *Digital Integrated Circuits, A Design Perspective*. Prentice-Hall, 1996.

[9] E. E. Swartzlander. “Computer Arithmetic”. In: *Los Alamitos, CA: IEEE Computer Society Press* 1 (1990).

[10] Y. Yankova et al. “Automated HDL generation: Comparative evaluation”. In: *Circuits and Systems*. IEEE, May 2007, pp. 2750–2753.

[11] Y. Yankova et al. “DWARV: Delftworkbench automated reconfigurable VHDL generator”. In: *Field Programmable Logic and Applications, 2007. FPL 2007*. 2007, pp. 697–701.