International Journal of Applied Information Systems (IJAIS) ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 12 - No. 13, May 2018 - www.ijais.org

Improvement in Logisim to Digital Systems Simulation in Higher Levels of Abstraction and Synthesis

Tiago da Silva Almeida
Universidade Federal do Tocantins,
Computer Science Department
Palmas / TO, Brazil

Pedro Henrique de Castro Lima
Universidade Federal do Tocantins,
Computer Science Department
Palmas / TO, Brazil

Rafael Lima de Carvalho
Universidade Federal do Tocantins,
Computer Science Department
Palmas / TO, Brazil

Warley Gramacho da Silva
Universidade Federal do Tocantins,
Computer Science Department
Palmas / TO, Brazil

ABSTRACT

The development of digital systems requires an extreme attention by the circuit designer due to the different abstraction domains that the same system could be. This fact brings many issues and challenges in circuit design, due to the wide range of levels and representations. There are many details the designer have to concern, such as area, performance, architecture and energy consumption. To aid in different representations, this paper brings up a framework that is able to translate schematics of digital systems built using the CAD tool Logisim, into implementations at the hardware description level, to help in issues and to teach future designers in the academy. It was built a checker of a model after and before of the synthesis to ensure the model validity and tests. The HDL chosen was SystemC because it is easy to compile and check in any open source C++ compiler. The set of tests applied to 41 different circuits models have shown that the proposed tool works effectively ensuring the desired output.

General Terms:

Hardware, Electronic design automation, Hardware description languages and compilation

Keywords

Digital Systems, Synthesis, Computer Aided Design, Logisim

1. INTRODUCTION

Microelectronics is a key field to construct more efficient and effective circuits. In a typical design, there are lots of levels of abstraction in different issues, which increase the complexity of designs. To resolve these issues, both industry and academy usually employ a large gamma of computational tools, referred as Computer Aided Design (CAD) tools.
Nowadays, CADs are almost indispensable and with them, it is possible to optimize the design in many levels, reuse components, share them and so on. In academy is even more important to make the concept abstract for future engineers and designers of new technologies.
A major problem using CAD is related to high costs. The manufacturers create CADs complex and very useful in different designs, unfortunately at great costs. This fact harms the dissemination of technicologies and methods to perform better architectures of circuits, especially in developing countries and universities with a restricted budget.
To help to solve this issue, in this paper it is proposed an improvement to a simple digital simulator in order to accommodate new features to the translation of a schematic digital circuit to a hardware description language (HDL). It was used the Logisim simulator and the SystemC as HDL. Logisim [7] was chosen because it is open source, free and multiplatform. It is frequently used in the academy and has a good documentation. Moreover, SystemC [3] was chosen because it has an open source license and it can be compiled by any C++ compiler. Moreover, it allows integrating with other CADs.
It can be cited the work in [2] for which the goal is to improve the Logisim. This work was named Logisim-Evolution, it allows to synthesize the schematic in FPGA using VHDL or Verilog language. However, the focus of the proposed work is in simulation instead of implementation of circuits.
The process of design or codesign is fundamental in microelectronic design and it has been intensively researched over many years, e.g. in [9] it is proposed a translator between SystemC and VHDL, with focus on SoC (System on Chip). SoCs represent an important step for microelectronics, it allows to embed many components in the same package, creating a mixture of specific and general purpose applications.
In [13] it has been built a model named speculative code motions, for which the blocks of code are modeled as a directed graph. The authors claimed that it is useful to alleviate the effects of programming styles. Despite the goal in [13] be different from this work's, it is an important evaluation of translation decisions in a design which reflects in better or worse final circuit designs.
In addition, both [12] and [6] have worked with SystemC through TLM (Transaction Level Modeling) [10] in order to simulate systems. Both of them takes the SystemC model and transform it into a formal verification model, which allows checking upon the system attributes. Moreover, in [12] the authors used the UML language as well as in [6] the chose technology was Petri nets. In cases, the model was represented as ESL (Electronic System Level), because it is considered an ideal level of codesign [11]. In summary, there are other works involving translation/synthesis, like [14], [17], and [8], beyond others which only evaluate the SystemC performance on simulations, like [4], only to cite some.
The present work, however, represents an effort to reach another step towards the microelectronic design with respect to the integration of open source and public license tools in order to help both designs of microelectronic projects as well as disseminate the knowledge in this area.
The paper is further divided into the following sections: Section 2, which it is explained the proposed methodology as well as the employed methods to translate models; in Section 3 is presented the performed experiments and their respective results; lastly, the Section 4 summarizes the conclusions and points out some future appointments on this research field.

2. METHODOLOGY

The framework aims at a translation of a source code from a base language. In this case, the source code is composed of circuit diagrams drawn from a CAD tool for a hardware description language. This translation of the source program, known as the compilation, can be associated with a change in the level of abstraction, called synthesis. Thus, the design is described in compilation phases and the step of obtaining the input uses the source program of the tool Logisim. In order to illustrate the general idea, Fig.1 shows a step-by-step of the framework process.
The models generated by Logisim are diagrams drawn in a construction area or drawing area. The components are inserted in this area and consequently mapped on a 2D (X, Y) coordinate scale. Although the modeling of a circuit is elaborated in forms of graphical schemes, there is a way to obtain information about each component such as the number of inputs and outputs, bit range and type of component (multiplexer, demultiplexer or adder), without the need to process the graphics maps of the drawing area.
The Logisim source code has a class called "*Analyze*". This class was made specifically for the combinational analysis step of the constructed circuits. When invoking the combinational analysis function of a design, depending on the types of circuits used, a boolean expression is generated or, if the circuit does not use logic gates directly, a truth table is generated and a boolean expression is set up for each possible output.
The next step in generating the initial code is the storage of the data obtained through Combinational Analysis, in models that contain information about the inputs, circuit name, outputs, and expressions that govern this circuit. The " com.uft.logisim.entity " package is the part of the synthesis *framework*.
For the representation of the circuit as a whole, there is a class called "*CircuitModel*", which stores the real name of the circuit created together with the list of inputs and outputs from its form of the Input and Output structures. The CircuitModel structure and the classes contained in it were created so that they can be easily adapted to the intermediate code created for sharing the information collected.
The last element within the model representation is the class "*CircuitBasic*". Such a model has been defined for the aid of data interchange between the structures of the Logisim and the CircuitModel of the synthetic framework. CircuitBasic receives the Logisim project information in use, the desired circuit and a key-value structure of the input and output components, where the key is represented by the component, in this case only pins, and the value is its respective name.
The simple transformation of a circuit diagram into a hardware description language solves the problem of not detailing the other circuit domains (structural, physical, and behavioral). However, the problem of the time required for the development of the electronic devices and the high number of described components are not solved by the translation itself. It is necessary to perform a refinement of the elements used through optimization heuristics.
In order to allow optimizations through external tools, it is interesting to use an intermediate code that will be sent to a minimizer and returned to the compiler to perform the remaining steps of the circuit translation.
Such an intermediate code is based on JSON (JavaScript Object Notation) for structuring the data. The source code of the *Logisim* toolkit was developed in Java which makes it highly compatible with JSON.
A template for the information exchange code was developed specifically for this project following the structure of the class *CircuitModel*. The information is based on results or outputs of the modeling of microelectronic devices. Therefore, for each obtained solution, an object containing the entries that generated such result, the expressions referring to that output and the label thereof are passed to the minimizer as in the example of Fig. 2 which describes a simple multiplexer 4 x 1, e.g.. The code coming from the minimization step follows the same format as the initially submitted template.
The generation of the intermediate code is performed within the class *CircuiParser*. Once truth table and Boolean expression have been loaded into the "*AnalyzerModel* " object of the *CircuitParser* class, there is the padding of the *CircuitModel* object, input variables in a list of inputs and output variables in a list of outputs. In each output, element is included the original expression and the minimized expression applied to the product of sums rule as standardization of the format of the expressions.
Once the *CircuitModel* object has its variables loaded, then the intermediate code generation occurs in the JSON format. This generation is triggered by the *generateJSON* method of class *CircuitParser*. The file is generated through an API (Application Programming Interface) helper called *Gson* [1] which performs objects to the JSON format.
Reading an external JSON file is handled by the "*MiddleCodeReader*" class of the "com.uft.logisim. extract" package through a static method for reading the contents of the file and converting it for the CircuitModel model through the Gson API. With the file loaded from the framework, the next step is to convert the obtained data (inputs and outputs) into a code template that follows a structure similar to the code end, which is the code in SystemC.
The initial code, which is also an intermediate code for the circuit optimization step, is not in the desired code pattern in SystemC as can be seen in the comparison between the Fig. 2 in JSON and 4 in SystemC. Therefore, before the initialization of the lexical analysis phase, it is necessary to match the obtained information organized into a modular template of a circuit (Fig. 3).
Analyzing the final code, represented in Listing 4, it is noticed that there exists a modular structure called "*SC_MODULE*" which is the module that involves a circuit and its components. Such a module is described by a name, statement of input and output variables characterized by the logical type "*sc_logic*" a constructor denoted by "*SC_CTOR*" and a method that processes entries according to the behavior described by the Boolean expression of this circuit.
The adaptation of the JSON code for the template is performed by the "*CompilerCodeParser*" class of the "com.uft.logisim.extract" package through the "createCompilerMiddleCode" method which standardizes the input structure in something closer to the final code to performing the lexical analysis.
The definition of the tokens allowed in this language is done in the abstract class "*BooleanAlgebraPatterns*" of the "com.uft.logisim.pattern" package. The list of tokens and their respective regular expressions is arranged in the Table 1
The lexical analyzer of this work performs two functions:

—verify the existence of the input value in the language dictionary;

—assist in the generation of the final code by replacing the input with tokens which are known by SystemC as can be seen in the Table 1.

Through the analysis of the number of allowed tokens it is noticeable that the language for specifying a digital component is small, however the framework has been implemented in order to allow future expansions for a greater detailing and generation of codes besides the modular structure here presented.
In order to allow greater freedom in language construction work, the lexical analyzer has been completely built specifically for the modular SystemC code of a circuit. Its structure was divided into phases that are: phase of conversion of expression to the fixed form, phase of generation of non-deterministic finite automata with empty movements through the Thompson algorithm [5, 16],

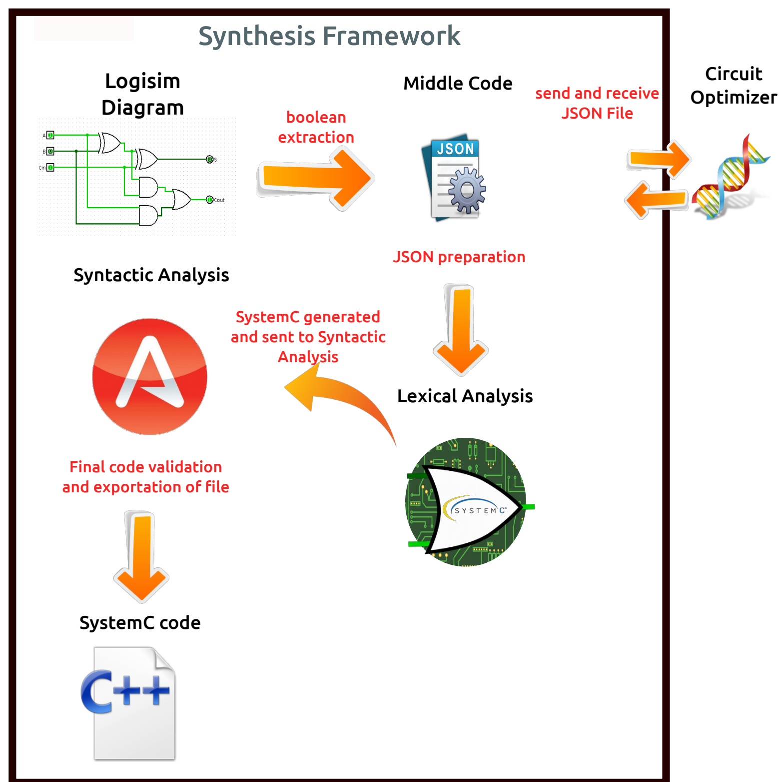

Fig. 1. Descriptive diagram of all steps in the proposed framework for Logisim. The black rectangle represents our framework improvement for Logisim. Inside is described the compilation phases which does the check upon at the generated code in SystemC.

```
1  {
2    "circuitName": "mux4x1",
3    "inputs": [
4      {
5        "name": "a"
6      },
7      {
8        "name": "b"
9      },
10     {
11       "name": "c"
12     },
13     {
14       "name": "d"
15     },
16     {
17       "name": "ctrl1"
18     },
19     {
20       "name": "ctrl2"
21     }
22   ],
23   "outputs": [
24     {
25       "expression": "d ctrl1 ctrl2 + c ctrl1 ~ctrl2 + b ~ctrl1 ctrl2 +
           a ~ctrl1 ~ctrl2",
26       "minimizedExpression": "(a + ctrl1 + ctrl2) (b + ctrl1 + ~ctrl2)
           (c + ~ctrl1 + ctrl2) (d + ~ctrl1 + ~ctrl2)",
27       "name": "code"
28     }
29   ]
30 }
```

Fig. 2. Intermediate code with representation in JSON for later refinement of the model through optimization processes.

```
1  MODULE[mux4x1] {
2   Inputs:a, b, c, d, ctrl1, ctrl2;
3   Outputs:code;
4   CONSTRUCTOR[mux4x1] {
5     METHOD[process];
6     SENSITIVE <<a<< b<< c<< d<< ctrl1<< ctrl2;
7   }
8   VOID process[]{
9         code = [a_READ__OR_ctrl1_READ__OR_ctrl2_READ_]_AND_[
              b_READ__OR_ctrl1_READ__OR_NOT_ctrl2_READ_]_AND_[
              c_READ__OR_NOT_ctrl1_READ__OR_ctrl2_READ_]_AND_[
              d_READ__OR_NOT_ctrl1_READ__OR_NOT_ctrl2_READ_];
10  }
11 };
```

Fig. 3. Template of multiplexer example used in lexical analysis for framework of improvement for logisim. From this template the real SystemC code is generated.

```
1  #include "systemc.h"
2
3  SC_MODULE (testbench)
4  {
5  SC_MODULE (mux4x1){
6          sc_in<sc_logic> a,b,c,d,ctrl1 ,ctrl2;
7          sc_out<sc_logic> code;
8
9          SC_CTOR (mux4x1){
10
11         SC_METHOD (process);
12         sensitive << a<< b<< c<< d<< ctrl1<< ctrl2;
13         }
14
15         void process(){
16                 code = (a.read() | ctrl1.read() | ctrl2.read()) & (b.
                       read() | ctrl1.read() |
17                    ~ctrl2.read()) & (c.read() | ~ctrl1.read() | ctrl2.
                        read()) & (d.read() |                        ~ctrl1.
                        read() | ~ctrl2.read());
18         }
19 };
```

Fig. 4. ystemC code generated for multiplexer example. Its the final code generated from schematic circuit in Logisim.

phase of conversion of the generated automaton to a finite deterministic automaton and phase analysis of the input by all the automata of the analyzer.
The class that implements this transformation is "*RegularExpressionConverter*" which makes use of two stacks, a final stack and a stack of operators. The stack is described by the "*Stack*" class defined inside the "com.uft.logisim.interpret" package. The elements that make up *Stack* are objects of the Type "*Node* ". Each *Node* stores a symbol only. In addition, there is an abstract class for the precedence evaluation called "*RegularExpressions*" which classifies the operator into integer values according to its priority degree.
The lexical analysis algorithm uses the concept of a stack automaton, yet the algorithm implemented by the synthesis framework does not merge all AFDs into a single one like many implemented algorithms. Each automaton, which represents a token, is separated into a block from a list.
The class performing the lexical analysis is labeled "*LexicalAnalyzer*" and is included in the " com.uft.logisim.automata.lexical.controller" package. The lexical analyzer makes use of a list of tokens and a list of automata that are in the same quantity, because an automaton represents a token.
The syntactic analysis of the *framework* was done using the ANTLR version 4 tool that makes use of the *Adaptive* technology LL (*) or ALL (*). Such a tool has the potential to generate syntactic and lexical parsers given a grammar without indirect recursion to the left. The ALL (*) technology performs a runtime grammar analysis by finding a recognition sequence through navigation within the grammar [15].
In this last phase of this compiler, the objective is to verify if the construction coming from the lexical analyzer is in agreement with the definitions of the Language SystemC used in this framework. The ANTLR V4 execution environment automatically generates classes for a lexical and syntactical re-analysis arranged in the "com.uft.logisim.syntax.parser" package.
The processing of the input and use of the generated parsers is done by the static method "*syntaxParse*" of the class "*GrammarRunner*". Such a method creates a lexical analyzer from the input and sends the result of the tokens to the parser doing the evaluation. If errors are found, these errors code are returned.
The grammar of the ANTLR tool understands that tiny identifiers are production rules whereas uppercase identifiers are tokens.
The use of a syntactic analyzer through ANTLR V4 provides a quick and easy evaluation of the lexical result, preventing basic errors in the construction of a circuit, such as undefined inputs or outputs. Other errors such as not defining input or output pin names and using special characters in variable labels are solved internally by the *Logisim* itself and during the template generation step. If the lexical analysis of framework does not associate

Table 1. Relation between tokens used in regular expression by lexical analysis in framework proposed.

Token	Regular Expression
IDENTIFIER	(a+...+z)(a+...+z + 0+...+9)*
[	(
]	)
{	{
}	}
;	;
<<	<<
,	,
=	=
∼	(not_)+(NOT_)
—	(_or_)+(_OR_)
&	(_and_)+(_AND_)
ˆ	(_xor_)+(_XOR_)
.read()	_READ_
sc_in	Inputs:
sc_out	Outputs:
<sc_logic>	LOGIC
sensitive	SENSITIVE
SC_METHOD	MODULE
type	VOID
SC_MODULE	METHOD
SC_CTOR	CONSTRUCTOR

an input to a language token, the token ERROR is thrown for the syntax check step, and lexical re-analysis does not recognize such a word, preventing syntactic analysis by guaranteeing reliability of results.

3. RESULTS AND DISCUSSION

All of the tests performed followed the steps described above and were run on a computer with i7 1.84 GHz processor third generation, 16 Gb of RAM running on a 64-bit Linux Ubuntu 16.04 distribution. The version of SystemC used was the 2.3.1 which is the most recent stable version when this work was done. In total, 41 different circuits were modeled and each one was generated: a JSON file, a Circuit SystemC file, a file of test and the main file or *Main* file, the last three in the *cpp*.
The Table 2 contains the results of all tested circuits. The type of circuit tested and their respective quantities of inputs and outputs are arranged in this table. The Logisim column is a reference to the production of the schematic diagram indicating whether it was constructed and performs the desired function, for example, if a 1-bit adder really does the sum. The JSON and SystemC columns describe whether the generation of the two files was successful, while the Simulation column is the result of the SystemC code tests produced after the simulation via *g++*.
All the circuits tested showed the expected behavior, there were no errors of generation of the codes in any stage and the compiler was successful in the recognition of the data inputs and translated them in an equivalent way. Differences in compile time arose when boolean expressions with more than 100 lines were tested.

4. CONCLUSION

In this work, a new solution for the synthesis of combinational circuits from the CAD *Logisim* tool was presented in a description language of hardware which in this case was SystemC. The production process of this *software*, characterized as a framework, has been described in detail.
A 4x1 (four by one) Multiplexer circuit is modeled to demonstrate how the tests were performed and under what measurements the results were qualified. There were a series of tests, totaling 41 circuits tested in total and all were successful, demonstrating the functionality and reliability of the tool.
Although the framework is able to perform its functions effectively, it is not efficient. Template generation is a slow process, especially in cases of Boolean expressions considered long (more than 100 lines of expression). The next step would be to optimize the compiler by defining a new way of mapping the template or passing the responsibility to the lexical analyzer in order to identify variables in processing and tokenize them as data read.
Others points are important in this synthesis process, for sure, is necessary implement formal verification methods in more complex models to achieve codesign in TLM representation. However, at this point, the translation/synthesis model work as well in our goal.
Our tests were performed only in combinational circuits because there is a limitation in Logisim. Although there are sequential elements in Logisim, such as flip-flops and registers, it can not able to simulate properly sequential circuits. Backpropagation signals, like an internal NAND gate internally in a latch or flip-flop, are not allowed. This fact limits our framework to sequential circuits properly. However, it can expand the grammar to flip-flops and other components.
The propose this paper was present the framework, but it is necessary to measure the implications of it in a higher education with this contribution. Another task is to improve Logisim itself to simulate more complex circuits and architecture.

5. REFERENCES

[1] Google-gson, 2012.

[2] Logisim evolution git repository, 2014.

[3] Logisim: a graphical tool for designing and simulation logic circuit, March 2017.

[4] S. S. Abrar, M. Jenihhin, J. Raik, S. Kiran A., and C. Babu. Performance analysis of cosimulating processor core in vhdl and systemc. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 563–568, Aug 2013.

[5] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Monica S. Lam. *Compilers: principles, techniques, & tools*. Pearson/Addison Wesley, 2nd ed edition, 2007.

[6] I. E. Bennour. Systemc tlm2-protocol consistency checker using petri net. In *2016 11th International Design Test Symposium (IDT)*, pages 193–198, Dec 2016.

[7] Carl Burch. Logisim: a graphical tool for designing and simulation logic circuit, March 2014.

[8] D. C. Caf, F. V. dos Santos, C. Hardebolle, C. Jacquet, and F. Boulanger. Multi-paradigm semantics for simulating sysml models using systemc-ams. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pages 1–8, Sept 2013.

[9] C. Cote and Z. Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 717–720 vol.2, 2002.

[10] D. D. Gajski. System-level synthesis: From specification to transaction level models. In *2009 International Conference on Communications, Circuits and Systems*, pages 1134–1138, July 2009.

[11] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, Oct 2009.

[12] M. Goli, J. Stoppe, and R. Drechsler. Automatic equivalence checking for systemc-tlm 2.0 models against their

formal specifications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 630–633, March 2017.

[13] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. *IEE Proceedings - Computers and Digital Techniques*, 150(5):330–7–, Sept 2003.

[14] S. Ouadjaout and D. Houzet. Rapid integration of reusable functional ips with systemc vci adapters. In *Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.*, pages 236–239, Dec 2004.

[15] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[16] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[17] Chen Xi, Lu Jian Hua, Zhou ZuCheng, and Shang YaoHui. Modeling systemc design in uml and automatic code generation. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 932–935 Vol. 2, Jan 2005.

Table 2. Results of all study cases to test the proposed framework. The check sign means the translation or simulation occurred like expected.

Circuit	Input	Output	Logisim	JSON	SystemC	Simulation
Full adder 1 bit	3	2	✓	✓	✓	✓
Full adder 2 bits	5	3	✓	✓	✓	✓
Full adder 3 bits	7	4	✓	✓	✓	✓
Full adder 4 bits	9	5	✓	✓	✓	✓
Full adder 5 bits	11	6	✓	✓	✓	✓
Subtractor 1 bit	3	2	✓	✓	✓	✓
Subtractor 2 bits	5	3	✓	✓	✓	✓
Subtractor 3 bits	7	4	✓	✓	✓	✓
Subtractor 4 bits	9	5	✓	✓	✓	✓
Subtractor 5 bits	11	6	✓	✓	✓	✓
Half adder 1 bit	2	2	✓	✓	✓	✓
Half adder 2 bits	4	3	✓	✓	✓	✓
Half adder 3 bits	6	3	✓	✓	✓	✓
Half adder 4 bits	8	4	✓	✓	✓	✓
Half adder 5 bits	10	4	✓	✓	✓	✓
Comparator 1 bit	2	3	✓	✓	✓	✓
Comparator 2 bits	4	3	✓	✓	✓	✓
Comparator 3 bits	6	3	✓	✓	✓	✓
Comparator 4bits	8	3	✓	✓	✓	✓
Comparator 5 bits	10	3	✓	✓	✓	✓
Demux1x2 1 bit	2	2	✓	✓	✓	✓
Demux1x2 2 bits	3	4	✓	✓	✓	✓
Demux1x2 3 bits	4	6	✓	✓	✓	✓
Demux1x2 4 bits	5	8	✓	✓	✓	✓
Demux1x2 5 bits	6	10	✓	✓	✓	✓
Divider 2 bits	6	4	✓	✓	✓	✓
Divider 3 bits	9	6	✓	✓	✓	✓
Divider 4 bits	12	8	✓	✓	✓	✓
Even parity	2	1	✓	✓	✓	✓
Majority	3	1	✓	✓	✓	✓
Multiplier 2 bits	6	4	✓	✓	✓	✓
Multiplier 3 bits	9	6	✓	✓	✓	✓
Multiplier 4 bits	12	8	✓	✓	✓	✓
Mux2x1 1 bit	3	1	✓	✓	✓	✓
Mux2x1 2 bits	5	2	✓	✓	✓	✓
Mux2x1 3 bits	7	3	✓	✓	✓	✓
Mux2x1 4 bits	9	4	✓	✓	✓	✓
Mux2x1 5 bits	11	5	✓	✓	✓	✓
Odd parity	3	1	✓	✓	✓	✓
Mux4x1	6	1	✓	✓	✓	✓
Demux1x4	3	4	✓	✓	✓	✓