M. Radziewicz
Szczecin University of Technology
(Żolnierska 49 st., 71-210 Szczecin, Poland,
E-mail: mradziewicz@wi.ps.pl)

Translation of VHDL Sequential Statements

(Recommended by Prof. V. Simonenko)

VHDL is one of the most popular languages used in logic synthesis tools. It has variety of statements which make it powerful and flexible tool. But, as the result, it is rather difficult to create a compiler of VHDL language, especially the one which will be used in a logic synthesis. There is little information about translation algorithms used to generate hardware representation from VHDL sources. The algorithms for few sequential statements of VHDL language are developed. Apart from the algorithms themselves the paper presents a lot of information about translation process itself and all possible problems which may occur during it. Proposed solution was implemented in a compiler which uses Boolean equations as an output format. The paper includes results of tests which were performed to check practical usability boundaries of proposed algorithms.

VHDL – один из наиболее популярных языков, используемых в средствах логического синтеза. Он содержит множество операторов, которые обеспечивают его мощность и гибкость, поэтому создание компилятора языка VHDL, ориентированного на использование в логическом синтезе, – сложная задача. Информации об алгоритмах трансляции, поступающей от разработчиков VHDL, недостаточно для создания технических средств. Разработаны такие алгоритмы для некоторых последовательностных операторов языка VHDL. Предложенное решение реализовано в компиляторе, использующем логические уравнения как выходной формат. Приведены результаты тестов, выполненных для проверки границ практической применимости предложенных алгоритмов.

K e y w o r d s: electronic design automation, FPGA, VHDL, logic synthesis.

Introduction. The best thing about FPGA [1] chips is their constant grow in terms of complexity, speed and general abilities to carry sophisticated algorithms inside. In order to take full advantage of those new chips, design software must be improved as well. Nowadays, a typical synthesis tool is usually based on one of hardware description languages (HDL). Despite the number of differences, those tools are quite similar because a synthesis design flow is rather hard set. It helps designers to switch between products of different vendors.

HDL languages has gone long way from their beginnings. Now it is impossible to imagine digital circuit design without them. The languages of course dif-

fer from each other, because they have been created with a certain initial set of constrains or to solve specific design problem. For example, a main domain of Verilog was to help simulations whereas for VHDL it was a documentation of projects. They are also based on classical programming languages like C or ADA from which they partially adopted syntax and semantic.

VHDL [2] is one of the most popular hardware description languages. The purpose of its creation was to provide better documentation means. Now it supports simulation and synthesis process as well. Its syntax is very clear and easy to understand. The language provides variety of programming constructs which help designers in their tasks. Strong simulations support helps in a verification process. But what is an asset from a hardware designer point of view, can be a very big flaw for people who make such tools. And in fact it is. A process of creating new synthesis software based on VHDL is very long and difficult. There is very little information about the subject, because most of the tools are commercial products. Small number of academic publications, which usually do not cover the subject entirely. The point of this article is to fill gaps in knowledge concerning design process of logic synthesis tools which are based on HDL. Information presented in the paper is the result of a project whose goal was to create synthesis software at least as powerful as Synopsys's FPGA Express©. The tool uses Boolean equations as its output format. That unusual representation has several vital advantages:

it is purely mathematical form, completely hardware independent,

there are various well known optimisation methods for Boolean equations,

it can be simulated.

This paper focuses on sequential statements of the VHDL language. They form well distinguished part of all translation problems.

Related works. As it was stated earlier, a number of publications related to the topic is quite limited. The reasons are obvious. Most of EDA tools are property of commercial companies. It is unlikely to expect them to share their secrets with others in order to improve generally accessible knowledge. Synthesis tools market is worth a lot money, so competition is high. Typical manuals which come with design software, usually present only basic usage information. It does not explain how the tool works inside, and why it does particular thing in a certain way. Such information is sufficient for hardware designer even if it is not complete. The tool is like a 'black box'. Something goes in, something goes out, but nobody really knows what exactly happens inside. Design software suppliers provide as little information as possible. They cannot be blamed for these practices because they just intend to protect their market advantage.

There are many software companies which specialise in hardware design applications. Nowadays these tools support the whole design process, not only synthesis. It is an 'all-in-one' approach which makes a hardware designer's job

easy. Usually the whole package contains: an integrated development environment, a simulator, and a synthesiser. A part from that there a few supporting tools like floor plan editor and variety of optimisers. Those tools accepts many different input formats. The most widely know companies and products are: Synopsys (FPGA Express, Design Compiler), Altera (MAX+PLUS II,Quartus II), Xilinx (ISE), Mentor Graphics (Leonardo Spectrum), Cadence.

In order to synthesise VHDL code using commercial tool, designer must choose a target FPGA chip beforehand. The output of the tools is made for specific integrated circuit and cannot be used with the other. That lack of good description of commercial applications is understandable. Real puzzle is, why the number of academic publications is so limited. Nonetheless there are few works worth mentioning.

Alliance is a complete set of tools [3, 4] for synthesis of VHDL code. First version was very limited. It could accept only a very small subset of VHDL statements, and it was impossible to mix in one input file different coding strategies (behavioural, data-flow, and structural) [5]. After some time a serious improvement has been noticed. Nowadays, many previous restrictions are no longer valid. The improvement has been achieved with help of a completely new application to the whole design toolbox. Its task is to convert any VHDL file to the form which can be accepted by the rest of programs [6]. As it comes to the documentation, a typical user manual is provided. There is no information about how the tools work.

Another approach uses Extended Timed Petri Net (ETPN) which is extension of Petri Net as an intermediate form [7— 9]. The method in its first step obtains a data-flow graph and an ETPN which serves as control logic for the design translated from VHDL code. Afterwards the ETPN is being optimized. Final step is to generate a structural VHDL representation which stands for an output of the tool. The whole process was implemented in a tool named CAMAD[10].

The third conception presents Mekenkamp [11,12]. It proposes to translate VHDL code into an intermediate form called SIL. Afterwards all necessary synthesis operations (e.g. loop unrolling) are done on the SIL [13] model. In the end, an optimized SIL is translated back into VHDL code.

As it comes to the Boolean equations and their usage as an output format, the Altera Corporation and its software MAX+PLUS II and Quartus II support it in a limited way. They can be seen in a floor plan editor window [14,15] and compilation report.

Translation from VHDL sources to Boolean equations. Boolean equations which were utilised in the project are very simple. There are only three logical operations used: and, or, not. A Boolean equation is created only if there are assignment statements inside process body. For each single bit of a signal or

variable there will be exactly one equation produced. The equation can be in one of the two forms:

simple, when it represents combinational logic,

complex, when it represents sequential logic.

In the second case, there are at least two equations. They stand for the base to build a flip-flop or a latch. The memory elements are constructed from predefined templates. Above rules are valid even if there are more than one assignment statements for a particular signal or a variable\footnote{As it comes to variables, there are exceptions}. Now it is time to discuss problems which may occur during translation process of VHDL sequential statements. Each instruction will be presented independently.

Assignments. When it comes to generating Boolean equations, VHDL statements can be divided into two groups:

those which provide new equations,

those which use previously generated equations.

The first group is very small and consists of only one type of statement: the assignments. So if there are no assignments in the process body, Boolean equations will not be produced at all because the rest of VHDL constructs rely on what assignments provide. There are two problems concerning assignments translation process:

assignment target identification,

significant differences between variables and signals.

The first job is to find out what a particular assignment statement is changing. Because of VHDL's complexity this task is quite difficult and operates on several levels which are: variable(signal), field or dimension (there can be several such levels), and in the end at bits level. Every level of identification is related to a specific VHDL concept. The variable level deals with aggregates mechanism, which is an ability to change value of more than one variable at a time. Next level concerns record and array (multidimensional as well) data types. Additional problem at this level is a slice mechanism. The last level reflects a fundamental feature of VHDL data entities. They are all in fact built from single bits.

The difference between variables and signals is another major problem. The value of a variable is changed immediately after an assignment. As it comes to a signal, it keeps its old value until the end of the process occurs. Another issue is that only the last assignment to a signal is valid, all the previous ones have to be discarded.

Variables represents memory. The main assumption is that every time a memory is written, it is done to a different address. As the result, a new variable is created (in semantic meaning), every time a value of the original one is changed.

Boolean equations which have been generated, are stored inside a data structure called Equation Container (EC). Apart from equations the structure holds additional semantic information. There is one such structure for each variable or signal. The EC was designed to carry all data types allowed by VHDL.

A process of generating Boolean equations for an assignment statement consists of several steps:

1. Generation of Boolean equation for the right side of the assignment. That part slightly exceeds the main topic of the article and is well presented in following publications: [16—18].

2. Identification of the target (or targets) of the assignment. It was partially presented above. The point is to find all signals (variables) and their bits which were affected by the particular assignment. The possible problems were covered earlier in the paper.

3. Creation and update of EC structure for each target. In this step the equations are stored inside EC structures along with additional semantic information. If it is the first assignment for the target, just a clean EC is created and filled with equations. Additionally for a variable, a special field is set indicating that it was written. Things work differently if there is an EC structure created before. For a signal the case is simple: new equations replace the old ones. A variable assignment requires a new EC structure to be created. It will contain all the newly generated equations, but not only them. If there are empty fields (bits) in it, they should be filled with equations from the old EC structure, of course, if it contains the lacking ones.

Example 1. Generating Boolean equation for a signal assignment:

```
entity test is
  port (
    a,b,c : in bit_vector(0 to 1);
     z: out bit_vector(0 to 1)
    );
end test;

architecture arch of test is
begin
  process
    begin
      z<=a;
      z<=b;
      z<=c;
    end process;
end arch;
— Equations
```

```
z(0)=(c(0));
z(1)=(c(1));
```

Example 2. Generating Boolean equations for a variable assignment:

```
entity test is
  port (
    a,b,c: in bit;
    z,z1: out bit
    );
end test;

architecture test5 of test is
begin
  process
    variable tmp: bit;
  begin
    tmp:=c or a;
    z<=tmp and b;
    tmp:=a;
    z1<=tmp and b;
  end process;
end test5;
— Equations
tmp_1_0=(c|a);
z=(tmp_1_0&b);
tmp=(a);
z1=(tmp&b);
```

If statement. An if statement allows conditional code execution. Only one branch can be activated at the time. If the activation condition of a particular branch is fulfilled, the code inside is executed and then control goes to the first statement after the if block. The main concept of a translation process is to join all the equations obtained from assignments (or other instructions) located inside if branches by those created from if's branches' conditions. There will be one equation for every different target. The whole procedure can be divided into following steps:

1. Generation of Boolean equations for if conditions. Equations which will be created must guarantee that only one branch could be activated.

2. Searching for equations inside each branch. Equations are stored inside the ECs. All the ECs inside a particular scope (process, if branch, case branch etc.) are linked together in order to form list.

3. Creating final equations. In this step a final equation is created, one for each distinct target. The second task is to determine a type of logic of the equation. There is a simple rule which helps with this problem. The equation will be of a combinational type if the target was assigned in every branch of the if statement. The if must be completed which means that else branch have to be presented as well. The equations are created accordingly to formula.

$$X = \sum_{i=1}^{n} We_i X_i, \tag{1}$$

where i — branch index; n — total number of branches; X — an assignment target; X_i — an equation for a right side of the assignment in i-branch; We_i — an entering condition for the i-branch.

A formula (1) is correct for combinational logic sources only. A sequential logic requires a latch to be created. In order to do so, two Boolean equations are generated accordingly to formulas

$$D = \sum_{i=1}^{n} We_i X_i Z_i, \tag{2}$$

$$C = \sum_{i=1}^{n} We_i Z_i, \tag{3}$$

where Z_i — a special variable which is set to value 1 if there is an assignment for a target X in the i-branch and to value 0 otherwise; D — data input; C — clock input. The formula (2) represents the data input of the latch, the formula (3) stands for the clock signal. The rest of the latch's logic is obtained purely by using of a template.

If a variable or signal was previously assigned (before it was done for the second time inside the if) it is sometimes possible to omit latch creation. In order to achieve this, it is enough to use the old value of a variable for all those branches which did not contain an assignment. Such approach is expressed by formula:

$$X = \sum_{i=1}^{n} We_i X_i Z_i + \sum (We_i \,!Z_i) Y_i, \tag{4}$$

where Y_i — the old value of the variable or signal (from previous assignment).

Entering conditions. In all equations presented so far there is an important (and unexplained yet) part called the entering condition. In general terms, it is a Boolean equation which when it is fulfilled activates a particular branch. There

is a one such equation for every branch and it is created from the original if statements' conditions

$$We_i = \left(\prod_{i=1}^{j<i} W_j \right) W_i, \tag{5}$$

where W_j — a Boolean equation obtained from the expressions placed between keywords if and then for all preceding branches; W_i — the same as the former, but for the current branch.

Case statement. A case statement allows conditional code execution as well as if, but each branch is treated independently. As the result there could be activated more than one branch at the time. Such a situation cannot be synthesised properly, therefore the original behaviour of the instruction must be altered slightly. It was achieved by creation of few constrains which limits possible cases' forms. If the particular case statement is planned for synthesis they have to be fulfilled. The requirements are as follows:

the control expression (the one after keyword case) must be of an integer or enumerate type (enumerate type array as well);

entering values must be static;

entering conditions set for each branch has to be separable form the others;

the size of entering value and the control expression must be the same (meaning a number of bits);

all possible values of the control expression} must be covered by entering condition (use of an others branch is allowed).

The translation process works quite the same as for an if statement (formulas (1) — (4). The dissimilarity concerns creation of entering conditions. it is done in a different way. First of all, the equations for entering values must be created:

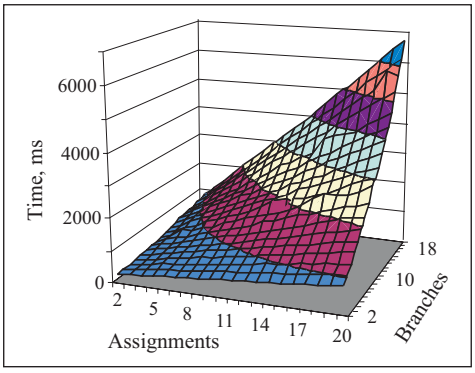
$$W_k = \prod_{l=0}^{p-1} \begin{cases} l < n \wedge W_{k,l} = 1 \Rightarrow S(l), \\ l < n \wedge W_{k,l} = 0 \Rightarrow !S(l), \\ l > n \wedge l < p \Rightarrow !S(l), \end{cases} \tag{6}$$

where W_k — an entering value; S — a name of the control expression; $S(l)$ — single bit of the control expression; n — number of bits of entering value; p — size (bits) of control expression; l — bit index.

Afterwards, they are used to create entering conditions:

$$We_i = \sum_{k=1}^{m} W_k. \tag{7}$$

During the last process it is important to check all the constrains presented above. Implementation of them is not as easy as it may look.



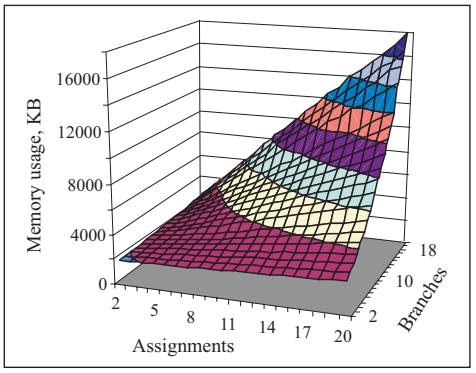


Fig. 1. Time of compilation of if statement in relation with a number of branches and assignments inside

Fig. 2. Memory usage during compilation of if statement

As it comes to the others branch it is sometimes more convenient to create entering condition using a different approach:

$$We_{\text{others}} = !\left(\sum_{i=1}^{n-1} We_i\right). \tag{8}$$

Instead of looking for unused entering values, previously created entering conditions are taken.

Variables and latches. Accordingly to what was said before, conditional execution of VHDL code may leads to sequential logic. It happens if a value of a signal must be remembered between consequent process activation. In case of variables there are certain situations when it can be avoided. Variables live only inside a body of a process, so they not affect outside environment directly [19]. Therefore previously presented restrictions can be altered a little. Speaking in a nutshell a latch is created if:

the variable fails for general rule for choosing type of logic;

its value is used in the body of process before its initialisation.

To put these rules in practice, it is necessary to analyse the whole process body.

Performance and computational complexity. Now it is time to verify the presented approach. First step will show its theoretical performance, expressed in terms of a computational complexity. Later the benchmark tests will be presented.

Estimation of computational complexity of a compiler is very difficult task. First of all, an input set is unlimited in a form, and size. Secondly, a compilation is compound and multi level process. It is nearly impossible to find a single distinct operation which can be used as a base of calculations. In other words, obtained value

of computational complexity cannot be precise. Nonetheless it should show enough information to get general picture of compiler performance.

Because of lack of better methods, such estimation of computational complexity was done for the algorithms presented above. As a basic operation, generation of single Boolean equation was chosen. The computation complexity for an assignment and if(case) statement are expressed by formulas: $O = n, O = n\,(k+1)$. The n is a total number of bits for which Boolean equations were generated. For an if(case) statement there is an assumption that every branch has the same amount of assignments (single bit).

As an example of real results, the Fig. 1 and 2 show how a time and memory usage vary during compilation of an if. Surprisingly, benchmarks tests examples which were used to check performance of implementation showed that theoretical deliberations were not far from reality. It can be seen clearly that correlation between time and number of assignments is linear, exactly as in the formula (1). For the two remaining instructions quite similar characteristics were obtained.

Table 1. **Real life test example**

Operation	Elements		Time, s	Memory, MB
	Gates	Flip-Flops		
Add	32929	32	13.83	6.45
Mul	98339	32	50.30	40.22
Div	128035	32	297.89	70.73

Table 2. **Real life test example — Xilinx ISE 6.2 for Spartan2E**

Operation	Size (gates)	Elements			Time, s	Memory, MB
		LUT	IOB	Flip-Flops		
Add	100k	626	97	32	11	68,43
Mul	100k	1072	97	32	14	72,66
Div	200k	2775	97	32	39	104,40

Table 3. **Real life test example — Altera Quartus II 4 for Cyclone**

Operation	Elements			Time, s	Memory, MB
	LUT	IOB	Flip-Flops		
Add	679	98	32	19	42,40
Mul	1172	98	32	24	43,50
Div	2733	98	32	59	45,00

Time of compilation and memory consumption are not the only measures which can be used to assess performance and usability of HDL compiler. Other parameters which can to be taken into consideration are:

size of generated integrated circuit (given as a number of gates for example);

power consumption;

the highest possible working clock.

For a purpose of studies presented in the paper the first parameter was taken into account and verified. In order to do that another test was performed this time using more complex VHDL source. The point was to check how the compiler behaves when it has to handle industry level test case. As a test subject floating-point arithmetical operations were chosen. They are reasonable complex examples but still able to implement without much of sequential logic which was very important as the article main topic is related to combinational logic. The Table 1 presents such information, Tables 2 and 3 give a snapshot of commercial tools efficiency. The commercial tools used for comparison purposes require to choose one of a few target architectures before a synthesis process can be started. Both tools were designed by companies which main aim is to produce FPGA chips. As the result the tools support only those FPGAs which belong to the same firm. So the comparison is not as accurate as it should be, but the main point is to show how a tool presented in the paper behaves, and this task has been realised well enough.

Conclusions and future works. Presented results show that tool based on knowledge presented in this article is not far behind typical commercial products. Considering the fact the compiler was in its beta version, and there are still a lot of things which could be better implemented, the obtained performance is quite promising. The results proved that Boolean equations can be used as an output format of a synthesis compiler.

Future works should concern possible performance improvement and addition of synthesis control options for a compiler. It would give designers better way to implement their projects.

VHDL – одна з найпопулярніших мов, які використовують у засобах логічного синтезу. Вона вміщує безліч операторів, що забезпечують її потужність та гнучкість, тому створення компілятора мови VHDL, орієнтованого на використання у логічному синтезі є складною задачею. Інформація про алгоритми трансляції, що надходить від розробників VHDL, є недостатньою для створення технічних засобів. Розроблено алгоритми для деяких послідовнісних операторів мови VHDL. Запропоноване рішення реалізовано у компіляторі, який використовує логічні рівняння як вихідний формат. Наведено результати тестів, виконаних для перевірки меж можливості практичного використання запропонованих алгоритмів.

1. *Clive* «Max» Maxfield.—Design Warrior's Guide to FPGAs. — Elsevier, 2004. — 560 p.
2. *IEEE* Standard VHDL Language Reference Manual.— IEEE Std 1076-1987. — IEEE Standards Board, 1991.

3. *Greiner A., Pęcheux F.* Alliance: A complete set of cad tools for teaching VLSI design.— 1992. — 8 p.
4. *Équipe* Achitecture des Systémes et Micro-Électronique.// Alliance: A Complete CAD System for VLSI Design. — Laboratoire MASI/CAO-VLSI, Institut de Programmation Université Pierre et Marie Curie (PARIS VI). — 2004. — 12 p.
5. *Équipe* Achitecture des Systémes et Micro-Électronique// Alliance documentation for version 3.2.— Laboratoire MASI/CAO-VLSI, Institut de Programmation Université Pierre et Marie Curie (PARIS VI). — 1992. — http://www-asim.lip6.fr/recherche/alliance/olddoc/.
6. *Equipe* Achitecture des Systémes et Micro-Électronique// Alliance documentation for version 5.0. — Laboratoire MASI/CAO-VLSI, Institut de Programmation Université Pierre et Marie Curie (PARIS VI).— http://www-asim.lip6.fr/recherche/alliance/doc/.
7. *Éles P., Kuchcinski K., Peng Z., Minea M.* Compiling VHDL into a high-level synthesis design representation// EURO-DAC '92: Proc. of the conference on European design automation. — Los Alamitos, CA, USA, 1992. — P. 604—609.
8. *Éles P., Minęa M., Kuchcinski K., Peng Z.* Synthesis of VHDLl concurrent processes// EURO-DAC '94: Proc. of the conference on European design automation. — Los Alamitos, CA, USA. — 1994. — P. 540—545.
9. *Éles P., Kuchcinski K., Peng Z.* Synthesis of systems specified as interacting VHDLl processes. — Integr. VLSI J, 1996. — P. 113—138.
10. *Peng Z.* Synthesis of VLSI systems with the CAMAD design aid// DAC '86: Proc. of the 23rd ACM/IEEE conference on Design Automation. — Piscataway, NJ, USA. — 1986. — P. 278—284.
11. *Mekenkamp G. E.* A New Approach to VHDL-Based Synthesis// PhD thesis University of Twente, January 1998. — 150 p.
12. *Molenkamp B. E., Hofstede J., Krol T. , Mekenkamp G. E., Middelhoek P. F. A.* A syntax based VHDL to CDFG translation model for high-level synthesis// VIUF Proc. Spring 1996, February 1996. — P. 89—97.
13. *Molenkamp E., Mekenkamp G. E., Hofstede J., Krol T.* Sil: an intermediate for syntax based VHDL synthesis// VIUF Proc., April 1995. — P. 5.1—5.9.
14. *MAX+PLUS II* Getting Started Manual.— 8.1 edition. Altera Corporation, 1997.— P. 114—116.
15. *Quartus II* Version 5.0 Handbook.— Altera Corporation.— www.altera.com, 2005.
16. *Bielecki W.* Kompilator języka VHDL do projektowania układòw logicznych.— Pracownia Poligraficzna Wydziału Informatyki Politechniki Szczecińskiej, 2002. — P. 1—12.
17. *Liersz M. K.* Algorytm generowania równań boolowskich dla instrukcji przypisania zawierającej odwołania do tablic w języku VHDL//Kompilator języka VHDL do projektowania układów logicznych. — Wydział Informatyki Politechniki Szczecińskiej, 2002. — P. 67—74.
18. *Moscicki M.* Generowanie równań boolowskich dla funkcji i procedur języka VHDL// Kompilator języka VHDL do projektowania układów logicznych.—Wydział Informatyki Politechniki Szczecińskiej, 2002. — P. 123—132.
19. *Bhasker J.* A VHDL Synthesis Primer — Second Edition. — Star Galaxy Publishing, 1998. — 320 p.

Поступила 17.12.07