Methods and algorithms for a high-level synthesis of the very-large-scale integration

OLEG NEPOMNYASHCHY [1], ALEXANDR LEGALOV [1], VALERY TYAPKIN [1], IGOR RYZHENKO [1], VLADIMIR SHAYDUROV [1,2]
[1]Siberian Federal University
660041, Krasnoyarsk, Prospect Svobodny, 79
RUSSIAN FEDERATION
[2]Institute of Computational Modeling of Siberian Branch of the Russian Academy of Sciences
660036, Krasnoyarsk, Akademgorodok, 50/44
RUSSIAN FEDERATION
2955005@gmail.com, legalov@mail.ru, tyapkin@mail.ru, rodgi.krs@gmail.com, http://www.sfu-kras.ru, shaidurov04@mail.ru, http://icm.krasn.ru

Abstract: – We develop methods and algorithms for a high-level synthesis and a formal verification of the architecture for very-large-scale integration (VLSI). The proposed approach is based on the functional-flow paradigm of parallel computing and enables one to perform architecture-independent VLSI synthesis by the construction of a computing model in the form of intermediate structures of control and data graphs. This approach also provides an opportunity to verify a design at the formal description stage before the synthesis of the register-gate representation. Algorithms and methods are developed for the construction and optimization of an intermediate representation of a computing model, the verification, and going to the register-gate description of VLSI. The stages of the high-level VLSI synthesis are formed in the context of the proposed technique. An example of the synthesis of a typical module is considered for a digital signal processing. Results of the practical modeling are presented for an example.

Key-Words: – Parallel computing, functional programming, high-level synthesis, formal verification.

1 Introduction

Due to constantly increasing complexity of *the very-large-scale integration* (VLSI) architecture, *high-level synthesis* (HLS) is the main area in development of design techniques [1]. Modern routes of HLS are based on the imperative paradigm and the related programming languages. Since the parallel data-flow processing is typical for digital VLSI, it is necessary to involve it into synthesis. However, the use of sequential imperative languages for the description of VLSI considerably complicates recognizing parallel structures in programs. Besides, with known approaches, the register-gate representation of a circuit obtained by HLS is hardware-dependent, i.e., it is initially oriented to a specific chip or to the family of chips of a specific manufacturer.

Nowadays, a number of procedures is available for distinguishing parallel structures in imperative programs [2]. Since the problem is not trivial, heuristic methods are used to obtain a solution in each case. In so doing, looking for optimal solution under given restrictions is often inefficient for a complex project. Once a project has been completed, it turns out to be obsolete.

Modern VLSIs possess natural parallelism. Hence, efficient solutions can be found with the use of paradigms of functional programming since functions can be carried out in parallel and independently from each other. In its turn, *the functional-flow* (FF) parallel approach enables one to avoid searching and processing of parallel structures since an FF parallel program implies the initial description of a problem with maximal parallelism.

The implementation of a functional approach for development of digital circuits is presented currently for languages Lava, Hume, F#(Kiwi), and Erlang [4, 8 – 10].

Language Hume is developed for embedded real-time systems and consists of two levels: the expression layer and the coordination one. The expression layer describes the work of compute nodes (boxes); and the coordination one does the interaction between the sites in terms of finite-state machine. The execution of this language to *the field-programmable gate array* (FPGA) and the realization of its interpretation on the base of software cores PowerPC and MicroBlaze are considered in [10].

The project Kiwi is initially supposed to support a translation from the language F# into the languages for hardware description. In this project, the support of parallelism in the language is partial and oriented to standard streams of the platform .NET. Aim of the project was initially an attempt to combine the development of hardware and software in one platform and language. During translation into description at *the register transfer level* (RTL), streams are translated into parallel-working units. However, in this approach, there is no built-in support for parallelism in the language as well as in imperative ones.

In the language Lava [8] on the basis of Haskell, the main emphasis is on the level of topology and arrangement of elements on the chip. Support for parallelism in Lava is also realized like in the original language Haskell.

The approach [9] with the language Erlang for transfer of processes into the hardware description language converts them into separate computing modules that form a network-on-chip. An advantage of Erlang consists in the built-in support for parallelism in comparison with other functional languages discussed above.

Nevertheless, most of the works in this field is not widespread. Also known languages and methods have such a common and significant drawback as the lack of support for massive parallelism at level of language and calculation model.

We propose a technique for the architecture-independent VLSI synthesis [3] based on the FF parallel paradigm. It allows one to perform the efficient transformations of high-level representation and to carry out the formal verification and optimization of the VLSI architecture at initial stages of synthesis. This results in efficient representation of complex functional VLSIs at high abstraction levels, testing with maximal covering, and development of optimal solutions in the VLSI architecture on the gate level.

2 Justification and description of the method

In the general case, the high-level VLSI synthesis involves the following stages:

1. translation of a high-level description into an intermediate representation;
2. operation scheduling;
3. resource allocation;
4. synthesis of a data processing scheme;
5. synthesis of a control scheme.

In conventional HLS routes with an imperative language in the initial description at the translation stage, *the data-flow graph* (DFG) and *the control-flow graph* (CFG) are recognized; or an integrated version of *the control-data-flow graph* (CDFG) is formed. The scheduling stage consists in time distribution of operations excluding a data availability conflict. At the resource allocation stage, for each operation the available hardware resources (computing units) are allocated. At the final stage, the control scheme is generated for computing units relative to data availability.

Each of the stages 2 and 3 is an NP-complete computational problem and is interrelated to subsequent stages. Hence, with using an imperative description language, the available methods for the optimal solution of the problem are computationally unfeasible or non-optimal. Usually one or several parameters (for instance, performance, propagation delay, power consumption, or the area of chip) are given as restrictions for synthesis and considered to be a metric of optimality for the solution of a synthesis problem. These parameters are interrelated and often mutually exclusive which also complicates the search of an optimal solution. With this approach, the synthesis results in several versions of a solution. Each of them is related to imposed restrictions. Thus, the choice of an optimal solution is rather nontrivial. The problem of high-level VLSI synthesis with the choice of optimal solution under given restrictions is known in literature as *the design space exploration* (DSE).

Contrary to conventional methods, the proposed high-level synthesis technique is based on the functional approach and enables one to simplify the solution of such problems. In the context of this technique, initial algorithms of the operation of VLSI are represented in the developed functional-flow programming language. This allows one to form the architecture-independent representation of a single-chip system in the form of programs with large-scale parallelism. With the functional-flow approach, the compilation of such a program results in an intermediate representation in the form of DFG and CFG. With their help, a set of versions of architecture designs for VLSI is obtained with given parallelism degree. In Fig. 1 a block diagram of a high-level design route is shown for conventional and functional-flow synthesis techniques.

The DFG/CFG recognition for the conventional approach is similar to the compilation into an intermediate representation for the functional-flow approach. However, formats of DFG and CFG representations are different. Moreover, compilation from one into another representation is admissible, for example, for compatibility of hardware tools used at this stage.

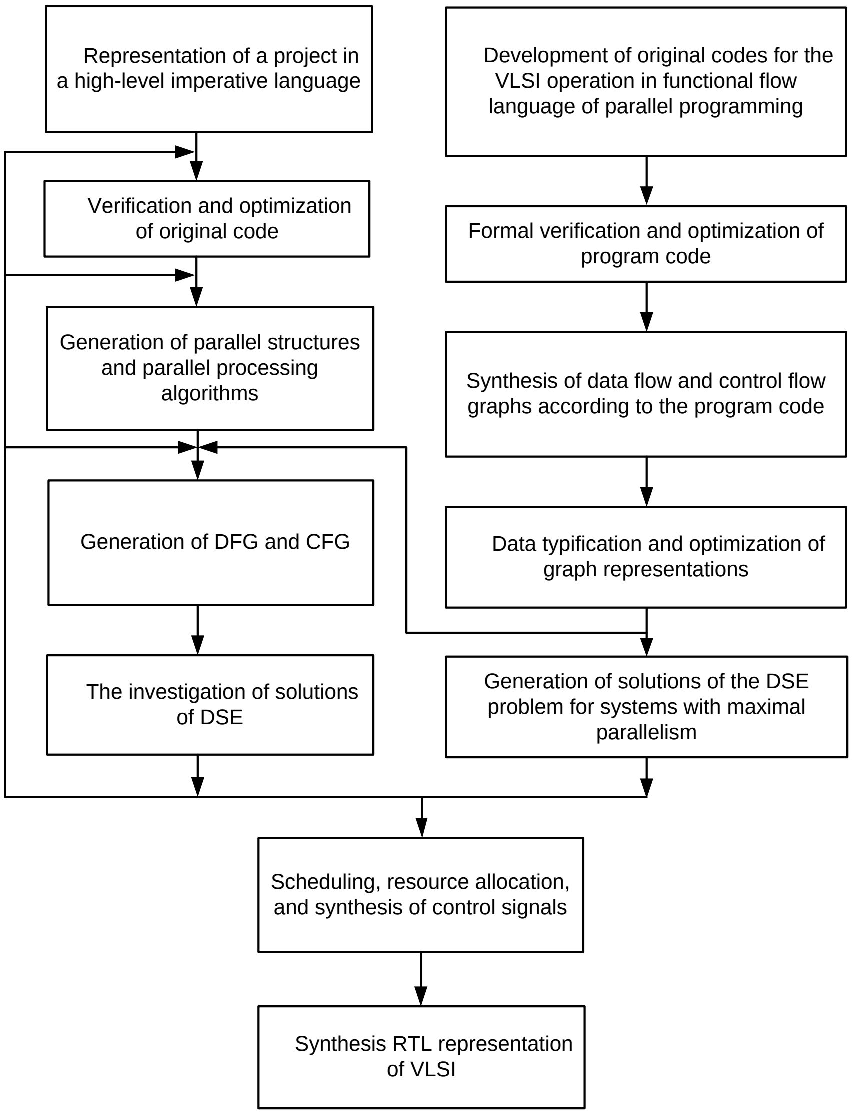
Representation of a project in a high-level imperative language
Verification and optimization of original code
Generation of parallel structures and parallel processing algorithms
Generation of DFG and CFG
The investigation of solutions of DSE
Development of original codes for the VLSI operation in functional flow language of parallel programming
Formal verification and optimization of program code
Synthesis of data flow and control flow graphs according to the program code
Data typification and optimization of graph representations
Generation of solutions of the DSE problem for systems with maximal parallelism
Scheduling, resource allocation, and synthesis of control signals
Synthesis RTL representation of VLSI

Fig. 1. High-level synthesis routes for the conventional and functional-flow approaches

With the conventional approach, DFG is obtained in the form of the couple G = (V, E) where V is the set of vertices that have input and output data ports; and E is the set of arcs that connect these ports. Semantics of the graph functioning is reduced to the data processing from input ports and transmission to output ones with the signal of data availability. When constructing DFG by the imperative description, the transformation is performed into the single assignment form. Vertices of the graph can involve any operations and functions that are independent of conditions.

CFG involves control dependences between base blocks. The dependences express the conditions for a base block to operate. Here base block is meant that it is unconditionally performed completely from start to finish. The vertices of CFG involve conditions whose computing generates the condition for a vertex of DFG to come into action.

According to the developed route of the high-level VLSI synthesis on the base of the FF approach, the following design stages can be recognized.

– The development of algorithms for the VLSI operation in the FF programming language. At this stage, the high-level architecture-independent representation of original algorithms is performed in the FF programming language.

– The formal verification and optimization of program code. According to the design task, the debugging of algorithms for the VLSI operation is performed without binding to a target platform.

– The synthesis of the intermediate VLSI representation in the form of DFG and CFG. First the synthesis and optimization of DFG are performed. Then the generation of the intermediate representation of argument and constant types is executed according to the developed program code with given restrictions. At the same time, the synthesis and optimization of CFG are performed.

– The data typification and the optimization of graph representations. The data typification is performed according to the type specifications generated at the previous level. The optimization of DFG and CFG is performed.

– The intermediate DFG/CFG synthesis. This stage is performed provided that it is necessary to go to the related stage of the conventional HLS or to make concurrent loops for the conventional and FF parallel synthesis.

– The solution of the DSE problem. These solutions are generated for systems with maximal parallelism. This provides the maximal covering of the space of solutions under given restrictions and the automatic search of an optimal solution.

– The scheduling, the resource allocation, and the synthesis of control signals.

– The synthesis of the register-gate representation of VLSI.

Consider the main stages of the developed route. As an example, we take the multiplication of complex numbers being a popular operation for digital signal-processing systems.

The development of algorithms for the VLSI operation in the FF programming language

Below we present the listing for the function in the developed FF language.

```
// multiplication of complex numbers
ComplexNumsMult << funcdef params
{
num1 << params:1;  //the first complex number
num2 << params:2;  //the second complex number
a << num1:1; // real part
b << num1:2; // imaginary part
c << num2:1; // real part
d << num2:2; // imaginary part
return << ( ((a, c):*, (b, d):*): –, ((a, d):*, (b, c):*):+);
}
```

The formal verification and optimization of program code

The use of the FF programming language provides parallelism on the operation level. The absence of other parallelism levels enables one to simplify the verification process because the analysis of additional resource conflicts is not required here like that in conventional architectures.

The set of axioms for base functions of the language enables one to use them further for the analysis of correctness of FF programs. To this end, axioms are successively applied to base operators and then the “convolution” is formed by the described rules.

The formal verification means a proof of program correctness. It consists in establishing correspondence between a program and its specification. The main advantage of the formal verification is in possibility to prove the absence of errors in a program; whereas the testing just enables one to find errors. Moreover, the formal verification suggests the analytical studies of properties of a program on the basis of its code. The purpose of verification is achieved by a rigorous mathematical proof of the correspondence between a program and its specification.

Contrary to imperative languages used in the conventional approach, the FF programming language involves some specific structures that allow one to develop efficiently architecture-independent applications. These structures involve:

– parallel lists that allows one to store data and function sets whose interpretation is performed in the parallel way;

– delayed lists for the storage of code fragments which start to run once the expansion operation is applied to a list.

Moreover, these lists can be transferred as an argument of a function.

In this case, the data typification is not used at high hierarchy levels of the VLSI description. The availability of specific structures of the language and the absence of data typification provide complete architecture independence of a design.

However, when going to the register-gate level at subsequent stages of the synthesis, the strict typification is required. To this end, at the compilation stage the corresponding structures are involved in the language; among them digital (integer and float) types with specification of word size with an arbitrary bit number and specification of dimension and type of an argument of a function. This is necessary since the absence of typification at low stages of the high-level synthesis gives no way of the calculation of constant expressions at the stage of optimization because a word size can depend on the target architecture. In addition, the VLSI architecture requires the explicit specification of a word size up to bit width. Hence, expressions involving the constant calculations admit optimization. For example, for the expression (3, 4, 5, 6):((1, 2): +) <=> 5 that involves only constants, the number 1 + 2 = 3 of the element of the list is calculated and the third element (the constant 5) is derived.

Along with the specification of the argument type for scalar quantities, at the compilation stage the dimension of lists is specified. This provides the possibility to transform parallel lists and to transform recursions into loops at the next compilation stage.

When compiling, those structures of the FF language are transformed which were not transformed immediately in VLSI elements and nodes. Such a transformation provides the further synthesis in the FF language as well as the use of elements of conventional HLS routes. Thus, it becomes possible to use available algorithms for the solution of the DSE problem as well as to develop hybrid (FF and conventional) ones. The solution of the problem is reduced to the development of an efficient algorithm for going from a system with large-scale parallelism to VLSI architecture for a target platform or a chip.

In the compilation process, some structures of the language (for instance, delayed and parallel lists) may not be transformed immediately into nodes and elements of a chip while other ones are subject to some restrictions. For example, the synthesized program may not involve delayed lists as well as parallel lists and lists of data whose size is unknown before execution. In addition, with the VLSI platform the implementation of recursive functions is complicated or impossible. In this context, when compiling functions for the developed VLSI platform, the following transformations are performed.

a). According to the axioms of the language, the delayed lists are transformed in parallel ones.

For example, for the delayed list of arguments consisting of three elements x, y, and z for functions f and g, the transformation operation is performed. In the language mnemonics this has the following form:

{x, y, z} : {f, g} <=> [x, y, z] : [f, g].

In this case the dimension of the delayed lists can be always determined at given stage. Hence, the dimension of the obtained parallel lists is determined as well.

b). The parallel list interpretation is performed. For example, for the parallel list of arguments x, y, and z for functions f and g, the interpretation results in the language mnemonics looks as follows:

[x, y, z] : [f, g] <=> [x:f, y:f, z:f, x:g, y:g, z:g]
[a, [b, c], d] <=> [a, b, c, d]
[x, y, z] : () <=> ([x, y, z]) <=> (x, y, z).

These transformations are admissible provided that the dimension of parallel lists is determined;

c). Tail recursion is replaced by a loop [7].

The above transformations of a program are used for its optimization. Interpretation of parallel lists simplifies the functioning of operators since the optimization results in elimination of the parallel lists whose processing requires a calculation of the number of incoming signals. Moreover, if restrictions are imposed with the architecture of the target chip, these transformations can be used more efficiently. In particular, the parallel lists can be completely eliminated which simplifies operators.

Practical implementation of algorithms is presented in [2] for opening some versions of parallel lists.

Synthesis of the intermediate VLSI representation in the form of DFG and CFG

The DFG generated at the FF program compilation stage is the VLSI description in the FF language represented in the form of intermediate format. The difference in the concept of the main and intermediate format representation for an original program is in the description of links. In program code the description is carried out “top-down” whereas in the intermediate “graph” representation it is performed in the reverse order from a function to arguments. In Fig. 2 the data flow graph is shown for the program for multiplication of complex numbers.

In Fig. 2 the following symbols are used:
: means a vertex of an interpretation operation;
(----) denotes a vertex of a data list;
⯃ denotes a constant;
▭ denotes an argument of a function.

Once the DFG is generated, the typification is performed, i.e., all types of arguments and constants are defined. For the above example, the dimension of all arguments (complex numbers) is the same and is defined as integer of certain length. After the typification stage the argument consisting of two lists of integers of dimension 2 is put at vertex 0. For this argument, input ports (denoted by a, b, c, and d) are generated. Then the interpretation operations at the vertices from 1 to 6 are calculated since the values of functions and data for these vertices are known at the synthesis stage.

As a result, the graph is reduced and the vertices from 3 to 6 involve the corresponding values of the parts of a complex number in the form of direct links with the ports of an input argument. In Fig. 3 the data flow graph is presented after the typification stage for the given example.

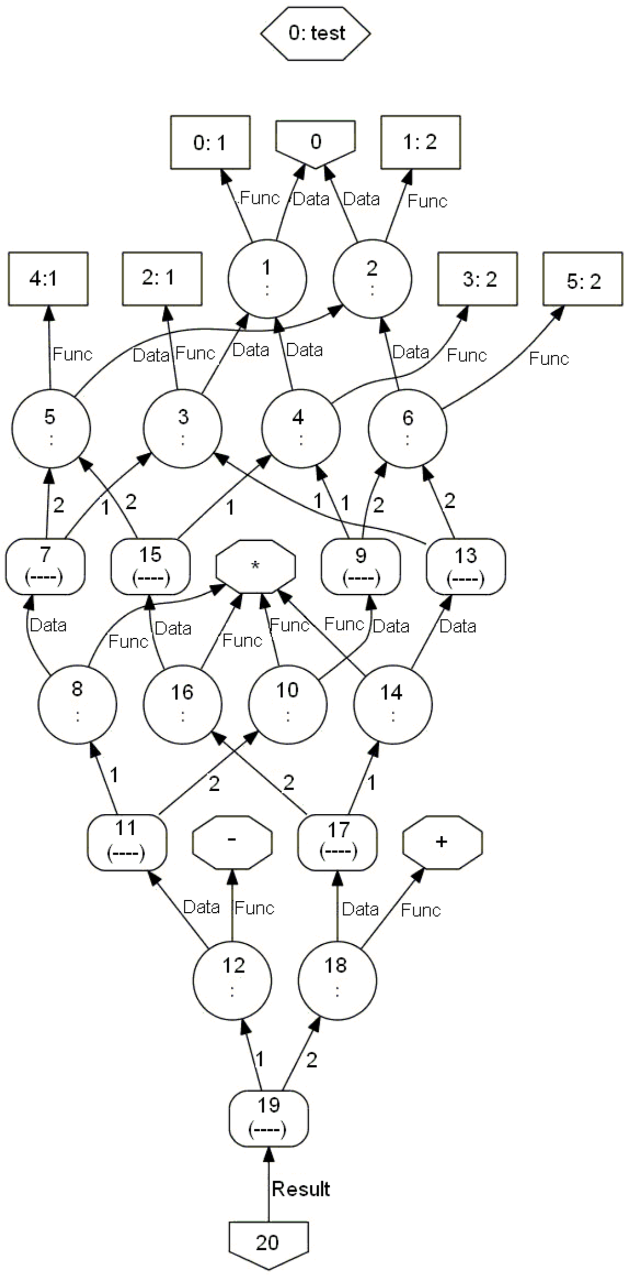

Fig. 2. The DFG for the program for the calculation of the product of complex numbers

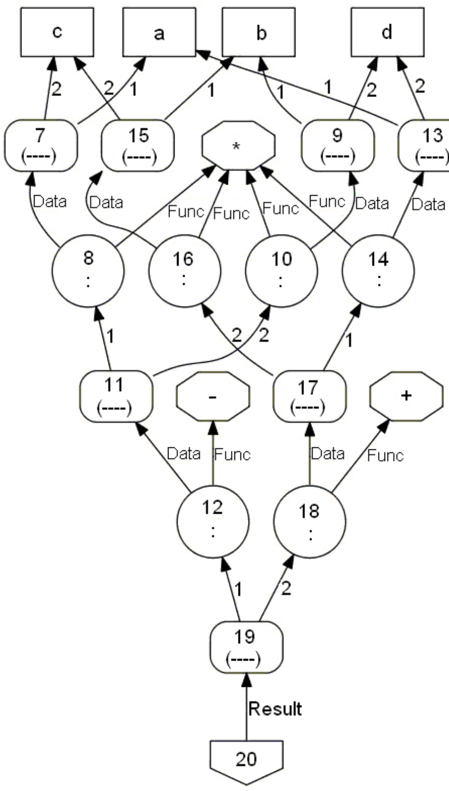

Fig. 3. The DFG of the program after optimization

CFG synthesis

At the next stage, the studies of the space of the DSE solutions are performed with the use of the transformed graph and given resource restrictions. Fig. 4 shows that 4 parallel multiplications and 2 parallel additions / subtractions remain in the graph. Assume that the resource restrictions are given as 2 multiplications / additions / subtractions per step.

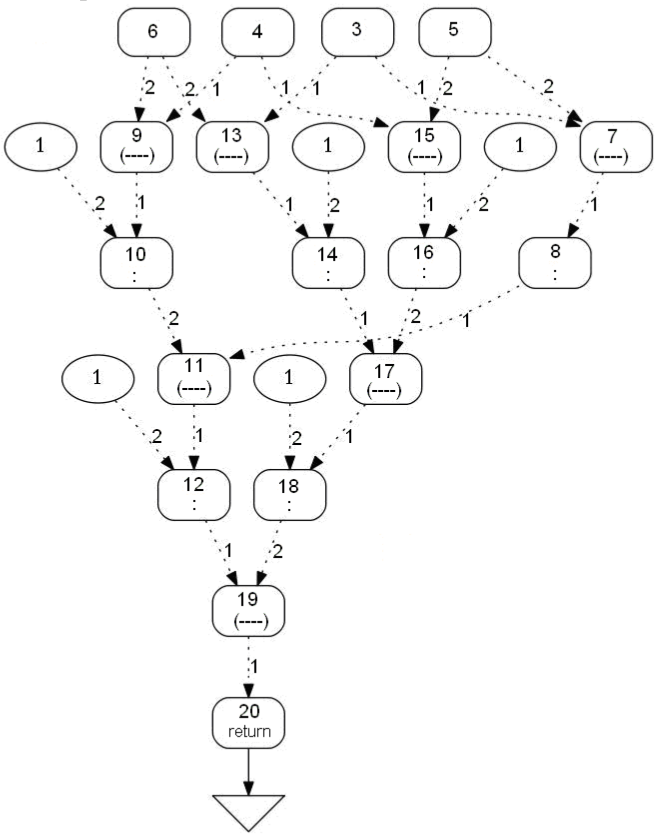

Fig. 4. The CFG after optimization

Then the data processing scheme is generated to be three-step, namely, 2 multiplications at each of the first and second steps and addition / subtraction at the third step. For the output vertex 20, an output port involving 2 elements is synthesized.

Synthesis of the control scheme is implemented on the basis of CFG. When generating it, the same vertices are eliminated as those from DFG at the optimization stage. CFG for the example is used for the control scheme synthesis (Fig. 4).

We use the following symbols:

: means a vertex of an automaton of an interpretation operation;

(----) is a vertex of an automaton of a data list;

⬭ is a constant ready signal;

▭ is an input port, a data-ready signal.

The vertices 3, 4 and 5, 6 are input ports for data-ready signals. The vertices 9, 7, 15, 13, 11, 17, and 19 are control vertices of the data lists. For these vertices, automata are synthesized in the form of a counter that outputs a ready signal with coming input signals equal to the dimension of a list. Here the size of these automata equals 2. For the vertices 8, 10, 14, 16, 12, and 18 of the interpretation operation, the Boolean "AND" scheme is synthesized. The vertices 3, 4, 5, and 6 are couplers of input data-ready signals. The vertices marked by 1 denote constant ready signal corresponding to a function of interpretation operator. Thus, at the synthesis stage, the "AND" scheme is transformed in a ready signal.

Control scheme synthesis

With the FF synthesis, the vertices of CFG correspond finite automata that take and process data-ready signals. Contrary to the conventional approach where some conditions are calculated at a vertex of CFG, here this vertex works with data-ready signals rather than with data and conditions calculated from them. According to [3], in the original control flow graph automata of the following types can be recognized:

– automaton of a constant;
– automaton of a data list;
– automaton of a parallel list;
– automaton of the interpretation operator;
– automaton of return of the result of a function.

At the optimization stage, the automaton of a constant is replaced by a constant data-ready signal. The automaton of result return is an output port for data-ready signal output. With totally opening parallel lists at the optimization stage, the automaton of parallel lists is not required.

The automata of a data list and the interpretation operator are implemented in the form of finite automata. The automaton of a data list generates the ready signal provided that the number of data-ready signals at its input equals the number of elements of the list. If the dimension of a list is known at the compilation stage, the automaton is implemented in the form of a counter that releases the ready signal as the overflow signal comes. Another version of the implementation of this automaton is the scheme of the Boolean "AND" and the next register.

At the optimization stage, parallel lists are converted into single operations. Hence, the automaton of the interpretation operator always has one data-ready signal and one function-ready signal at its input. Such an automaton is the Boolean "AND" scheme which combines input data- and function-ready signals.

The synthesized control scheme for the example is shown in Fig. 5.

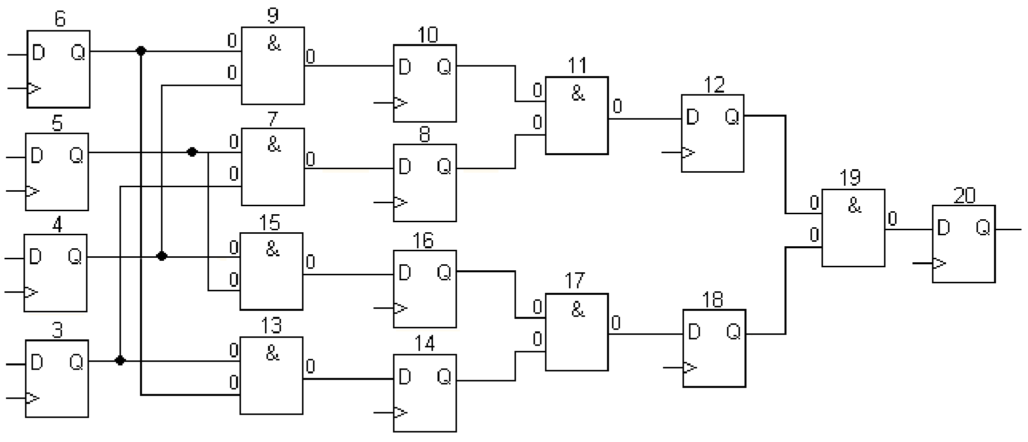

Fig. 5. Synthesized control scheme

The numbers on the scheme (Fig. 5) denote the relations between elements of the scheme and vertices of the optimized CFG. Ready signals for the data processing scheme are outputted from the corresponding vertices (3 – 6, 8, 10, 14, 16, 12, and 18). The output ready signal is generated in register 20 and the input ready signals are recorded in registers 3 – 6.

3 Conclusion

Nowadays one of the main problems in the development of modern VLSI technologies is the gap between the number of logic elements on a single base plate and the number of elements that can be actually designed and verified during an economically expedient period. Despite considerable progress in the conventional VLSI design and in promising directions of the high-level architecture design, there remains a number of problems to be solved in development of systems with parallel data processing and configurable architecture. At the present time, efficient techniques for the development of the VLSI architecture design are available only with binding to a target platform as well as methods and tools for the design support and formal high-level verification of the VLSI architecture design. With rare exception, programming languages (that are used in the present state-of-art for one-chip parallel data processing systems) are intended either for circuit description or for conventional programming.

Contrary to available methods and techniques for the high-level synthesis at the entire system level, the proposed technique for the architecture-independent synthesis enables one to work in terms of the principles of the system organization of computational process with the implementation of the obtained model on a target chip rather than in terms of available hardware or blocks for a one-chip system.

The use of the FF model of calculations, parallelism support at operation level, and the parallel flow model at all stages of the high-level VLSI design provide a qualitatively new level in the one-chip system design.

The presented results of the principles of going from the description in the FF language to the RLT description means that the main problems of the high-level VLSI synthesis may be solved on the base of FF approach.

The work was financially supported by the Russian Ministry of Education (Contract No. 14.578.21.0021 – RFMEFI57814X0021).

References:

[1] S.M. Logesh and D.S. Ram, A Survey of High-Level Synthesis Techniques for Area, Delay and Power Optimization, *International Journal of Computer Applications*, Vol. 32, No. 10, 2011.

[2] A. Mycroft and R. Sharp, Hardware/software co-design using functional languages, *Proceedings of Conference "Tools and Algorithms for Construction and Analysis of Systems"*, 2001, pp. 236–251.

[3] O.V. Nepomnyashchy, A.I. Legalov, and N.J. Sirotinina, High-Level Design Flows for VLSI Circuit, *Journal of Siberian Federal University. Engineering & Technologies*, Vol. 7, No. 6, 2014, pp. 674–684.

[4] J. O'Donnell, Generating Netlists from Executable Circuit Specifications in a Pure Functional Language, *Proceedings of Workshops in Computing*, 1993, pp. 178–194.

[5] A.I. Legalov, O.V. Nepomnyashchy, I.V. Matkovsky, and M.S. Kropacheva, Tail Recursion Transformation in Functional Dataflow Parallel Programs, *Automatic Control and Computer Sciences*, Vol. 47, No. 7, 2013, pp. 366–372.

[6] R. Namballa, N. Ranganathan, and A. Ejnioui, Control and data flow graph extraction for high-level synthesis, *Proceedings on VLSI of IEEE Computer society*, Vol. 1, 2004, pp. 187–192.

[7] D.D. Gajski, M. Meredith, A. Takach, and P. Coussy, An Introduction to High-Level Synthesis, *IEEE Design & Test of Computers*, Vol. 26, Issue 4, 2009, pp. 8–17.

[8] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, Lava: Hardware design in Haskell, *Proceedings of International Conference on Functional Programming*, 1998, pp. 174–184.

[9] P. Ferreira, C. Ferreira, and C. Alves, Erlang inspired Hardware, *Proceedings of International Conference on Field Programmable Logic and Applications*, 2010, Vol.1, pp. 244–246.

[10] A. Al Zain, W. Vanderbauwhede, and G. Michaelson, Hume to FPGA, *Proceedings of 10th International Symposium on Trends in Functional Programming*, 2010.

[11] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, Vol. 16, No. 2, 2006, pp. 157–196,

[12] V. Skylarov and I. Skilarova, FPGA-based implementation and comparison of recursive and iterative algorithms, *Proceedings of International Conference on Field Programmable Logic and Applications*, 2005, pp. 235–240.