

DFC: A Dataflow C language

Ivan Grigorov^{1,2}

¹Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS)

²National Research University – Higher School of Economics (HSE)

Email: grigorovia@ispras.ru

Abstract—Electronic designs are developed using Hardware Description Languages (HDL), such as Verilog and VHDL. These languages describe electronic designs at the Register Transfer Level (RTL). As electronic designs increase in both size and complexity the task of developing them using HDL becomes difficult. There comes the concept of High-Level Synthesis (HLS). HLS tools translate programs described at the behavioral level to the RTL. Most of them use imperative languages, such as C/C++, as their input languages. However, these languages are based on von Neumann model which does not describe parallelism. This usually results in a poor quality of the resulting electronic designs. So the use of the language with a model which describes parallelism is preferable. One of such models is the dataflow model.

In this paper we present DFC, a language that uses synchronous dataflow to explicitly express parallelism in programs while remaining time-agnostic. DFC supports the usage of configurable IP cores. DFC also accepts different optimization criteria which help developers to control synthesis process.

Keywords—HLS, Dataflow, IP core, IP reuse, IP-XACT, RTL library, Verilog, Dataflow languages.

I. INTRODUCTION

Hardware Description Languages (HDL) have been used to develop electronic designs since 1980s. Two of these languages have become de-facto standards for developing electronic designs, namely Verilog [1] and VHDL [2]. These languages describe the desired functionality at the Register Transfer Level (RTL). This level has to deal with low-level details such as timing constraints, explicit register allocation, etc. Designing circuits at this level is a tedious and an error-prone task.

High-level synthesis (HLS) addresses this issue by translating a high-level behavioral description of a program to RTL. Most HLS tools accept programs written in imperative languages. For example, Vitis HLS [3] translates a C/C++ program to either a Verilog or a VHDL program; Bambu [4] and LegUp [5] translate a C program to Verilog; Xilinx' AccelDSP [6] uses Matlab as an input language and Verilog or VHDL as an output language. The use of these languages supports widespread adoption of HLS. However, the results show that the designs, synthesized by these tools, mostly exhibit poor quality compared to the corresponding hand-written implementations. One of the reasons for this is that von Neumann model, which underlies these imperative languages, does not describe parallelism. Other tools use the input languages with underlying models which are more suitable for describing parallelism. For example, MaxCompiler [7] uses Java-like language MaxJ and XLS [8] uses Rust-like language DSLX, which follow the synchronous dataflow model [9]. These tools produce RTL models, comparable with the hand-written RTL models in terms of performance.

The other important issue in hardware design is component reuse. Nowadays electronic designs need to implement complex logic. To implement this complex logic from scratch requires a lot of time. The idea is to decompose this complex logic into smaller parts and construct the design using existing solutions for the smaller parts. These existing solutions are called Intellectual Property cores (IP cores).

This concept is called IP reuse. If IP core has an open implementation, then the integration is straightforward. However, due to the intellectual property issue sometimes it is not possible to get an open implementation. Unfortunately, existing HLS tools that follow the dataflow paradigm either do not have the support of external IP cores [10] or support only static (non-parameterized) external IP cores with open implementations [11].

In this paper we introduce DFC, a language that follows the paradigm of synchronous dataflow. DFC has a C-like syntax which eases the adoption of the language by the developers. DFC has an internal library of configurable standard components such as adders, multipliers, multiplexers, etc. which are tailored in an optimization step to achieve better quality of the design. DFC supports the use of external HDL libraries of configurable IP cores which are described using IP-XACT [12] standard. It also gives the developers an ability to control the design synthesis by applying various optimization criteria. DFC is a part of the bigger project Utopia, developed in Institute for System Programming of the Russian Academy of Sciences (ISPRAS).

II. BACKGROUND

The purpose of HLS is to allow programmers without the knowledge and experience in the field of hardware design to construct RTL models. HLS usually takes the following steps. First, a behavioral description is transformed into some formal representation that explicitly exhibits intrinsic parallelism of the description. In the following step allocation of available resources (functional units, memory, etc.) is performed. Next, the operations are scheduled into clock cycles. Finally, resource binding assigns operations and data elements to resources, allocated in the previous step. It should be noted that HLS also performs the interface synthesis.

One approach in HLS is to use one of imperative languages, for example, C/C++ [13][14][15]. Most programmers are used to imperative languages, which are based on von Neumann model. Therefore, the programmers do not have to learn new languages and, moreover, they don't need to develop a new way of thinking in terms of unfamiliar models.

However, von Neumann model, which underlies these languages, does not describe parallelism. This results in performance loss of these RTL models when compared to the hand-written implementations. So HLS tools use facilities to orchestrate parallelism in computations, for example, pragmas. Still, the imperative nature of these languages still plays a major part in the performance loss.

Other tools use languages with computation model which describe parallelism in some form [16][17][18]. One of these models is the dataflow model. The dataflow model describes a program in a form of a graph. Nodes represent operations on data. The operations themselves can be described as dataflow graphs so the model is hierarchical. Arcs represent data dependencies between operations. Data are carried on tokens which travel along the arcs. Dataflow model operates on potentially limitless streams of data. Nodes can execute in parallel assuming there are no data

dependencies between them. A node can execute (or fire) when there are enough tokens on the inputs. A dataflow model is said to be synchronous if the number of consumed and produced tokens is constant for each firing in every node. In the cycle-static model the number of tokens consumed and produced changes periodically. Nodes in the dynamic dataflow model consume and produce various numbers of tokens depending on the input data [19][20]. Since the firing rules are static in synchronous dataflow, the scheduling can be done at compile time. This means that complex scheduler logic will not be implemented in hardware resulting in more productive schemes compared to other types of dataflow.

III. RELATED WORK

In recent years various HLS tools have emerged, both free and commercial. In this section we review HLS tools which use languages based on dataflow models.

MaxCompiler: MaxCompiler [21] is a mature tool for designing heterogeneous systems. It follows the synchronous dataflow computing paradigm. The input language of the tool is MaxJ with a Java-like syntax. It produces synthesizable designs in VHDL language. A program is described as a set of communicating components named *Kernels* which are responsible for computation. Communication between *Kernels* is orchestrated by a *Manager* component. Unfortunately, the support of external IP cores is lacking. It only supports the use of static (non-parameterized) IP cores. It also does not support HDL code generators. Moreover, to integrate an external IP core one must manually integrate this IP core in system.

XLS: XLS [22] is an open-source HLS tool for producing synthesizable designs in Verilog and SystemVerilog from high-level behavioral descriptions. The program is described in DSLX language with a syntax similar to that in Rust [23] language. The underlying model of the language is synchronous dataflow. A program in DSLX is called a module and consists of functions which describe computations. Communication is organized by means of *procs*. A *proc* contains a config function that initializes constant *proc* state and spawns any other dependent *procs* needed for execution and a recurrent next function that contains the logic to be executed by the *proc*. *Channels* are used to get and send information between different *procs*. DSLX does not support the usage of external IP cores at all.

IV. CONCEPT

A. Language constructs

We introduce DFC, a synchronous dataflow language with the support of external HDL libraries. DFC produces synthesizable designs described in Verilog language. DFC has a C-like syntax, helping developers in learning this language. The reuse of IP cores is one of the key factors in achieving productivity in hardware design. So one of the features of DFC is the use of IP core libraries specified by IP-XACT specification. DFC also gives the developers an ability to control synthesis by applying different optimization criteria such as performance, area and energy consumption. The proposed language is described below.

A program in DFC is divided into *Kernels*. An example illustrating program structure presented in Listing I. *Kernel* performs computations. *Kernels* are essentially graphs of pipelined arithmetical units. Other *Kernels* can be instantiated in the *Kernel* and connected to the *Kernel*. Also, a *Kernel* is

responsible for loading and integrating external HDL libraries. A simple analogy for a *Kernel* is a Verilog module [24].

DFC define *basic* and *composite* types. There are three *basic* types: *fixed* type, *float* type and *bits* type. *Fixed* type represents fixed point number type. *Float* type represents floating point number type. *Bits* type represents a stream of raw bits. The bit width of all *basic* types is configurable. For the *fixed* type one can also configure its sign and the number of bits for the integer and the fraction part. For the *float* type the sign and the number of bits needed to represent the mantissa and the exponent can also be configured. There are two *composite* types in DFC: *tuple* type and *tensor* type. *Tuple* type represents a mathematical tuple, that is, a finite sequence of elements. *Tensor* type represents a mathematical tensor. Table I describes implemented operations for *complex* and *basic* types.

It should be noted that if there is a type mismatch the developer must explicitly transform data from one type to another. This is done using *cast* operator. *Cast* operator is defined only on *basic* types and accepts the following transformations: *fixed* to *float* and vice versa, *float* to *bits* and vice versa and *bits* to *fixed* and vice versa. It also must be used in cases when there is a mismatch in type size.

DFC define control flow constructs: *ternary operator* (? :), *if else* and *switch case* constructs with the standard semantics.

Sometimes there is a need to operate on more than one data value in a particular stream, for example, when computing a moving average. Because DFC operates on streams index operation cannot be used to access data at a particular location in a stream. To do this a *stream offset* operation is introduced with the following semantics. Positive offset values allow to access next values relative to a current position in a stream and negative offset values allow to access previous values relative to a current position in a stream. Zero offset value corresponds to a current position in a stream. DFC supports only static offsets, meaning the value of the offset must be known at compile time. In Listing II there is an example of the program using offsets to calculate a three-point moving average.







The key feature of DFC is the support of external HDL libraries. An external HDL library is a collection of IP cores usually from a particular vendor. In DFC these libraries need to be specified using IP-XACT standard [25]. It should be noted that DFC supports both static and configurable IP cores, such as parameterized modules in Verilog or VHDL. It also supports the use of external HDL code generators. This particular feature highly improves design space exploration. Ideally, an external HDL generator is accompanied with the corresponding estimation functions. These functions allow to estimate important characteristics of an IP core to be generated by the generator. This allows to choose the desired configuration based on input optimization criteria without the actual synthesis. This is done during an optimization step. In the following steps external HDL code generators will be called with different parameters and generate different implementations for the desired functionality. These implementations will be integrated in the output design resulting in different design configurations. One of these design configurations will be chosen depending on optimization criteria. Listing III gives an example of using external HDL libraries in a DFC program.

B. Language semantics

To explain the semantics of DFC language a simple DFC program is considered. The code of this program is presented

in Listing IV. The program takes as an input a number of *fixed* type. If the number is greater than 255 then the result is taken as the sum of the current input value, the previous input value and the next input value, divided by 3. Otherwise the result is the sum of the current input value and 1. The dataflow graph corresponding to this code is presented in Figure 1. For illustrative purposes nodes in the graph are divided into different types. Node types and their semantics are described in Table II.

TABLE II. Kernel dataflow graph node types.

	Source node receives input data tokens from external sender
	Sink node sends output data tokens to some receiver
	Constant node sends data tokens of constant value x
	Stream offset node allows to access next/previous value at position n relative to the current position in the input stream
	Computation node performs some arithmetic or logic operation op as well as <i>cast</i> operation
	Mux(multiplexer) node is used for making decisions

First, an input stream and an output stream are defined. This creates the source node (1) and the sink node (2). *If else* construct is synthesized into the mux(multiplexer) node (3). Comparison operator inside the *if else* construct is synthesized into the computation node (4). The constant inside the condition is synthesized into the constant node (5) which output is then connected to the computation node (4), along with the output of the source node (1). The output of the computation node (4) is then connected to the input of the mux node (3). The body of the *if* block is synthesized into the stream offset node (6), the stream offset node (7), the computation node (8), the computation node (9), the computation node (10) and the constant node (11). The output of the source node (1) and the output of the stream offset node (6) are connected to the input of the computation node (8). The output of the computation node (8) is connected to the input of the computation node (9) along with the output of the

stream offset node (7). The output of the computation node (9) is connected to the input of the computation node (10) along with the output of the constant node (11). The body of the *else* block is synthesized into the computation node (12) and the constant node (13). The output of the source node (1) is connected to the input of the computation node (12) along with the output of the constant node (13). The output of the computation node (10) and the output of the computation node (12) are connected to the inputs of the mux node (3). Finally, the output of the mux node (3) is connected to the input of the sink node (2).

Following the dataflow paradigm, nodes execute (or fire) in parallel if there are no data dependencies between them. A node executes if there is enough data units (tokens) on the inputs, passing its output data tokens to other nodes. So the data flows from sources to sinks provided there are no loops in the dataflow graph. Pipelining is automatic so the programmer can focus on other aspects of development. DFC operates on streams, that is, on potentially unbounded data sequences.

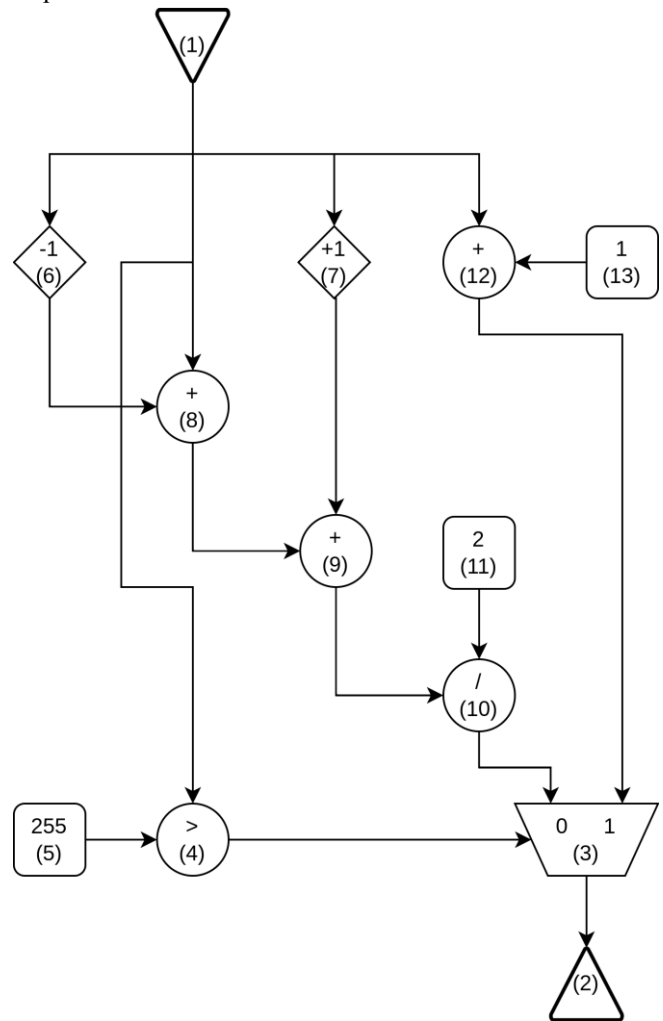


Fig 1. A Kernel dataflow graph.

V. CONCLUSION

In this paper we presented DFC, a dataflow language for constructing electronic designs with support of external IP core libraries. We described its semantics and its key features. Future work may include the support of pragmas helping to achieve better quality of results.

REFERENCES

- [1] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005, 2006. DOI: 10.1109/IEEESTD.2006.99495.
- [2] *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-2019, 2019. DOI: 10.1109/IEEESTD.2019.8938196.
- [3] *Vitis HLS* – <https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0090-vitis-hls-hub.html>
- [4] *Bambu: A Free Framework for the High-Level Synthesis of Complex Applications* – https://panda.deib.polimi.it/?page_id=31
- [5] *LegUp High-Level Synthesis* – <http://legup.eecg.utoronto.ca/>
- [6] *AccelDSP* – <https://www.xilinx.com/support/documentation-navigation/development-tools/mature-products/acceldsp.html>
- [7] *MaxCompiler* – <https://www.maxeler.com/products/software/maxcompiler>
- [8] *XLS: Accelerated HW Synthesis* – <https://google.github.io/xls/>
- [9] E. A. Lee and D. G. Messerschmitt. *Static scheduling of synchronous data flow programs for digital signal processing*. IEEE Trans. Comput., C-36(1), 1987. P. 24-35. DOI: 10.1109/TC.1987.5009446
- [10] *XLS: Accelerated HW Synthesis* – <https://google.github.io/xls/>
- [11] *MaxCompiler* – <https://www.maxeler.com/products/software/maxcompiler>
- [12] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. IEEE Std 1685-2014, 2014. DOI: 10.1109/IEEESTD.2014.6898803.
- [13] *Vitis HLS* – <https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0090-vitis-hls-hub.html>
- [14] *Bambu: A Free Framework for the High-Level Synthesis of Complex Applications* – https://panda.deib.polimi.it/?page_id=31
- [15] *LegUp High-Level Synthesis* – <http://legup.eecg.utoronto.ca/>
- [16] *Intel FPGA SDK for OpenCL* – <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-openccl/overview.html>
- [17] *MaxCompiler* – <https://www.maxeler.com/products/software/maxcompiler>
- [18] *XLS: Accelerated HW Synthesis* – <https://google.github.io/xls/>
- [19] G. Bilsen, M. Engels, R. Lauwereins, J. Peperstraete. *Cycle-static dataflow*. IEEE Transactions on Signal Processing, 44(2), 1996. P. 397-408. DOI: 10.1109/78.485935.
- [20] T.M. Parks, J.L. Pino, E.A. Lee. *A Comparison of Synchronous and Cyclo-Static Dataflow*. Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, 1995. P. 1-7. DOI: 10.1109/ACSSC.1995.540541.
- [21] *MaxCompiler* – <https://www.maxeler.com/products/software/maxcompiler>
- [22] *XLS: Accelerated HW Synthesis* – <https://google.github.io/xls/>
- [23] *Rust Programming Language* – <https://www.rust-lang.org/>
- [24] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005, 2006. DOI: 10.1109/IEEESTD.2006.99495.
- [25] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. IEEE Std 1685-2014, 2014. DOI: 10.1109/IEEESTD.2014.6898803.

LISTING I. DFC program structure.

```

DFC_KERNEL(Kernel1) {
    static const int SIZE = 4;

    DFC_KERNEL_CTOR(Kernel1) {

        std::vector<dfc::stream<dfc::sint16>> lhs;
        std::vector<dfc::stream<dfc::sint16>> rhs;
        ...
        for (std::size_t i = 0; i < SIZE; i++) {
            res[i] = lhs[i] + rhs[i];
        }

    }

};
...
DFC_KERNEL(Kernel2) {

    DFC_KERNEL_CTOR(Kernel2) {
        std::vector<dfc::stream<dfc::sint16>> blk;
        ...
        DFC_CREATE_KERNEL(Kernel1);

        dfc::instance("Kernel1", "KernelInstance1");
        dfc::connectionToInstanceInput("KernelInstance1", blk[0], "lhs_0");
        dfc::connectionToInstanceInput("KernelInstance1", blk[1], "lhs_1");
        dfc::connectionToInstanceInput("KernelInstance1", blk[2], "lhs_2");
        dfc::connectionToInstanceInput("KernelInstance1", blk[3], "lhs_3");

        dfc::connectionToInstanceOutput("KernelInstance1", blk[8], "res_0");
        dfc::connectionToInstanceOutput("KernelInstance1", blk[9], "res_1");
        dfc::connectionToInstanceOutput("KernelInstance1", blk[10], "res_2");
        dfc::connectionToInstanceOutput("KernelInstance1", blk[11], "res_3");
        ...

    }

};
...

```

LISTING II. Calculating three-point moving average.

```

DFC_KERNEL(MovingAverage) {

    DFC_KERNEL_CTOR(MovingAverage) {

        dfc::stream<dfc::sint16> input;
        dfc::stream<dfc::sint16> result;
        result = (input.offset(-1) + input + input.offset(1)) / 3;

    }

};

```

LISTING III. Usage of external HDL libraries.

```

DFC_KERNEL(Kernel) {
DFC_IMPORT_HDL_LIBRARY("path_to_library/ipxact_catalog.xml");

DFC_KERNEL_CTOR(Kernel) {
  dfc::external_generator(FFT_generator, "FFT_generator_instance");
  dfc::stream<dfc::sint16> first;
  dfc::stream<dfc::sint16> second;
  dfc::stream<dfc::sint16> result;
  dfc::connectionToInstanceInput("FFT_generator_instance", first, "first");
  dfc::connectionToInstanceInput("FFT_generator_instance", second, "second");
  dfc::connectionToInstanceOutput("FFT_generator_instance", result, "res");
}
};

```

LISTING IV. An example of DFC program.

```

DFC_KERNEL(MovingAverage) {
DFC_KERNEL_CTOR(MovingAverage) {

  dfc::stream<dfc::sint16> input;
  dfc::stream<dfc::sint16> output;
  if (input > 255) {
    output = (input.offset(-1) + input + input.offset(1)) / 3;
  } else {
    output = input + 1;
  }
}
};

```

TABLE I. DFC operations.

Type / Operation	fixed	float	bits	tuple	tensor
Assignment (=)	✓	✓	✓	✓	✓
Addition (+)	✓	✓			✓
Subtraction (-)	✓	✓			✓
Multiplication (*)	✓	✓			✓
Division (/)	✓	✓			✓
Bitwise operations: AND(&), OR(), XOR(^), NOT(~)	✓		✓		
Shifts: logical (<<, >>), arithmetical (<<<, >>>)	✓		✓		
Comparison operators: EQ(==), NE(!=), LE(<=), GE(>=), LT(<), GT(>)	✓	✓	✓		
Indexation ([])	✓	✓	✓		✓