Improved VHDL Input for High-Level Synthesis Tool xTractor

Peeter Ellervee, Eero Ivask, Margus Kruus

Department of Computer Engineering, Tallinn University of Technology, Raja 15, 12618 Tallinn, Estonia,
e-mail: lrv@cc.ttu.ee, ieero@pld.ttu.ee, kruus@cc.ttu.ee

ABSTRACT: In this paper, an improved version of VHDL compiler for an academic high-level synthesis tool xTractor is presented. A synthesizable subset of behavioral VHDL, accepted by the compiler, is described in brief. Improvements that allow descriptions at higher abstraction level are outlined. The compiler translates the behavioral VHDL subset into control oriented flow-chart like internal description IRSYD. The mapping of higher abstraction level VHDL constructs into IRSYD is described in more details. A short description of the synthesis tool is presented.

1 Introduction

The development of modern microelectronic industry allows to implement complex digital systems on a single chip. To design such a system is not a trivial task anymore because the increasing complexity rises a whole set of different problems. In general, they are related to the need to bring the new product onto the market as fast as possible. The limited productivity of designers is one of the bottlenecks. One way to increase the productivity is to rise the abstraction level designers are working with. This assumes that also the tools used to automate the design process can handle higher abstraction levels as well.

Various hardware description languages (HDL) have gained much popularity among hardware designers during the last decade because of the advantages they offer over traditional schematic techniques. The main advantage is the possibility to use the same description both to model the behavior and as a starting point for schematic synthesis. Another important feature of most of the HDL-s is the possibility to describe an algorithm at higher abstraction levels thus hiding target technology dependent hardware implementation details. At the same time, the introduction of high-level synthesis (HLS), a.k.a. behavioral synthesis, promised to automate the transformation of a design from system/behavioral level to register-transfer level as efficiently as the introduction of logic synthesis automated transformation from logic to physical level. Most of the HLS tools (methodologies) make use of HDL-s as the language of input and output data.

VHDL (VHSIC Hardware Description Language) is one of the most popular languages, if not the most popular one. VHDL supports top-down, bottom-up, and mixed development methodologies. Designs in VHDL can be made technology-independent. That is, no redesign is needed, or a very limited redesign is required, when switching to a new technology. Development times in VHDL are shorter and maintenance is simpler compared to the traditional schematic design (see, e.g., [1]).

The main problem when using VHDL is that only a limited set of constructions can be mapped onto hardware without different possible interpretations. This limited set is called synthesizable subset and it is defined differently for different abstraction levels. In this paper, we describe the principles of mapping a VHDL subset onto flow-chart like internal representation used by an academic HLS tool xTractor [2]. Improvements added to the first version of the front-end compiler [3] are described in more details.

The used internal representation, a subset of IRSYD [4], is in essence a directed graph in which the arcs explicitly represent the flow of control. Four types of blocks are used currently: *entry* and *exit* blocks to mark beginning and end of the control flow, *operation* blocks to encapsulate computational activity, and *condition* blocks to describe branching on a variable. Edges of the graph can have special associations that represent either wait statements of the source code or states of the Mealy FSM corresponding to the IRSYD. An example code with the corresponding piece of flow-chart is shown in Fig. 1.

The paper is organized as follows. Supported VHDL constructs are described in section 2. The principles of mapping VHDL constructs onto IRSYD ones is explained in section 3. A brief overview of the xTractor tool is given in section 4.

2 Supported VHDL constructs

VHDL is a very complex language and it was originally designed for simulation and not for synthesis. Therefore a synthesizable subset of the language had to be defined. The abstraction level used in HLS simplified the task – only constructs used for behavioral descriptions are needed (see, e.g., [5]). Initially, we excluded constructs that could not be mapped directly onto bit-vectors or operations with them because additional decisions were needed. For instance, encoding is needed for enumerated data types and bit-layout is needed for floating point numbers. For the improved version of the compiler, various

additions were made. These additions rely either on simple assumptions (described together with supported constructs) or on additional information provided by the designer via parameters for the compiler. The front-end (compiler) is oriented to VHDL'93 standard because VHDL'86 lacks few constructs often used by hardware designers, e.g., built-in shift operations. The supported constructs are listed as follows.

Design units: Initially, only *entity* and *architecture* were supported. A design can still have one entity and one architecture (no configurations are supported for simplicity) but to support reuse and higher abstraction levels, user defined *package*s have been added. Also, dividing the design between multiple input files is supported now. It should be noted that although *library* constructs are recognized, they are essentially ignored. The same applies for standard IEEE packages (supported data types and conversion functions are built-in). This was needed to avoid conflicts between the code recognized by the compiler and VHDL standard.

For entity, *generic* and *port* declarations are supported. Initially, only port declarations were allowed but to support parameterizable designs, generics were added. Four directions are allowed for ports – *in*, *out*, *inout*, and *buffer*. The last one is mapped onto 'out' direction because IRSYD does not have the restriction that output ports can not be read.

Data types: The supported types are *boolean*, *bit*, *bit_vector*, *integer*, *signed*, *unsigned*, *std_logic*, *std_logic_vector*, and their *subtype*s – all of them can be mapped directly onto signed/unsigned bit-vectors (sets of wires in hardware). One of the most important improvements, together with the support of packages, was the support of subtypes because they are often used for refinements in the design process. In addition, *enumerated* data types have been added but with restrictions – values must match rules for VHDL identifiers, that is, for instance, enumerated types with characters as elements are not supported. The encoding of enumerated data types is based on an assumption that the first element gets the binary code corresponding to 0, the second gets the code corresponding to 1, and so on. Of course, this limits possible encodings – it is easy to change the order of elements in the type declaration but, for instance, one-hot encoding is not possible – and the use of pragma like constructs and/or user defined attributes has been added to the list of future improvements.

Conversion functions, needed by the strict rules of VHDL, are ignored because all supported data types map onto the same underlying type, that is, onto bit-vector. Three kinds of data objects – *constant*s, *variable*s, and *signal*s – of supported types can be declared. One-dimensional *arrays* of allowed types are supported. Elements of *record*s are split into separate data objects.

Operations: All operations with supported data types are allowed – they map onto basic operations with bits or bit-vectors. The only difference is between few signed and unsigned operations, e.g., comparisons must treat the most significant bits differently. Enumerated and *std_logic_vector* types are treated as unsigned bit-vectors. It should be noted that the compiler accepts even operations not allowed by VHDL standard, e.g., logic operations between integer types. This simplification was made to avoid complex context analysis and can be justified by an assumption that the functionality of the code is first validated using a VHDL simulator that should catch all violations of VHDL syntax/semantics.

Attributes: The acceptance of some built-in attributes is one of the improvements to support parameterizable VHDL coding styles. The attributes are divided into two groups depending on where they can be used. Data type and object related static attributes – *high*, *low*, *left*, *right*, *length*, *range*, and *reverse_range* – can be used either in declarations or in expressions as constants. In fact, these attributes are mapped onto IRSYD constants and must be known at compile time. Signal behavior related attributes – *event* and *stable* – can be used only in *wait* statements (see section 3 for details).

Concurrent statements: Only a single process is allowed per architecture because of the underlying single thread execution model of HLS. Architectures with multiple concurrent processes should be divided between different sub-components. Structural hierarchy is not supported because it does not have anything to do with a functionality of an algorithm and can be easily handled by register-transfer or logic level tools, for instance. The only other supported concurrent statement is a simple assignment between a signal and a port.

Behavioral statements: All statements that can be mapped onto control flow represented by a flow-chart (IRSYD) are supported (see section 3 for details):

- ***Assignment*s** and ***expression*s** are divided into atomic operations and mapped onto operation blocks;
- ***If-then-elsif-else*** statements are mapped onto sequences of operation and condition blocks;
- ***Case*** and ***loop*** statements are first mapped onto *if-then-elsif-else* constructs and then onto flow-chart blocks. The improvement was that attributes can be used to define loop boundaries; and
- The extension of supported ***wait*** statement styles was one of the major improvements – now *on* clause can be used to define the clock signal, *until* clause to define clock edge and additional conditions, and *for* clause to set timing constraints. The use of full semantics of the *wait* statement assumes that some extra information is provided for the VHDL front-end compiler – the name and phase of the clock signal (if not detectable from earlier *wait* statements) and frequency of the clock signal. The later is needed to build counters corresponding to timing constraints.

Some of the unsupported constructs, especially those that support behavioral hierarchy, will be added later. They are not necessary to describe modules at behavioral level but

allowing them makes descriptions more flexible and generic. The constructs to be added later are *function* and *procedure*. We do not intend to allow constructs that are either simulation oriented, e.g., configuration and file operations, or are at non-behavioral abstraction levels, e.g., components and data-flow constructs. The higher level constructs should be replaced with supported constructs during earlier design phases, and the lower level constructs can be easily handled by corresponding back-end synthesis tools, assuming that these constructs are not fed into the HLS tool xTractor.

3 Mapping VHDL onto IRSYD

This section describes the correspondence between VHDL and flow chart constructs.

Data objects: All supported VHDL data objects are mapped onto IRSYD constants, variables, or signals. This applies also both for generics and values of enumerated data types. Such a mapping is valid because for synthesizability, it is safe to assume that the values of these objects are known at the compile time. All supported data types are converted into signed or unsigned bit-vectors (IRSYD data types). The only exception is the clock signal – it is converted onto *bit* type to avoid conflicts with back-end logic synthesis tools.

Entity and architecture: All ports in a VHDL entity are kept as ports also in IRSYD. Signals and variables remain intact in IRSYD, only data types are converted into signed or unsigned bit-vectors when necessary. VHDL constants are preserved also as constants in IRSYD but necessary type conversions are performed.

Process supports now only *wait* statements for timing control. The earlier support for the sensitivity list was removed to avoid misinterpretations – a process with the sensitivity list is typically used to describe combinational circuits and therefore should be handled by register-transfer and/or logic level synthesis tools. According to VHDL semantics, there should be at least one wait statement in every control flow path of the process to avoid infinite loops.

Behavioral statements in VHDL process are translated into operation and control blocks of IRSYD. Fig. 1 illustrates how *loop* and *case* constructs, and assignments are mapped onto a sequence of IRSYD blocks. The other statements are also mapped onto sequences of control flow blocks. The underlying principle is defined by the fact that the control flow in IRSYD is essentially defined by two block types – condition and operation blocks. All other blocks have supporting roles.

- ***Assignments*** are mapped onto operation blocks - each operation in a block corresponds to an operation in the expression of the assignment. Timing controls, i.e., *after* clauses, are not supported.
- Condition calculations in ***if-then-elsif-else*** statements are mapped onto operation blocks like assignments. Boolean result of the calculated condition is evaluated by a condition blocks that takes care of branching.
- Comparisons in ***case*** statements are mapped onto operation blocks (like in *if-then-elsif-else* statements). Statement sequences of branches are parsed in recursive manner. It should be noted that due to the use of condition blocks, the parser does not require that all combinations are covered or the 'when others' construct is used – the default branch always exists.
- ***Loop*** initialization and boundary condition calculations are mapped onto operation blocks and flow control onto condition blocks.
- The complexity of supported ***wait*** statements has been extended. The basic construct is "wait on clock_name until clock_name=clock_phase;" that corresponds to a special associations on the corresponding flow-chart edge. The clock signal name and phase are detected automatically and checked for conflicts compared against other wait statements. In addition, clock signal can be defined by using compiler parameters. The use of more complex conditions after *until* is mapped onto while loop. Timing control in the wait statement is allowed – an extra counter will be created. To support that, the clock frequency (or period) has to be defined by the designer.

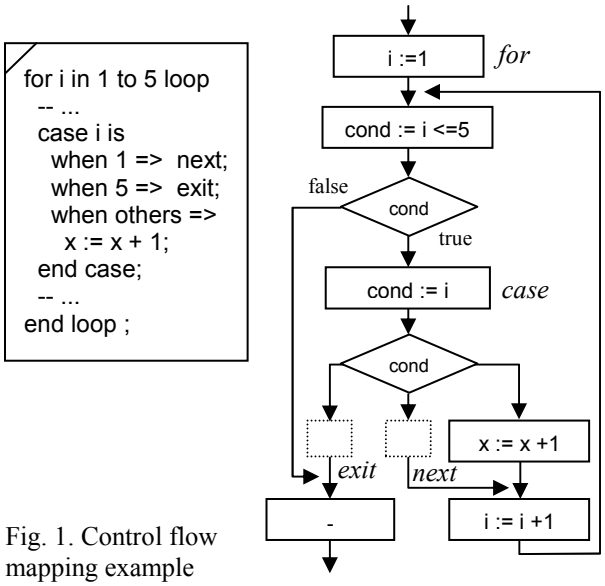

Fig. 1. Control flow mapping example

Statement sequences of conditional branches and loop body are parsed in recursive manner. Also, all statements can have labels.

4 HLS tool xTractor

xTractor is an academic high-level synthesis tool that was developed to test synthesis methodology of control and memory intensive systems (CMIST). The overall synthesis flow (see Fig. 2) is similar to the synthesis flow of any HLS approach (see, e.g., [5]). The three main steps can be outlined as follows:

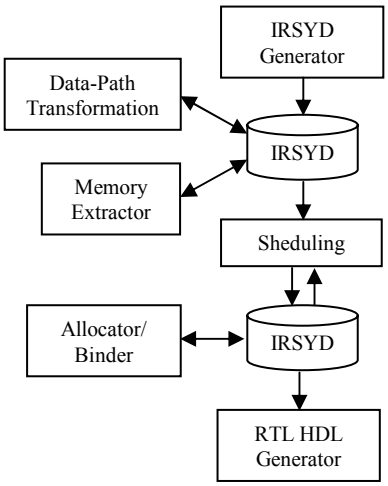

Fig. 2. xTractor synthesis flow

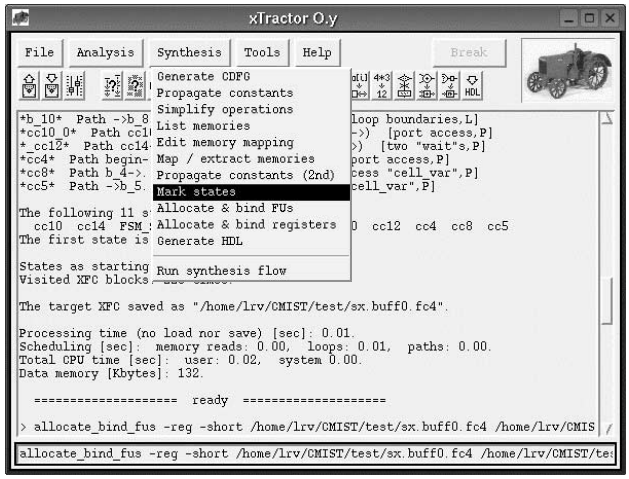

Fig. 3. GUI of xTractor

- in the partitioning phase memories are extracted from the initial behavioral description;
- operations are assigned to states (control steps) during the scheduling phase; and
- unified allocation and binding assigns operations to specific functional units.

The separate steps of the flow are executed by component programs use subset of IRSYD. Two of the component tools are used as input and output of the tool-set. One of the programs - **IRSYD Generator** - compiles either a subset of C or VHDL into IRSYD. The C compiler is based on an extended Rat-C compiler [6]. The details of VHDL compiler were presented in [3]. The compiler uses construction toolset PCCTS [7] from Purdue University to build lexers and parsers. The second tool - ***RTL HDL Generator*** - generates register-transfer level VHDL code for back-end logic-level synthesis tools. Some simpler **Data-Path Transformations** can be applied before (or after) every main step. **Memory Extractor** lists arrays and/or maps them onto memories.

It should be noted that although most of the synthesis steps can be skipped, the **Scheduling** phase is an exception because it is the only step that assigns states to the behavioral control flow. So-called as-fast-as-possible (AFAP) scheduling strategy is used. ***Allocator/Binder*** allocates and binds operations and variables into functional units and registers. Multiplexers are allocated and bound together with related functional units and registers.

A separate component is an interactive shell that organizes the overall synthesis flow, that is, in which order and with which parameters the component programs are called. The shell organizes also the graphical user interface (GUI) and menu systems. Fig. 3 shows the main window with report of the scheduling step. xTractor has been used to synthesize modules from industrial examples. Results of these experiments have been reported in [2][3][4]. The latest improvements of the tool have been focussing onto making it useable for educational purposes.

6 Conclusion

We have presented the key issues of mapping descriptions in behavioral VHDL onto IRSYD constructs, an internal representation suitable for control and memory intensive digital systems synthesis. Extensions to the initial simple synthesizable subset have been described. The extensions were added to rise the abstraction level of high-level synthesis thus freeing the designers from the need to focus onto implementation details. The compiler has been tested in the design flow of an academic high-level synthesis tool xTractor.

Acknowledgements: This work was supported in part by Estonian Science Foundation grants No 5601 and 6717.

References

[1] B. Cohen, *VHDL Coding Styles and Methodologies*. Kluwer Academic Publishers, 1999.

[2] P. Ellervee, "xTractor: An Academic High-Level Synthesis Tool for Control and Memory Intensive Applications." *The 20th NORCHIP Conference*, Copenhagen, Denmark, pp. 253-258, Nov. 2002.

[3] E. Ivask, P. Ellervee, "VHDL Front-End for High-Level Synthesis Tool xTractor." The 9th Biennial Baltic Electronic Conference (BEC'2004), Tallinn, Estonia, Oct. pp.111-114, 2004.

[4] P. Ellervee, *High-Level Synthesis of Control and Memory Intensive Applications*. Ph.D. Thesis ISRN KTH/ESD/AVH--2000/1--SE, Stockholm, 2000.

[5] P. Eles, K. Kuchcinski, Z. Peng, *System Synthesis with VHDL*, Kluwer Academic Publishers, 1998.

[6] R. E. Berry, B. A. E. Meekings, *A Book on C*, Macmillan Publisher, 1984.

[7] Compiler construction toolset PCCTS -- http://www.polhode.com/pccts.html