High-Level Synthesis using Dependence Flow Graphs as the Intermediate Form

Ntsibane Ntlatlapa
Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347
ntlatns@eng.auburn.edu

January 20, 1998

Abstract- *This paper describes an ongoing high level synthesis project using dependence flow graphs as intermediate form. This HLS system takes a behavioral description written in a subset of VHDL and generates an RTL VHDL description that implements the design. We use dependence flow graphs as an intermediate form. We only describe the compilation part of the HLS system and dependence flow graphs.*

1 Introduction and Related Work

The research described here is an effort to develop a digital hardware synthesis system that automatically translates a behavioral algorithm description to structural implementation. This research is a sub-project of the project: Formally, Verified, Efficient Tools for High-Level Synthesis and Hardware-Software Codesign. The main goals of the project are to: produce digital design tools that are both efficient and correct and to show the feasibility of formally verifying design tools themselves, rather than requiring designers to apply verification techniques to each design produced.

These tools will form a system that will translate a behavioral description written in VHDL description language to a register-transfer level circuit in structural VHDL. Our target architectures are Xilinx and Altera *field programmable gate arrays (fpga).*

This work is partially based on work done by Hwang [6]. He developed a process-algebraic semantics for VHDL. A survey of high-level synthesis reveals that system designers are adopting VHDL as the language to describe the behavior of digital systems. These include the Honeywell's V-synth system [3] and IBM's HIS system [4]. This survey also reveals that designers continue to employ a control flow graph and a collection of data flow graphs as an intermediate form for synthesis. We adopted the *dependence flow graph* as our intermediate form. The dependence flow graph is an executable representation of data, control, timing and resource-usage dependencies present in the specifications of the digital systems, both hardware and software [8]. The dependence flow graph was originally developed for use in optimizing compilers [2]. Chapman [5] extended the operational semantics of software dependence flow graphs to apply to dependence flow graphs modeling hardware.

2 High-Level Synthesis

The high-level synthesis system, Figure 1 design flow starts with behavioral VHDL specification. We then build an abstract syntax tree, then build the dependence flow graph from the syntax tree. The dependence information maintained by dependence flow graphs is used for partitioning, optimization, scheduling and data path allocation to produce a register-transfer level circuit in structural VHDL. This circuit can then be input to a commercial tool set for low-level logic synthesis, placement and routing, and translation to device-specific format.

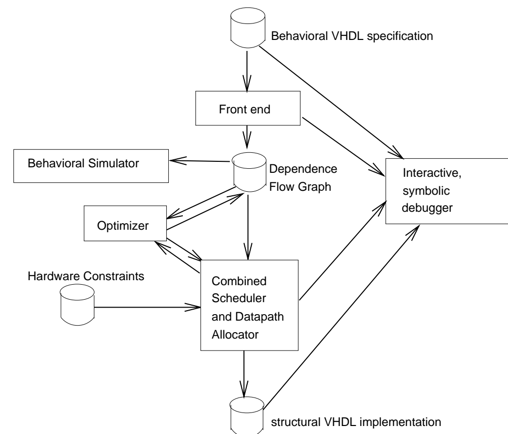

Figure 1: High-Level Synthesis System

3 Behavioral VHDL

VHDL is a language for describing digital electronic systems. It has been adopted as a standard by the Institute of Electric and Electronic engineers (IEEE) in the US [7, 1]. It allows description of the structure of the design (i.e. how it is decomposed into sub-designs, and how those sub-designs are interconnected). It also allows the specification of the function of designs using familiar programming language forms.

The primary unit of behavioral description of VHDL is the *process*. A process is sequential body of code which can be activated in response to changes in state. Statements in a process can execute concurrently. A process is specified in a process statement, with the syntax in Figure 2.

process_statement ::=
  [process_label :] **Process** *[(sensitivity_list)]*
    process_declarative_part
  Begin
    process_statement_part
  End Process *[process_label]*;

Figure 2: Process Statement Syntax

A process may contain a number of signal assignment statements for a given signal, which together form a driver for the signal. In order to model signals with multiple drivers, VHDL uses the notion of resolved types for signals. A resolved type includes in its definition a resolution function, which takes the values of all the drivers contributing to a signal, and combines them to determine the final signal value. A process executes all of the sequential statements, and then repeats, starting again with the first statement. The execution of a process is semantically equivalent to an execution of an infinite loop in Figure 3.

infinite_loop : **Loop**
Begin
⋮
End Loop *infinite_loop;*

Figure 3: Infinite Loop

A process may suspend itself by executing a wait statement. If the sensitivity list is included in the header of a process statement, then the process is assumed to have an implicit wait statement at the end of its statement part. The following two segments of VHDL code in Figures 4 and 5 are semantically equivalent.

Communication between processes is provided through the shared signal values. All the processes have a global view of the signals, once a signal is modified, all the processes waiting for that signal, execute the conditions in the wait statements. They then either resuspend themselves if the condition is false or continues with the next statement after the wait, if the condition is true.

Process
Begin
⋮
Wait On *s1;*
End Process*;*

Figure 4: Process without sensitivity list

Process*(s1)*
Begin
⋮
End Process*;*

Figure 5: Process with sensitivity list

4 Intermediate Representation - Dependence Flow graphs

We use the *dependence flow graph (dfg)* as our intermediate representation. The dfgs were developed for compiling imperative languages for data flow architectures. Dependence flow graphs integrate data and control dependence information into a single structure, making efficient algorithms for program analysis and optimization possible. They are also executable.

A dependence flow graph consists of set of nodes representing operations and a set of edges representing the dependencies and precedence relations that exist between those operations. If a node, say $node_d$ needs a node computed by $node_s$, then there must be a path from $node_s$ to $node_d$. Similarly if there is a dependence between two statements in the source program, then the must be a path between the corresponding nodes in the dependence flow graph. Control dependencies are represented by switches and merges that routes data and resource dependencies to several destinations based on the control condition's value.

We extend dependence flow graphs to model the timing information present in a specification written in behavioral VHDL, and to model parallelism and communication for a group of concurrently executing sequential processes. One of the most significant addition to the dfgs to handle communicating processes in VHDL is the *wait* node. The semantics of the wait node are

equivalent to that of the wait statement in VHDL. The inputs to the wait statements are the signal to wait on, the condition and the time. The dfg in Figure 6 shows the effects of a wait statement.

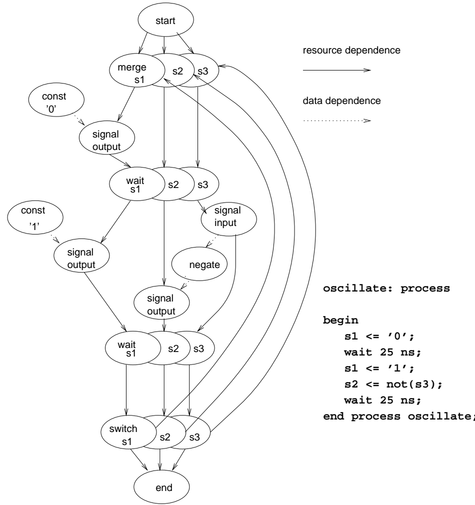

Figure 6: Example of a dfg with wait statement

5 Translating to DFG - Front End

The parser generates an *abstract syntax tree* from a behavioral VHDL description. Then, the dfg is generated from the syntax tree. The algorithm has two phases: in phase one the initial dfg is generated by walking the abstract syntax tree. Arcs for each identifier are routed into and out of every control region, keeping track of the name of the arc carrying the value of the each identifier.

One of the most important aspects of this research is the capability to formally verify our tools: we must show that our translation is correct by showing that the configuration given by the VHDL specification is equivalent to the one resulting from the execution of the dfgs according to their operational semantics.

To exploit all the parallelism within a single control region we perform a dfg-dfg transformation in the second phase. The transformation removes the arcs of identifiers from the control regions that do not use them. The transformation can be done in a single pass over the *dfg* if we process nested control regions from the innermost to outer most.

6 Behavioral Simulator

This is subject to further investigation. The idea is to model every node in the dfg with a VHDL entity and architecture describing its behavior.

7 Current Status

The dependence flow graph package that we use is called *Pigdin*, and it was originally written at Cornell University and then later modified at Auburn University. We have completed the extensions that are needed to model VHDL to that package, and the compilation phase from VHDL to DFG. The Datapath Allocator has also being completed by another researcher. We are currently working on the Simulator and Optimizer.

References

[1] P. J. Ashenden. *VHDL Cookbook*. Dept. Computer Science, University of Adelaide, South Australia, 1st edition, July 1990.

[2] M. Beck, R. Johnson, and K. Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 1991.

[3] J. Bhasker and H.-C. Lee. An optimizer for hardware synthesis. *IEEE Design and Test*, pages 20–36, Oct. 1990.

[4] R. Camposano, R. Bergamaschi, C. Haynes, M. Payer, and S. Wu. The ibm high-level synthesis system. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[5] R. Chapman. *Verified High Level Synthesis*. PhD thesis, Cornell University, Ithaca, New York, Jan. 1994.

[6] D. Hwang. *Using Untimed CSP to Formally Verify Compilation of VHDL for High Level Synthesis*. PhD thesis, Auburn University, Auburn, Alabama, Aug. 1995.

[7] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1987. IEEE std 1076, IEEE Press.

[8] K. Pingali, M. Beck, R. Johnson, M. Moudghill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, Jan. 1991.