A Verilog Preprocessor for Representing Datapath Components

Brian Davis & Trevor Mudge
EECS Dept.
University of Michigan, Ann Arbor

A Version appears in the Proc. 1995 International Verilog HDL Conference
March 27-29, 1995, pp. 90-98.

Abstract

This paper describes research leading to the generation of a preprocessor for the Verilog hardware description language. The function of this preprocessor is to support repeated feature instances in the datapath of a Verilog description for a digital system. Repeated features most commonly occur in the description of datapaths, where iterative structures like adders, multipliers and muxes are the basic building blocks. A deficiency of Verilog, its lack of support for repeated feature instances, is identified. Citations from Verilog users and industry organizations in support of inclusion of a repeated feature syntax are given. Several syntaxes for describing repeated features are presented. From these proposals, a single syntax for support of repeated feature instances is selected. A preprocessor is described that will parse the extended Verilog and translate it to supported Verilog. The challenges in the generation of the preprocessor are given, along with the solutions used to overcome them. The paper concludes with a status report on the preprocessor and thoughts for future development.

Introduction

Hardware description languages (HDLs), are the preferred method of designing digital systems in modern digital design environments. An HDL allows the designer to specify the behavior of a digital system with an unambiguous textual syntax, which provides an entry point into the design process. This specification can also serve as documentation for the digital system [1]. In fact, it was for documentation that HDLs were first employed.

Today, there are two HDLs in common use, Verilog, and the VHSIC Hardware Description Language, VHDL. Verilog is a C-based hardware description language and was originally developed by Gateway Design Systems [2]. Gateway Design Systems was later purchased by Cadence Design Systems, a corporation which provides both Verilog and VHDL based software tools. The specification for the Verilog HDL was released into the public domain in November of 1991. To facilitate this release, Cadence formed Open Verilog International (OVI), a private corporation responsible for maintaining the Verilog language specification in April of 1991 [3]. Recently OVI transferred this responsibility to the IEEE. Currently IEEE working group 1364 is developing the latest version of the Verilog HDL language specification.

.Need for a Repeated Feature Extension

The following example illustrates the need for a repeated feature in Verilog. Figure 1 shows part of a multiplier tree [4]. The first and second levels of partial products in the multiplier have already been generated, and the required functionality is to compress the results from the second level to produce a third level of signals. The third level is in turn fed into another array of compressors. After a number of compressors, determined by number of bits in the operands, the product is formed with a carry-propagate binary adder [4].

The structure of the multiplier array is shown in Figure 1. The source code to describe it using Cadence Verilog-XL compatible syntax, is given in Example 1. The code segment in Example 1 contains two module types, both of which are used here and in Example 2, but are presumed to be defined elsewhere. The FTC module is a four-two compressor, and the FA module is a full-adder. The four-two compressor is defined to have eight one-bit wide ports, the first five being inputs, the last three being outputs. The function of the four-two compressor is to take these five input bits of equal significance and creates two carry bits of next higher significance and one bit of the same significance. The four-two compressor has a characteristic that makes it extremely useful in the generation of multiplier circuits such as the one shown in Figure 1: one of the carry bits is independent of one of the input bits, allowing the carry chain delay to be reduced [5]. The full adder module is defined such that the first three ports are one bit wide inputs, and the last two ports are one bit wide outputs. This module has the functionality of the classic full adder. The points to notice in this example are

that it is highly repetitive and time consuming to enter. The complete description for the multiplier tree is even worse. Clearly, a way to specify repeated features would simplify the expression of such structures.

Extending the Verilog specification to support a repeated feature instance syntax will allow hardware designers using Verilog to specify a bit-slice feature or module, like the four-two compressor above, that can be instantiated across all bits of a datapath in a single statement. This is in contrast to current practice, illustrated in Example 1, where the hardware designer must specify a feature instance with a unique name and complete port connection list for each bit in the datapath. The only help the designer has is to "cut and paste" with the design entry tool. The advantages of a succinct repeated feature syntax are in an increased readability and a reduced likelihood of design errors.

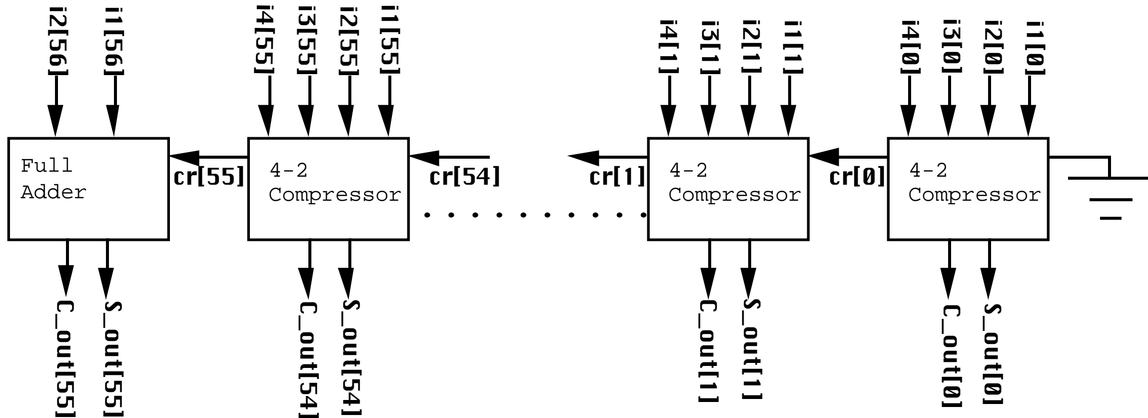

Figure 1. Part of a multiplier tree

```
module RST_row3(i1, i2, i3, i4, s_out, c_out);
input [56:0] i1, i2;
input [55:0] i3, i4;
output [56:0] s_out, c_out;
wire [55:0] cr; // non-propagating Carry chain

FTC    c0(i1[0], i2[0], i3[0], i4[0], TOP.Gnd, s_out[0], c_out[0], cr[0]),
       c1(i1[1], i2[1], i3[1], i4[1], cr[0], s_out[1], c_out[1], cr[1]),
       c2(i1[2], i2[2], i3[2], i4[2], cr[1], s_out[2], c_out[2], cr[2]),
             :                                       :
             :                                       :
             :                                       :
       c53(i1[53], i2[53], i3[53], i4[53], cr[52], s_out[53], c_out[53], cr[53]),
       c54(i1[54], i2[54], i3[54], i4[54], cr[53], s_out[54], c_out[54], cr[54]),
       c55(i1[55], i2[55], i3[55], i4[55], cr[54], s_out[55], c_out[55], cr[55]);
FA     c56(i1[56], i2[56], cr[55], s_out[56], c_out[56]);
endmodule
```

Example 1. Multiplier tree module using Verilog-XL syntax

The desire for a syntax that will allow Verilog to generate multiple instances of a module, feature or gate is wide spread, and recognition of this desire has been made at a number of levels. Verilog users require a repeated feature syntax, as can be seen from articles in the comp.lang.verilog newsgroup [6, 7, 8, 9, 10, 11, 12]. As mentioned earlier, the IEEE working group 1364 is currently in progress on its first version of a Verilog language specification. From discussions with members of this working group, it is apparent that the need for a repeated feature extension is a recognized issue of concern among in the working group [6, 10, 13]. Before the IEEE working group 1364 was formed, Open Verilog International was responsible for the Verilog language specification. The OVI organization recognized the need for a repeated feature extension to the extent that they included such an extension in their last release of the Verilog language specification, the OVI Language Reference Manual (LRM) version 2.0a [14]. Unfortunately this document was released only shortly before IEEE working group 1364 was formed, and no current Verilog tools conform to OVI LRM 2.0a.

Finally, in an issue of Electrical Engineering Times preceding the 1994 International Verilog HDL conference, an article questioned whether the OVI LRM 2.0a array of feature instances (AOI) construct would be maintained into the yet to be released IEEE working group 1364 specification [15].

Proposed Syntaxes

The array of feature instances syntax, specified in OVI LRM version 2.0a [14], is a direct extension to the Verilog language. Within this syntax, an instance of repeated features is viewed as no more than a single dimensional array of these features. This is consistent with the way ports, nets, registers and wires are currently specified in Verilog, i.e., arrays of single bit wide instances. By using the OVI LRM 2.0a syntax for arrays of feature instances, the segment of multiplier source code given in Example 1 could be reduced to the code segment in Example 2.

```
module RST_row3(i1, i2, i3, i4, s_out, c_out);
input [56:0] i1, i2;
input [55:0] i3, i4;
output [56:0] s_out, c_out;

wire [55:0] cr; // non-propagating Carry chain

FTC    col[55:0] (i1[55:0], i2[55:0], i3[55:0], i4[55:0], {cr[54:0], TOP.Gnd},
                    s_out[55:0], c_out[55:0], cr[55:0]);

FA     col56(i1[56], i2[56], cr[55], s_out[56], c_out[56]);
endmodule
```

Example 2. Multiplier tree module using OVI LRM 2.0a syntax

Since only arrays of feature instances can be generated and the arrays must have constant width specifiers, the OVI AOI syntax is limited in that it can only be used for generation of nonconditional feature instances. This is due to the fact that the number of instances being instantiated must be fixed at parse time. For specifying hardware, which is the purpose of the Verilog HDL, this does not constrain the usefulness of the syntax.

The OVI syntax allows for two types of nets to be passed into any array of feature instances. The rules for connections for an array of feature instances as given in the OVI LRM 2.0a are [14]:

- The bitlength of each port in the instance is compared with the module or primitive port's definition
- If the bitlengths are the same, the port expression is connected to each instance.
- If bitlengths are different, each instance gets a part select of the port expression as specified in the range, starting with the right-hand index.
- A warning message is issued if there are too many or too few bits to connect to all the instances. If too few bits are specified, the corresponding ports are zero filled.

The first type of net allowed within the OVI array of feature instances syntax, and described in the second OVI rule above, will be referred to as a static net. The static net is connected to each of the feature instances being created. The static net is recognized by the Verilog parser as being a net of the same width as the definition for the port into which the net is being passed. An example is the control line being connected to each bit slice of the 32 bit wide MUX shown in Figure 2. The second type of net supported by the OVI syntax, and described in the third OVI rule, will be referred to herein as an indexed net. Indexed nets are those for which only a portion of the entire net is connected to each feature instance declared in the array. An indexed net is recognized by the language parser as being a net of width wider than the definition for the port to which it is being connected. An example of an indexed type net would be the datapath inputs (a and b nets) connected to the 32 bit wide MUX shown in Figure 2.

Typically the width of the indexed net, N, is an even multiple of the width of the feature port, P, into which the net is being passed. If this is not the case, i.e. N/P is not equal to the integer number of features being instantiated, then a warning is generated, and the port connections for the feature instances are zero

filled from the left. This is the functionality described in the fourth OVI rule above. The AOI syntax does allow for indexed nets where the net slice being passed into each feature contains multiple bits, but our examples do not illustrate this capability. Indexed and static are the only two types of nets which the OVI LRM 2.0a array of feature instances syntax supports [14].

Before construction of any preprocessor was begun, other possible syntaxes for the support of repeated features were examined. A number of such syntaxes have been proposed. The first of these is the specification of a repeated feature syntax similar to the generate syntax used within VHDL [16]. In this regard, it is interesting to note that in 1992 Cadence studied the task of bringing full VHDL generate functionality to Verilog, but decided not to pursue this task for reasons of complexity. Specifically, adding generate functionality to Verilog would require the addition of such things as scoped declarations and multidimensional arrays to the Verilog language [9]. The next syntax examined for support of repeated features was the interpreted for-loop structure. This is an intriguing concept as it would be consistent with both the C programming language on which Verilog is based, and behavioral Verilog. It was determined that the biggest task in making an interpreted for-loop syntax work for support of repeated features would be generating unique feature instance names via the index variable of the loop. This avenue was not pursued because the functionality of an interpreted for-loop would be identical to that of the AOI syntax, but less compact. The final possible syntax for support of repeated features which was examined was a preprocessed or metaprocessed for-loop structure which is currently in use at some design facilities. This involves using a language such as C to generate the Verilog source code before it is parsed via a Verilog parser [17]. This avenue was not pursued due to its temporary fix appearance and the requirement of an additional step before parsing.

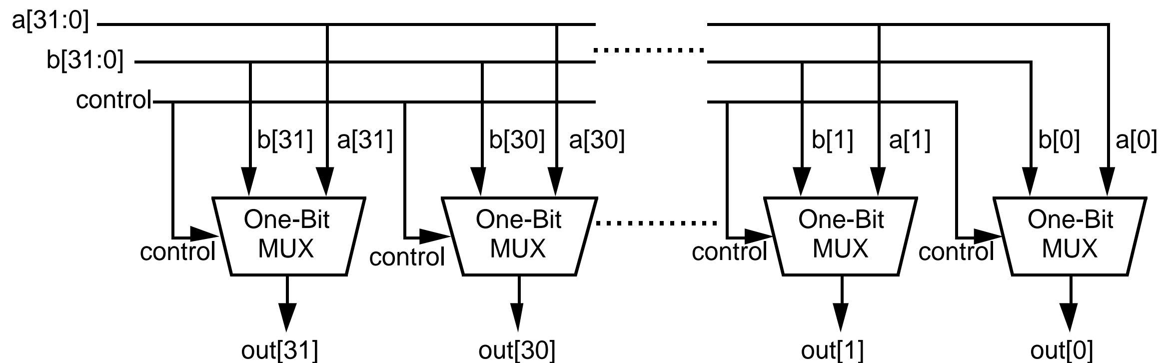

Figure 2. Array of feature (MUX) instances

Any of these proposed syntaxes could provide the functionality desired by the hardware designers requesting support for iterative constructs. We chose to support the array syntax for three reasons. First, this syntax was specified by an organization (OVI) which was at one time responsible for the Verilog language specification and the syntax therefor has some degree of sanctioned support. Second, the array of feature instances syntax is consistent with other portions of the Verilog language which are specified using range indicators('[a:b]'), such as wires and regs. Third, communications with members of IEEE working group 1364, made it clear that if any repeated feature syntax is supported in the first version of an IEEE language specification it would be the OVI LRM 2.0a syntax [10, 13, 18]. If the array syntax were to be present in the LRM released by IEEE working group 1364, this would result in a situation where all Verilog code written for the Verilog preprocessor would also be forward-compatible with future Verilog compilers and/or simulators.

Decision to Generate a Preprocessor

Extending Verilog can be handled in several ways. The most obvious is to build a new interpreter or compiler capable of dealing with the extended language. This has two undesirable features: 1) it involves an enormous undertaking; and 2) existing CAD tools designed to work with Verilog would be useless. Our solution was to build a preprocessor, VPP, that accepts the OVI LRM 2.0a array of instances extension as part of any source code and translates them into Verilog-XL compatible syntax. Thus, the output of the preprocessor is Verilog and all existing simulators and CAD tools developed for Verilog can still be used.

The first step in the creation of a preprocessor was the location or generation of a Verilog parser. According to the frequently asked questions (FAQ) posting on the comp.lang.verilog newsgroup, there are two Verilog parsers in the public domain, both of which utilize LEX and YACC for their parsing capabilities [19].

The first of these is the Berkeley Verilog parser written by S.T. Cheng at Berkeley (stcheng@ic.berkeley.edu) for automatic generation of BLIF-MV simulation code from Verilog code. The second public domain Verilog parser is a parser which was begun, but never completed, by F.W. Bennett (fwb@hpfcso.fc.hp.com). The possibility was also pursued of obtaining a parser from Cadence which would not be in the public domain. However, because of the timeframe when the Cadence parser would be available, and because the resulting preprocessor would then be proprietary, the choice was made not to use Cadence's parser. After analysis of the source code for the two public domain parsers, the Berkeley parser was selected because it was furthest along in development.

The Berkeley parser did not, however, contain functionality for the full Verilog specification. The grammars for Verilog-XL compatible syntax were fully specified within the YACC source code. The format for the dynamically stored representation of the described circuit was specified, and for the most part complete. There were, however, labels at a number of locations in the code which indicated areas where the source code for handling Verilog had yet to be completed. The majority of these "TODO" markers had to be completed before the preprocessor was fully functional.

The flow chart for the Verilog preprocessor is given in Figure 3. The preprocessor accepts as input a Verilog source file, which may or may not contain the array of feature instances syntax, parse this file, and output to disk a new Verilog-XL compatible file which will retain the functionality of the input source file. This output file can then be compiled by Verilog-XL, or an equivalent Verilog HDL simulation package and executed in the normal manner.

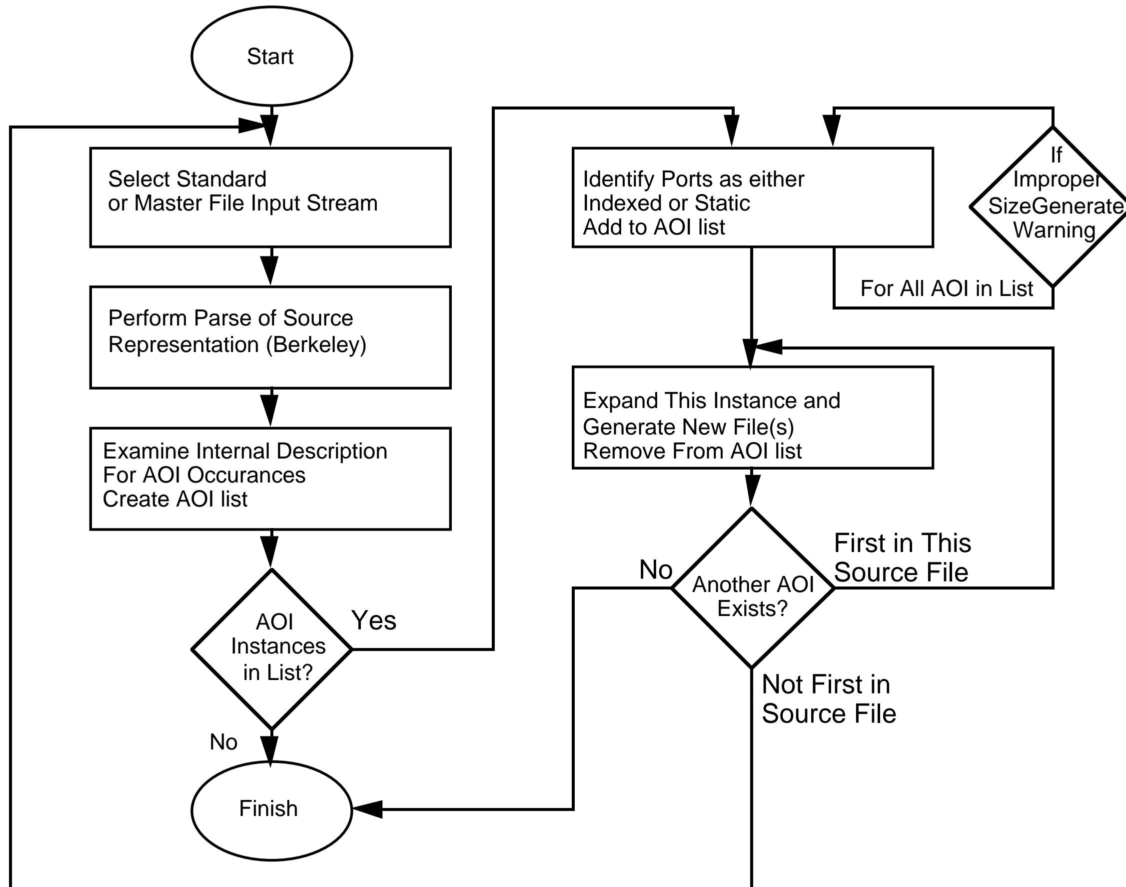

Figure 3. Verilog Preprocessor Flow Chart

One difference between the preprocessor output and the OVI LRM 2.0a specification is unavoidable. This is because in Verilog-XL compatible format, which is the requirement placed on the preprocessor output, each instance of a feature must have a unique name. In the array of instances syntax, all instances instantiated are given the same name, followed by square brackets containing a number. This is the format widget[k].

When generating Verilog-XL compatible syntax, this naming convention can not be used because the special case characters "[" and "]" would cause a parsing error. Therefor in the preprocessor output, the Kth instance of the array of features widget[(N-1):0] is renamed to widget_K. This renaming convention should keep the debugging process intuitive to the designer, and minimize the differences between the OVI specification and the preprocessor output.

Creating the Preprocessor

A variety of tasks were involved in the generation of the preprocessor. Changes were required to the Berkeley parser before it was useful in the generation of the Verilog preprocessor. These changes encompass items which exist in both Verilog-XL compatible syntax, and items which were required to be added for support of the new OVI syntax.

The primary change to the Berkeley parser was the addition of grammars to allow for an AOI to be successfully parsed rather than generating an error. This modification involved not only specifying these new YACC grammars, but writing the functions to handle these grammars when they are encountered, and the addition of variables to the dynamic representation for the storage of the new values present in the AOI syntax.

The next change required to the Berkeley parser was the ability of the parser to recognize hierarchical named nets. This capability and the modifications required to handle it is described later.

The master file capability, a commonly used feature of Verilog-XL, is not supported by the Berkeley parser, and therefor it was required that this functionality be generated in C code. Master files are commonly used where a Verilog project contains more than a single source file. The master file functionality allows the user to specify whether a Verilog source file is to be used as source or a library. This distinction is required for determination of top level modules. The C source code for dealing with master file(s) works in close conjunction with the Berkeley parser, and essentially provides an input stream to the LEXer.

The Preprocessing Stages

A necessary task in the creation of a preprocessor to transform one notation (the array syntax) into another notation (the Verilog-XL compatible iterative syntax) is the identification of the instances to be transformed. In this case, the only instances which need be examined are those where a module instance name has a range specifier ('[a:b]') after it. The code to perform this location was added, and is first to execute after the parse of the Verilog source code is complete.

After an instance of code utilizing the new syntax has been identified, each of the nets in the port connection list must be identified as one of the two types allowed by the OVI specification, static or indexed. To make this identification, the port connection and port definition lists must be compared. If an error due to non-compatible connection and definition widths is detected at this stage, it must be reported. Some width problems result in warnings, and some result in fatal errors, both must be handled smoothly by the Verilog preprocessor. This task takes as input a list created by the AOI location task, and adds to this list the connection type, static or indexed, for each port in the port list.

Once the instances to be transformed have been identified, and the nets passed into each port classified as either indexed or static, then the new file, with the array of feature instances iteratively specified can be generated. This unrolling is done in an process such that the Verilog preprocessor is called as many times as the maximal number of array of instances per input source file, see loop in Figure 3. The source files to the preprocessor are left unchanged. The new Verilog-XL compatible source files being generated are named by taking the file name of the input file and concatenating a ".#" on the end. The # can be any integer dependent upon how many iterations the preprocessor must perform to completely eliminate the array syntax. In the next version of VPP, the full unrolling of all AOI occurrences will be performed in a single pass.

Challenges in the Development of VPP

Some of the obstacles during the creation of VPP are quite subtle. Since the preprocessor is required to handle all existing Verilog syntaxes, all forms, even non-typical Verilog must be supported.

In the initial stages of building VPP a critical task was to identify the dynamic data structures used by the Berkeley parser to represent the digital system, and the addition of new pointers required specifically for the preprocessor. The structure used by the Berkeley parser makes heavy use of three types of storage formats, standard dynamic memory allocation using pointers, unions and structures, the ST toolbox for symbol tables, and the list toolbox for linked lists. Both of these toolboxes were developed in C at Berkeley and provide an easy interface to standard data structures. Once the task of familiarization with the dynamic circuit configuration was complete, additional structures and pointers were required to be added to those already existing for

specific support of the preprocessor.

The Berkeley parser, as noted, did not support the usage of hierarchically named nets in Verilog source code. The ability to specify a net via a hierarchical name allows the user to access a net declared in another module without explicitly passing the net through the module parameter list. The specification and grammars were present in the YACC source code, but the supporting C functions for these grammars were incomplete. Hierarchically named nets are a widely used capability of the Verilog language, however, and, consequently, VPP must support them, and have the ability to determine the width of hierarchically named nets to function in the manner desired. For this reason the C functions to support the grammars were written. The functionality and usage of hierarchically named nets is described in section 12.5 of the OVI LRM 2.0a, and is also present in Verilog-XL compatible Verilog. Providing this functionality in the preprocessor required not only the addition of hierarchical name parsing in LEX and YACC, but also identification of top level modules, and the ability to follow the module tree structure down to verify that the net exists, and determine its width. This capability also implies that the port connection widths can not be determined until the full Verilog source is parsed.

The next challenge encountered in the generation of VPP was the support of explicitly named port connections in unrolling the array of feature instances syntax. The specification for using explicitly named port connections is given in section 12.4.4 of the OVI LRM 2.0a, and is supported in Verilog-XL compatible Verilog [14]. The routines written to analyze the port connections for an array of instances operate on the ordered relationship between the port definitions and the port connections. Allowing for unordered access, i.e., using explicitly named port connections, considerably complicates the process of analyzing the port connections. The named port connection capability required a new set of functions at the stage in the preprocessor where the array of instances declaration is "unrolled" into multiple instances of the repeated feature. This new functionality for unrolling is required because some Verilog software packages require port connections to be made using the named syntax, and will fail to function even if the port connections are reordered into the correct sequence by the preprocessor.

The functionality to determine the width of nets was the next challenge in the generation of VPP. The determination of net width is required for the classification of port connections as indexed or static. Identification of net width is a significant task due to the variety of methods by which a net can be both declared and referenced in Verilog. These methods include named, bit selected, concatenated, hierarchically named and standard nets. In the process of adding the net width functionality to the Verilog preprocessor, variables were added to the dynamic circuit representation created by the Berkeley parser. These variables insure that the time consuming task of width determination is only carried out once for each port connection made in the circuit description.

The most difficult task in the process of transforming from the array of feature instances representation to the iterative representation is in the generation of the sub-nets when unrolling indexed net port connections. This task involves taking an N-bit wide indexed net and subdividing it into P (N/P)-bit sub-nets. Because of the wide variety of methods that Verilog supports for net definition, the task requires a substantial amount of logic to perform. Both this task, and the similar net width determination task must be handled recursively due to the possibility of concatenated nets. The bit-selection task makes extensive use of the net-width determination functions.

Future Extensions to VPP

With a preprocessor created to support the Open Verilog International array of instance syntax, a simple question is what other grammars could be added to help make the Verilog language easier to use. Another extension to the Verilog language specification which is present in the OVI LRM 2.0a is the addition of parameterized macros. This capability allows the designer to specify parameterized macros using syntax similar to the C programming language for use in Verilog source files. This is one possible extension which is being examined for addition to VPP.

Also useful in the creation of Verilog source files would be the addition of syntaxes to support not only single dimensional arrays of instances, but also two dimensional arrays or tree shaped repeated instantiations of features. However there is no widely agreed upon syntax for either of these structures. The functionality of these structures can be effected using the AOI syntax, however having a specific grammar for support of these structures could make the source code more descriptive for the reader.

Conclusions

The creation of VPP, a Verilog preprocessor provides an important tool for HDL research. This tool will provide a base for continuing research into the Verilog HDL at the University of Michigan.

One unique feature of doing this type of analysis at the parsing level is that it allows the hardware design

engineer to examine the file which is generated by the preprocessor, to learn how the code is represented in the postprocessed form. As mentioned earlier, this also allows for use of all existing Verilog CAD tools without requiring modification to their algorithms.

It is our hope that by releasing this tool into the public domain we can increase Verilog users awareness that the Verilog language is not static, but constantly changing. In doing this we may be able to prompt these users into voicing their opinions about what additions to the Verilog language they would find useful.

References

[1]. Navabi, Zainalabedin, "A High-Level Language for Design and Modeling of Hardware", Journal of Systems Software, Vol 18, 1992, p. 5-18.

[2]. Goel, Prahu, "Maturing of the HDL Methodology", Electronic Engineering, Vol 63, Num 777, Sept 1991, p. S15.

[3] Personal communication (email) with ovi@netcom.com, dated Tue, 6 Sep 1994 11:08:48, message id : <199409061808.LAA21840@netcom15.netcom.com>.

[4] Goto, G., et al., "A 54 X 54-b Regularly Structured Tree Multiplier", IEEE JSSC, Vol. 27, No. 9, pp. 1229-1236, September 1992.

[5] Santoro, "Design and Clocking of VLSI Multipliers", Palo Alto CA: Stanford University Technical Report CSL-TR-89-397, October 1989.

[6] Article #625 from comp.lang.verilog, written by jws@chronologic.com.

[7] Article #627 from comp.lang.verilog, written by steveg@cadence.com.

[8] Article #630 from comp.lang.verilog, written by george@ole.cdac.com.

[9] Article #652 from comp.lang.verilog, written by davidr@cadence.com.

[10] Article #674 from comp.lang.verilog, written by jws@chronologic.com.

[11] Article #675 from comp.lang.verilog, written by robertb@cadence.com.

[12] Article #693 from comp.lang.verilog, written by leung@storage.tandem.com.

[13] Personal communication (email) with jws@chronologic.com, dated Sat, 27 Aug 94 16:19:48 PDT, message id : <9408272319.AA00589@chrnlgc.chronologic.com>.

[14] Verilog Hardware Description Language Reference Manual (LRM) Version 2.0, Los Gatos, CA: Open Verilog International, March 1993.

[15] Electronic Engineering Times, "Verilog Users Demonstrate Strong loyalty", Issue 788, pp. 1, 41, 45-47, Mar. 14, 1994.

[16]. IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987, The Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1988.

[17] Article #593 from comp.lang.verilog, written by jwill@netcom.com.

[18] Personal communication (email) with roberts@Cadence.com (David Roberts), dated Wed, 2 Mar 94 16:36:08, message id : <9403022136.AA11475@tweety>.

[19] comp.lang.verilog newsgroup frequently asked questions (FAQ), posted regularly to newsgroup, available via ftp@ftp.cray.com, directory : /pub/comp.lang.verilog/.