A Conference for EE Designers, ASIC and CAD Professionals

PROCEEDINGS 1995

Sponsored by:

Open Verilog International

IEEE COMPUTER SOCIETY®

In cooperation with:

EIAJ

Electronics Industries Association, Japan

IEEE Computer Society Press

A Verilog Preprocessor for Representing Datapath Components

Brian T. Davis and Trevor Mudge

University of Michigan
Department of EECS - Advanced Computer Architecture Laboratory

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 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. 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. The paper concludes with a status report on the preprocessor and thoughts for future development.

1 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 using an unambiguous textual syntax. This precise description can then serve as an entry point into the design process, as well as being a basis for design validation. The HDL circuit specification can also serve as documentation for the digital system [1]. In fact, it was for documentation of digital systems that the predecessors of today's modern HDLs were first employed.

Today, there are two primary HDLs in common commercial 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) in April of 1991. OVI was created as a private corporation responsible for maintaining the Verilog language specification [3]. Recently OVI transferred this responsibility to the Institute of Electrical and Electronic Engineers (IEEE). Currently IEEE working group 1364 is developing the first IEEE version of the Verilog HDL language specification.

VHDL is an alternative HDL developed out of the U.S. Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980 [4]. VHDL is an ADA based hardware description language which is commonly said to be more flexible than Verilog, and which already contains in it's specification and syntax many of the capabilities being discussed in this paper.

2 Need for a Repeated Feature Extension

Computers are a tool for increasing the productivity of engineers. They allow the user to perform a number of tasks which would require large amounts of time without the use of a computer, such as the preparation of this paper. One of the ways in which a computer does this is to perform, for the user, repetitive tasks which have little or no variation, and can be specified in one simple directive. This use for computers is exactly what is being talked about in the discussion of a syntax for repeated features within Verilog. If a large set of repetitive tasks for the computer to perform can be easily specified, then there is no reason to require a human user to explicitly perform each of these tasks, occupying user time. Not allowing a user to specify many tasks via an iterative construct is in

This work was supported in part by the Advanced Research Projects Agency under ARPA/ARO Contract Number DAAH04-94-G-0327

0-8186-7082-7/95 $04.00 © 1995 IEEE

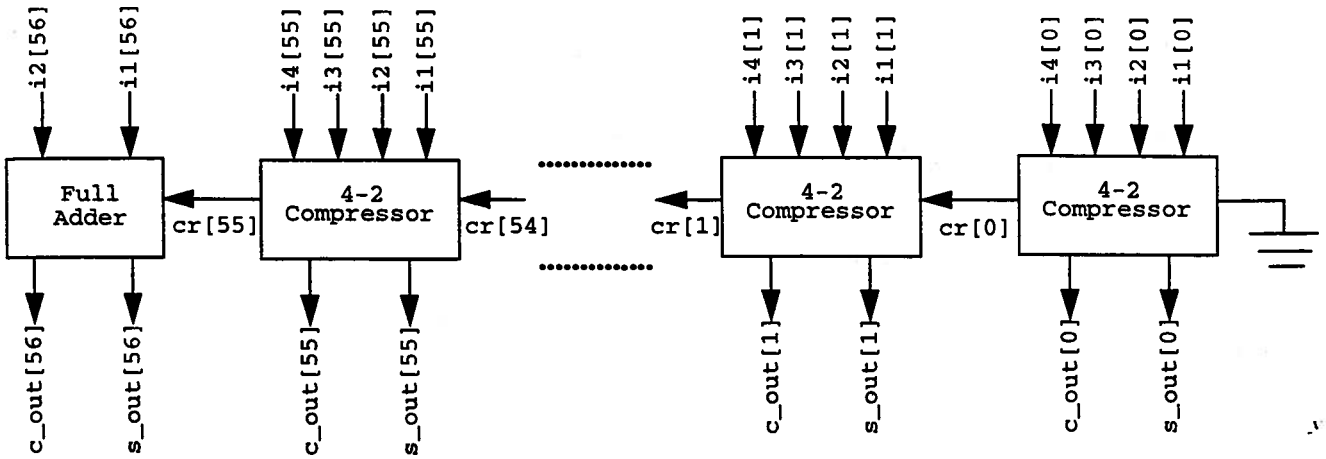

Figure 1. Row of a full multiplier array

direct conflict with the goal of utilizing a computer to increase user productivity.

The following example illustrates the need for a repeated feature syntax in Verilog. Figure 1 shows part of a multiplier tree [5]. The first and second levels of carry-save reduced partial products in the multiplier have already been generated, and the required functionality is to compress four partial products from the second level into two carry-save compressed partial products in the third level. These third level signals are then in turn fed into another array of compressors. After multiple levels of compressors, determined by the number of bits in the operands, the product is formed with a final carry-propagate binary adder [5].

The source code to describe the single row of the full multiplier array shown in Figure 1, using Verilog-XL compatible syntax, is given in Example 1. The code segment in Example 1 contains two module types, both of which are used in both Example 1 and 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, all of equal significance and create two carry bits of next higher significance and one bit of the input significance. The four-two compressor has a characteristic which 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 rippling for the row being used as an example to be to be eliminated [6]. 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 about this example are that the structure is highly repetitive and it would be time consuming for the user to enter the Verilog code representing this row. The complete description for the multiplier tree, with multiple variations on each row, is even worse. Clearly, any way to specify repeated features would simplify the expression of such structures.

```
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
```

Code Example 1. Multiplier row module using Verilog-XL Syntax

Extending the Verilog specification to support a repeated feature instance syntax will allow hardware designers to specify multiple instances of a gate, module or primitive in a single statement. This is especially useful in the specifications for datapaths where a bit-slice feature or module, like the four-two compressor in the example above, spanning many bit positions in an extremely regular fashion can be instantiated across all bits of a datapath. This full datapath in a single statement concept 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. Even this is dangerous, as it is very easy for the designer to make an error when performing the cutting and pasting by not changing a pasted value. Additional advantages of a succinct repeated feature syntax beyond simply a time savings for designers are an increased degree of readability and a reduced likelihood of design errors.

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. This desire has been illustrated by the sheer number of articles in the comp.lang.verilog Usenet newsgroup which make reference to, or discuss this issue [7][8][9][10][11][12][13]. As mentioned earlier, IEEE working group 1364 is currently considering its 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 in the working group [7][11][14]. Before 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, OVI Language Reference Manual (LRM) version 2.0a [15]. 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 [16]. This article reached no definitive conclusion, but it is worth note that the concern over a repeated feature syntax is widely recognized enough that it made it's way into publications such as Electrical Engineering Times.

3 Proposed Syntaxes

The array of feature instances syntax, specified in OVI LRM version 2.0a [15], is a direct extension to the Verilog language. Within this syntax, an instance of repeated features, where a feature can be a gate, primitive or module, 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. The difference illustrated here does not appear as significant as it truly is due to the ellipses used in Example 1. However this simple extension has a tremendous impact upon the size of the Verilog source code required for the description of the multiplier in [5].

```
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   C56(i1[56], i2[56], cr[55], s_out[56],
      c_out[56]);
endmodule
```

Code 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 unconditional feature instances. This is due to the fact that the number of features 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 to an array of feature instances are given in the OVI LRM 2.0a as [15]:

- 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.

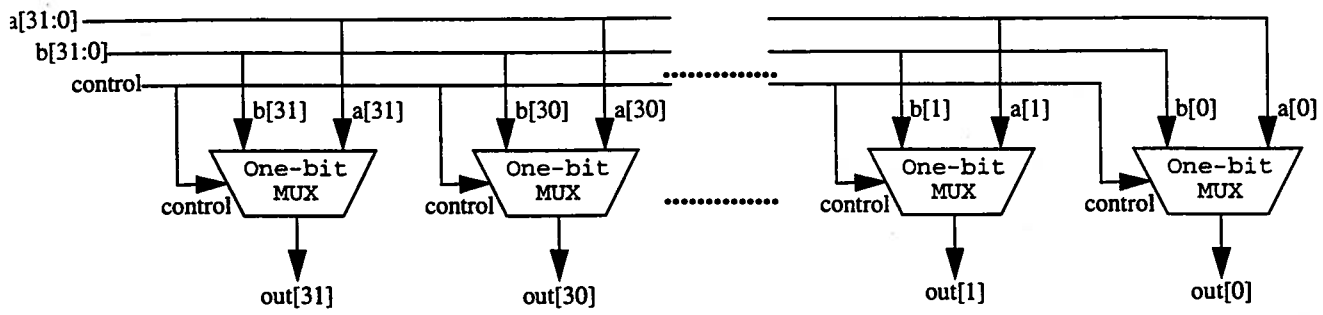

Figure 2. Array of feature (MUX) instances

- 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 meaning of each of these four rules will be described using an example. The example to be used is illustrated in Figure 2, and described in Example 3. Figure 2 shows a 32 bit wide MUX, a typical datapath element. Example 3 describes the same MUX utilizing a one-bit MUX which is presumed to be defined elsewhere, but is self explanatory.

```
module mux32b(a, b, control, out);
input [31:0] a, b;
input control;
output [31:0] out;

mux1b mux_bit[31:0](a, b, control, out);
endmodule
```

Code Example 3. Array of feature (MUX) instances

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 tool as being a net of the same width as the definition for the port into which the net is being passed. An example of a static net is the control line being connected to each bit slice of the 32 bit wide MUX shown in Figure 2, and illustrated in Example 3.

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, referred to as a part or bit select, is connected to each feature instance declared in the array. An indexed net is recognized by the Verilog tool 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, and described in Example 3.

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 made starting from the right and 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 part select being passed into each feature contains multiple bits, but our examples in this paper 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 [15].

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 for Verilog similar in function to the generate syntax used within VHDL [17]. 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 items as scoped declarations and multidimensional arrays to the Verilog language [10]. 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, upon which Verilog is based, and behavioral Verilog which currently allows the use of a for loop. It was determined that the defining 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. Supporting an interpreted for loop syntax using the preprocessor was not pursued because the functionality of an interpreted for-loop syntax for structural Verilog would be similar to that of the OVI AOI syntax, but less compact. Additionally, only a few functionalities could be supported by a for loop syntax, but not by the AOI syntax. Examples of these limitations are loops which would involve both behavioral and structural Verilog within the same for loop, or use of a variable within the indexing of the loop. There is no way in which a preprocessor could support utilization of variable within the indexing. The final possible syntax for support of repeated features which was examined was a metaprocessed for-loop structure which is currently in use at some design facilities. This involves using a programming language such as C, or a text manipulation language such as PERL to generate the Verilog source code from an intermediate representation before it is passed into a Verilog tool [18]. This avenue was not pursued due to its temporary fix appearance and the requirement of an additional step before parsing.

Any of these proposed syntaxes could provide the functionality desired by the hardware designers requesting support for iterative constructs. We chose to support the OVI array of feature instances syntax in the preprocessor 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, ports and registers. 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 [11][14][19]. 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 tools.

4 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 rendered useless. Our solution was to build a Verilog preprocessor, VPP. This preprocessor will accept the OVI LRM 2.0a array of instances extension as part of a Verilog source file, and for each source file it examines containing this extension, it will generate a new source file which utilizes only Verilog-XL compliant 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 Usenet newsgroup, there are two Verilog parsers in the public domain, both of which utilize LEX and YACC for their parsing capabilities [20]. 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. This parser was received from Mike Riepe (riepe@eecs.umich.edu) who performed a significant task in separating the parsing functionality from the BLIF-MV conversion code, and repairing memory leaks in the original tool. 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 a Cadence supplied parser. After analysis of the source code for the two public domain parsers, the Berkeley parser was selected because it was the furthest along towards completion.

The Berkeley parser in the state received 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, but the underlying C functionality to support the parser was not present. The format for the dynamically stored representation of the described circuit was specified, and for the most part finished. 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. C code had to be generated for the majority of these 'TODO' markers before the preprocessor was fully functional.

The flow-chart for the Verilog preprocessor is given in Figure 3. The preprocessor accepts as input Verilog source file(s), which may or may not contain the array of feature instances syntax, parses these file(s), and outputs to disk new Verilog-XL compatible file(s) which will retain the functionality of the input source file(s). These output file(s) can then be compiled by Verilog-XL, or an equivalent Verilog HDL simulation package and simulated or used in the normal manner.

One difference between the preprocessor output and the OVI LRM 2.0a specification is unavoidable. This is because in Verilog-XL compliant syntax, 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 features instantiated by the AOI statement are given the same name, followed by square brackets containing a number. 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 as opposed to widget[K] as specified in the OVI LRM 2.0a. This renaming convention should keep the debugging process intuitive to the designer, and minimize the differences between the OVI specification and the preprocessor output.

5 Creating the Preprocessor

A variety of tasks were involved in the generation of the preprocessor. Many 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 compliant source code, and items which were required to be added for support of the new OVI AOI syntax. All changes to the code were performed using ANSI C as well as LEX and YACC. One of the next steps in the evolution of VPP as a tool is to port this to GCC, FLEX and BISON respectively for greater consistency across hardware platforms.

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, critical tasks were the identification of 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 source representation used by the Berkeley parser makes heavy use of three types of storage formats: standard dynamic memory allocation using pointers to 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 in a similar manner as the standard classes provide a easy interface to these same data structures in C++. Once the task of familiarization with the internal circuit representation was complete, additional structures and pointers for specific support of the preprocessor were required to be added to those already existing.

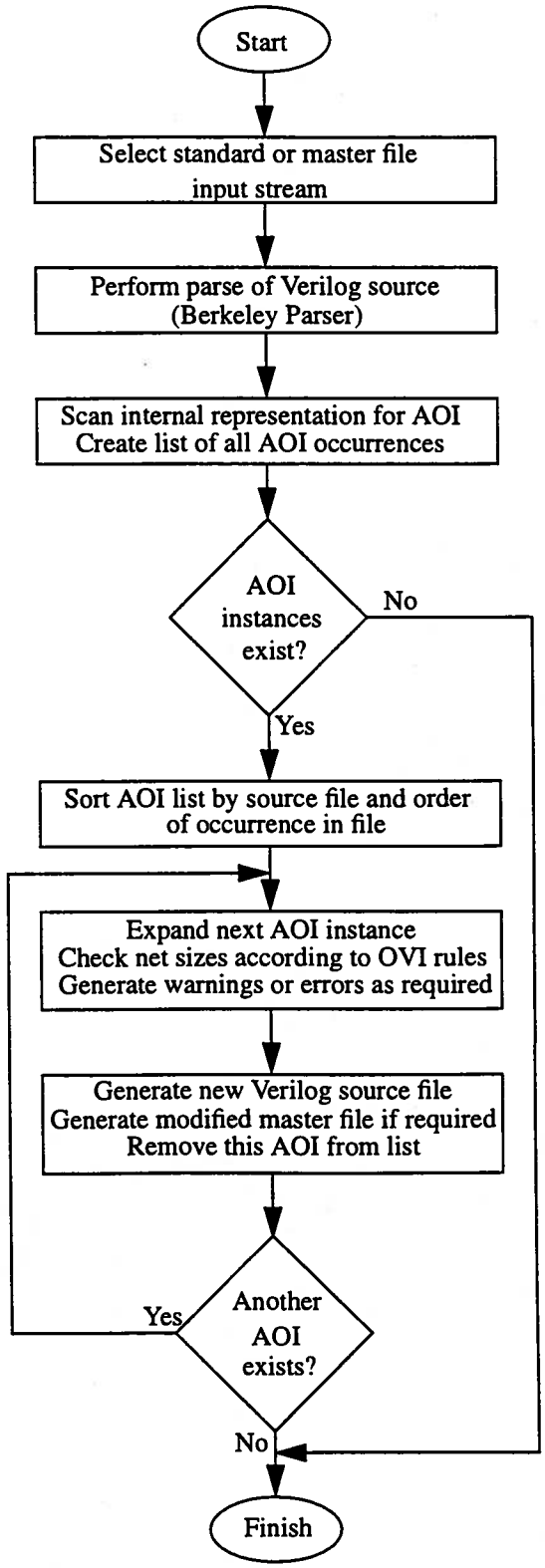

Figure 3. Flowchart for VPP

The primary change to the Berkeley parser was the addition of grammars to allow for an AOI statement to be successfully parsed rather than generating an error. This

modification involved specifying new YACC grammars, as well as writing the C functions to handle these grammars when they are encountered. The Berkeley parser is essentially a two pass parser. The source is read in, and the structures are allocated in the first pass. During the second pass a number of characteristic value are generated, checks for errors are made, and the internal representation may be printed dependant upon the command line arguments. The addition of the AOI functionality required significant changes to the second pass of the parser in all areas mentioned. It was also convenient to perform the net width determination function for all net references during this second pass. The addition of variables to the dynamic representation, for the storage of the new information contained in the AOI syntax, was also required.

The next change required to the Berkeley parser was the ability of the parser to recognize hierarchically named nets. The Berkeley parser, in the condition received, did not fully support the usage of hierarchically named nets in Verilog source code. The specification and grammars were present in the YACC source code, but the supporting C functions for these grammars were not present. The ability to specify a net via a hierarchical name allows the user to access a net declared in another module or source file without explicitly passing the net through the module parameter list. Hierarchically named nets are a widely used capability of the Verilog language, 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 for hierarchical nets 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 [15]. Providing this functionality in the preprocessor required identification of top level modules. Also the ability to follow the module tree structure down to verify that a specified net exists, and determine its width. This hierarchical net capability implies that the port connection widths can not be determined until the full Verilog source is parsed. For this reason the width determination for all net references is performed in the second the second pass, as mentioned earlier.

Another task 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 also supported in Verilog-XL compatible Verilog [15]. 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 statement is "unrolled" into multiple instances of the repeated feature. This new functionality for unrolling named port connections is required because some Verilog CAD tools require port connections to be made using the named syntax, and will fail to function if the port connections are not made using the named port convention, regardless of whether the connections are reordered into the correct sequence.

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, this task requires a substantial amount of logic to perform. Both this task, and the similar net width determination task must allow for recursion due to the possibility of concatenated nets.

One of the final, and more straight forward tasks achieved in the generation of the preprocessor was the addition of support for the use of master files in a Verilog project environment. The master file capability, another commonly used feature of Verilog-XL, was not supported by the Berkeley parser as received, and therefor it was required that C code to provide this functionality be generated. Master files are used where a Verilog project contains more than a single source file. The master file functionality allows the user to specify a set of Verilog source files which all taken together represent a digital circuit design. The master file also allows the user to specify whether each Verilog source file referenced in the master 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 files works in close conjunction with the Berkeley parser

without requiring changes to the parsing algorithm, and essentially provides an input stream to the LEXer.

6 The Preprocessing Stages

A necessary task in the creation of a preprocessor to transform one notation (the array of instances syntax) into another notation (the Verilog-XL compliant 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 feature instance name has a range specifier ('[a:b]') after it. The code to locate all places in the source code where the array of instances syntax is used, and create a list of all AOI statements was written. This is the first segment of code to execute after both passes of the parse of the Verilog source code is complete. If the program is unable to locate any statements utilizing the AOI syntax, then the preprocessor terminates.

If the list of statements utilizing the AOI syntax is not empty, then it this list is sorted. This sorting is done in order to simplify the later unrolling process. All AOI instances are sorted into a single ordered list using the source file in which they are contained as the first key item, and the reverse of the order in which they occur in this file as the second key.

After an instance of code utilizing the AOI 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 phase uses as input the sorted 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. This list is then used in the process of unrolling each of the AOI statements.

Once the instances to be transformed have been identified, and the nets passed into each port have been classified as either indexed or static, then the new file, with the array of feature instances iteratively specified can be generated. 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 # is a single digit integer dependant upon how many AOI statements are present in the source file being renamed.

The final step in the execution of the preprocessor is the generation of a new master file. This step occurs only if a master file was used for the input. Much like the Verilog source files, the original master file remains untouched, and a new master file is generated. This new master file contains all of the changes to the Verilog source file names which were made by the preprocessor such that running a Verilog CAD tool on the generated master file will reflect all changes to all source files in the project. The status of a source file, as source or library, also remains unchanged in the new master file. The name of the created master file is provided to the user at termination of the preprocessor.

7 Future Extensions to VPP

With a preprocessor created to support the Open Verilog International array of feature instances 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, and the functionality of these structures can be effected using the AOI syntax. For instance a two dimensional array of 2 elements by 2 elements can be represented as a one dimensional array of 4 elements, using the concatenated net description in Verilog. However having a specific grammar for support of these structures could make the source code more descriptive for the reader.

Essentially this preprocessor could be used for prototype testing of any new syntax which is being considered for addition to the Verilog language. This is the function it has served for arrays of instances, and now that the initial learning curve has been overcome, and the preprocessor is in place, the addition of new syntaxes to this tool would be relatively simple.

8 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 same users into voicing their opinions about what additions to the Verilog language they would find useful.

Acknowledgments

We would like to thank all of the people on the Internet who answered our questions during the creation of the Verilog preprocessor tool. Without their assistance and advice, this tool would still be in the conceptual phase.

We would also like to acknowledge once again Szu-Tsung Cheng for the development of a public domain Verilog parser. This LEX and YACC framework was the starting point for the Verilog preprocessor, and without this parser available, the code generation time for the preprocessor would have been at least doubled.

Much thanks also goes to Mike Riepe for his many roles in the generation of this preprocessor. Primarily for his work in preparing the code of the Berkeley parser for use in other tools, such as this preprocessor. Also for his expert advice while becoming familiar with the parser, and input into the appropriate functionality for this preprocessor during the development cycle.

Where to find VPP

The source code for the Verilog preprocessor, VPP is available via anonymous ftp at *ftp.eecs.umich.edu*, in the directory *people/btdavis*. Any questions about this source code should be directed to *btdavis@engin.umich.edu*.

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, Tue, 6 Sep 1994 11:08:48.

[4] Ashden, Peter J., "The VHDL Cookbook", University of Adelaide, South Australia, ftp@ftp.cs.adelaide.edu.au:/pub/VHDL-Cookbook.

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

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

[7] Comp.lang.verilog article #625, written by jws@chronologic.com, Wed, 16 Feb 1994 17:16:07.

[8] Comp.lang.verilog article #627, written by steveg@cadence.com, Thu, 17 Feb 1994 14:58:01.

[9] Comp.lang.verilog article #630, written by george@ole.cdac.com, Fri, 18 Feb 1994 18:11:16.

[10] Comp.lang.verilog article #652, written by davidr@cadence.com, Wed, 23 Feb 1994 19:09:53.

[11] Comp.lang.verilog article #674, written by jws@chronologic.com, Sat, 26 Feb 1994 22:25:18.

[12] Comp.lang.verilog article #675, written by robertb@cadence.com, Mon, 28 Feb 1994 19:48:59.

[13] Comp.lang.verilog article #693, written by leung@storage.tandem.com, Wed, 2 Mar 1994 17:33:11.

[14] Personal communication (email) with jws@chronologic.com, Sat, 27 Aug 94 16:19:48.

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

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

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

[18] Comp.lang.verilog article #593, written by jwill@netcom.com, Wed, 9 Feb 1994 08:20:00.

[19] Personal communication (email) with roberts@cadence.com (David Roberts), Wed, 2 Mar 94 16:36:08.

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

Comp.lang.verilog article numbers are as archived at ftp.cray.com