Technical Report

UCAM-CL-TR-485
ISSN 1476-2986

Number 485

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Three notes on the interpretation of Verilog

Daryl Stewart, Myra VanInwegen

January 2000

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-485*

Three notes on the Interpretation of Verilog.

Daryl Stewart and Myra VanInwegen

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
England

January 28, 2000

Abstract

In order to simplify the many constructs available in the Verilog Hardware Description Language two methods were used to normalise code before analysis, *scalarisation* and *hierarchy flattening.*

A method for scalarising Verilog expressions is described and the replacement of expressions with scalarised versions is considered. This then forms the basis of an implementation of Verilog expression evaluation and normalization.

The organisation of hierarchical designs is described and an algorithm for flattening designs is derived from this.

1 Introduction

This technical report comprises three papers written for the research project "Checking the Equivalence between Synthesised Logic and Non-Synthesisable Behavioural Prototypes".

The two subjects covered here, *scalarisation* and *hierarchy flattening* , were used to transform large hierarchical design descriptions with busses into single module descriptions with only scalar registers. This step represents a move from an abstract design towards a more concrete, real world version, since the realisation of a circuit can be thought of as a single mass of circuitry with many single bit registers (flip-flops or memory cells) and single wires.

The first two papers deal with the subject of expression semantics in the Hardware Description Language "Verilog". The dominant problem addressed here is how Verilog interprets the rules for expressions of various bit-widths.

Part I describes a scalarising algorithm and considers the use of scalarised expressions within Verilog code with respect to the simulation cycle differences they could introduce.

Section 6 of part II tackles expression evaluation and describes the implementation of an expression evaluator based partly on the earlier work. Section 7 covers the normalization of expressions within Verilog code which allows later analysis of the code to be simplified.

The third paper describes an algorithm for elaborating hierarchical verilog and producing a single flat module describing the same design, as used in the research project. The consequences of removing module boundaries include the modelling of port connections and identifier renaming across different module instantiations.

Decisions about how Verilog expressions should be evaluated were affected by several sources which are acknowledged within the papers themselves.

2 Acknowledgements

The work described here comprised various parts of the tools developed within the EPSRC project "Checking the Equivalence between synthesised logic and non-synthesisable behavioural prototypes" and was supported by EPSRC grants GR/K57343 and GR/L35973.

Part I
Scalarising Verilog Expressions

by Daryl Stewart

Here, we present a description in a denotational semantics style of an algorithm for converting Verilog expressions into a scalar form. The scalar form version of an expression should be interchangeable with the original vector form version with no discernable difference in simulation behaviour. Section 3 outlines several factors which make this difficult in practice.

The algorithm is described in a bottom-up fashion in section 4, starting with definitions of functions on the algorithm's data structures, and ending with a denotational style definition over the concrete syntax of expressions.

Finally, in section 5, we show how to incorporate the scalar expression list generated by the algorithm into the various syntactic categories which contain them.

The algorithm was constructed from a study of [5] and from experiments with Verilog-XL from Cadence Design Systems Inc. [7], the Viper/free simulator from interHDL [9] and the Veriwell simulator from Wellspring Solutions Inc. [8]

3 Representing Vectors as Scalar Expressions

Scalarisation forms a part of any synthesis process which produces bit-level descriptions of hardware. For example, the Verilog code

```
reg [1:0] a, b, q;
   initial q = a & b;
```

should synthesise to produce two 1 bit and gates, perhaps as described by:

```
reg a_1, a_0, b_1, b_0, q_1, q_0;
   and (q_1, a_1, b_1);
   and (q_0, a_0, b_0);
```

where there are no busses, only 1 bit registers.

We wish to produce scalar expressions, not instantiations of gates linked by wires, so a suitable form for the output of our algorithm may be:

```
reg [1:0] a, b, q;
   initial begin
      q[1] = a[1] & b[1];
      q[0] = a[0] & b[0];
   end
```

if we allow bit selects of vectors on the basis that they are 1 bit values.

3.1 Atomicity of Multiple Scalar Assignments

We soon realise that the above code is not interchangeable with the original. Consider `r = ^ q;` being executed somewhere in parallel with the above code. If we have a state where $q = 0$, $b = 3$ and $a = 3$, then in the original code, `q = a & b;` is executed *atomically* and $q = 3$ follows immediately, hence `r = ^ q;` sets $r = 0$ whether it executes before or after the assignment to `q`. If, however, the assignment to `r` executes in-between the two seperate blocking assignments in the scalarised version, then we will have a state where $q = 2$, leading to $r = 1$, a completely unreachable state given the original code.

In order to enforce the same atomic assignment behaviour on the scalarised code as on the vector code, we allow a special form of vector expression to be present in scalarised code - Verilog's *concatenation* operator, denoted by a list of expressions enclosed in curly braces. Our condition on the use of this operator from within scalarised code is that every expression within the concatenation is a truly scalar expression, and not a further concatenation. This gives us:

```
reg [1:0] a, b, q;
initial { q[1], q[0] } = { (a[1] & b[1]), (a[0] & b[0]) };
```

The only change here is in the way we have incorporated the list of scalar expressions into the surrounding concrete syntax of the assignment operator. In other words, we can consider the transformation of vector expressions into a list of scalar expressions without worrying about the use of the final result in a concatenation or otherwise. Hence our expression scalarising will only produce expressions made up from scalar expressions and not concatenations and we can be sure to meet our condition on the use of concatenations.

3.2 Contextual vs. Self Determination of Expression Width

Apart from the problem with maintaining atomic assignments, there are some subtle features of the Verilog expression evaluation semantics themselves. When evaluating an expression or sub-expression in Verilog, there is a need to know how wide the resulting expression will be. Every subexpression has a minimum width which is required to represent its result fully, but it must also be prepared to fill a larger, requested width. In general, we can fill extra bits by padding to the left with zeros, but this must be done at the correct subexpression level.

On the occasions when an expression is required to extend itself, the expression is said to be *context-determined.* Subexpressions of certain operators are context-determined, for example in bitwise binary operators, since they will be combined with another value on a bit by bit basis, so they need to match it in length. In assignments, the rvalue must be able to fill the full length of the lvalue, so such expressions are also context determined. The case statement is also defined such that the case expression and all the switch expressions are evaluated to the length of the longest among them.

In tables 6 and 7, functions CONTEXT and CONTEXTPAIR are used when a subexpression must match some context provided width. SELF is used for expressions whose width is solely determined by the expression itself. This is explained further in section 4.

Consider the following example, which contains two such contextual length evaluations:

```
reg [15:0] q;
reg [4:0] a;

initial
   begin
      a = 12;
      q = -8'd1 & -a;
   end
```

The first assignment has an integer on its right hand side. Although integers are 32 bits, only the last five are required and a is left holding ($5'b01100$).

The second assignment's first context arises from q being 16 bits, requiring the right hand side to provide 16 bits. The bitwise and has two operands, one of 8 bits and one of 5 bits, so it would normally evaluate each of these subexpressions to at least 8 bits - this is the second context. However, if this were the case then we would see the 8 bit result extended with 8 zeros. In fact, Verilog assigns the value ($16'b1111111111110100$) to q. This implies that both the operands of the bitwise and had been evaluated to 16 bits with 1's in their top 8 bits.

In the rule for $[\![\ e_l\mathbf{binop}e_r\ ]\!]$ (table 6), we see the MAXINDEX function is used to determine the context due to the operation, but this is not the value used in evaluating the subexpressions. Instead, the presence of CONTEXTPAIR forces the enclosing environment (be it an expression or, as in our example, an assignment statement), to either use this value, or to provide its own - which must be the same or larger.

3.3 Use of Signed Arithmetic with Integer Data Types vs. Unsigned Arithmetic with Reg Data Types

In Verilog there are two types for numeric values, *integer* and *reg*. Any arithmetic involving the former should use signed arithmetic, otherwise operands are treated as unsigned two's complement values.

In practice, this does not matter with addition and subtraction, since adding two's complements produces the correct results with respect to full signed arithmetic.

3.4 Arithmetic Carries

Scalarising arithmetic does have one major effect, due to the calculation of carries. A naive algorithm for scalarising addition could involve recalculating the whole carry chain for each successive bit of the result.

For two equal length lists of scalar expressions, $e_l = [l_n, ..., l_0]$ and $e_r = [r_n, ..., r_0]$, then ADD $e_l\ e_r = [a_n, ..., a_0]$ where

$$a_j = l_j \wedge r_j \wedge c_j$$

and

$$c_j = \begin{cases} 0 & \text{if } j = 0 \\ (r_{j-1} \ \& \ l_{j-1}) \mid (c_{j-1} \ \& \ (r_{j-1} \mid l_{j-1})) & \text{otherwise} \end{cases}$$

Notice that this definition matches Verilog's behaviour of *not* returning an extra most significant bit for the leftmost carry. If this is required, the subexpressions should be evaluated to a larger width, and ARITHPAIR allows this due to its use of CONTEXTPAIR.

The advantage of this is that the scalarisation can maintain the atomic assignment semantics. The disadvantage is that c_j appears in a_k for $k \leq j$ leading to scalar expression lists of size $\mathcal{O}(n^2)$.

A better approach is to allow the insertion and use of extra carry wires, which are continually assigned, or carry registers, which are assigned immediately before the expression which uses them. This requires the ability to invent fresh wire or reg names. The result of applying ADD to a pair of scalar expression lists would then be a scalar expression list representing the answer, and a list of (wire name by scalar expression) pairs with the least significant bit's carry first.

We choose to describe the rest of the scalarising algorithm in a way which allows either of these methods, since the former is easier to follow, and the latter is easily constructed with a few modifications to it.

4 Formal Description of the Scalarising Algorithm

In this section we define the scalarising algorithm as it presently stands, in both a carry generating form and a non carry generating form. These two forms differ in only a few places and table 2 contains the modifications which produce the former from the latter.

Tables 6 and 7 define a function over pieces of concrete syntax in a denotational style which relies on the functions defined in earlier tables. The division between *expressions* and *primary expressions* is a characteristic of Verilog. Inspection of these two tables shows that the result of $[\![\ e\ ]\!]$ is not a scalarised version of e. Although not immediately obvious, the result is in fact of the type `integer * (integer` $\mapsto$ `expression list)`. As mentioned earlier (section 3.2), every expression has a minimum bitwidth required to represent it and the `integer` which is returned is known as the *most significant bit's index*. By convention, the least significant bit is given index 0, so the actual number of bits required is one more than the most significant bit's index. The `integer` accepted by the function is the index of the most significant bit desired in the result.

In table 1 we see that the functions INDEX and SIZE, when applied to $[\![e]\!]$, will return the most significant bit's index and the number of bits respectively. We also have MAXINDEX, which returns the largest index required from between two expressions, and CATINDEX, which returns the index for a list of expressions which are to be concatenated. Notice that this is defined in terms of SIZE, since the concatenation's index is one less than the sum of the sizes of the constituent expressions. CATSIZE is defined in terms of CATINDEX.

At this point it is worth noting that in the tables which follow CAPITALISED names are functions we define (e.g. SIZE, INDEX, FUNC), whilst lower case names are standard functions (e.g. map, fst, snd). Furthermore, Verilog operators are ranged over by names in `lower case typewriter` font, and functions which construct them are assumed to produce a subexpression of the obvious form. For example, `bitand` (e_l, e_r) produces the Verilog expression $e_l \& e_r$ and `bitnot` e produces $\sim e$. Notice the definition of a curried version of `cond`, CONDRESOP, in table 1.

Two special cases are `shiftop` and `compop`. Each possible operator for these categories has a special constructor function defined in the *associativity lists* given in tables 3 and 4. Where `shiftop'` is seen, the function paired with the appropriate shift operator in the associativity list is used. Similarly for `compop'`.

Continuing the theme of destructors defined over the results of $[\![e]\!]$, we have FUNC which returns the `integer` $\mapsto$ `expression list` function, followed by SELF, which applies the INDEX of $[\![e]\!]$ to the FUNC of $[\![e]\!]$. In this way we can denote operands with *SELF-determined bitwidth*. In other words, SELF $[\![e]\!]$ returns a list of scalar expressions fully representing e.

The remaining functions in table 1 are defined differently for a carry producing algorithm, as per table 2, which is described in section 4.1.

When an expression is to fill a bitstring which is wider than its own size, the function 0PAD can be used, which uses 0EXTEND to extend to the left with zeros by an amount equal to the difference between the expression's index (r) and the index required (p). Note that $p \geq r$ is required.

The more complicated version of SELF $[\![e]\!]$ is CONTEXT $[\![e]\!]$ f which applies the FUNC of $[\![e]\!]$ to some other index (p). It also applies f to the resulting list of scalar expressions, which allows us to express a function over the scalar version of a subexpression before the subexpression is evaluated. As an example of this, see the definition of $[\![\sim e]\!]$, which yields its function of

fun	FOLD f $[e]$	=	e
fun	FOLD f $e :: es$	=	$f(e, (\text{FOLD } f \ es))$
fun	COPYCAT es 0	=	$[]$
|	COPYCAT es n	=	es@(COPYCAT es $(n-1)$);
fun	ZERO	=	COPYCAT $(1'b0)$;
fun	INDEX $(i, \_)$	=	i;
fun	MAXINDEX (ee_l, ee_r)	=	max(INDEX ee_l, INDEX ee_r);
fun	CATINDEX ees	=	(FOLD (+) (map(SIZE)(ees))) $-$ 1;
fun	SIZE ee	=	(INDEX ee) + 1;
fun	CATSIZE ees	=	(CATINDEX ees) + 1;
fun	FUNC $(\_, f)$	=	f;
fun	SELF ee	=	(FUNC ee)($rmINDEX$ ee);
fun	CONDRESOP e_c (e_t, e_f)	=	cond(e_c, e_t, e_f);
fun	0EXTEND es p	=	(ZERO p)@es;
fun	0PAD es r p	=	0EXTEND es $(p-r)$;
fun	CONTEXT ee f p	=	f(FUNC ee p);
fun	CONTEXTPAIR(ee_l, ee_r) f p	=	f((FUNC ee_l p), (FUNC ee_r p));
fun	SELFPAIR $pair$ f	=	CONTEXTPAIR($pair$)(f) (MAXINDEX $pair$);
fun	ARITHPAIR $pair$	=	CONTEXTPAIR($pair$)(ADD);
fun	UNOP op ee	=	FOLD(op)(SELF ee);
fun	TRUTH	=	(UNOP `bitor`);
fun	CONSTANT ee	=	decimal_of_bit_list(SELF ee);
fun	CONCAT	=	flat;
fun	MULTICAT	=	COPYCAT;

Table 1: Function Definitions for Scalarising.

type `integer` $\mapsto$ `expression list` by using CONTEXT to wrap a mapping of `bitnot` up with $[\![\ e\ ]\!]$. Whatever index is passed to FUNC $[\![\ \sim e\ ]\!]$ is passed to FUNC $[\![\ e\ ]\!]$ *before* map `bitnot` is applied.

CONTEXTPAIR is an extension of CONTEXT to deal with pairs of Verilog expressions which must both be evaluated to the same length before having some function (f) applied to them. The function f will commonly involve using zip to produce a single list of expression pairs. See the definition of $[\![\ e_l$ `binop` $e_r\ ]\!]$ for a good example of this.

SELFPAIR is like CONTEXTPAIR, except that it requires no index. Instead it generates its own using MAXINDEX and evaluates two Verilog expressions to the length of the longest of them.

ARITHPAIR is used by $[\![\ e_l + e_r\ ]\!]$, which is the only place where carries can be created but in this form of the algorithm, no carries are generated. ADD is described in section 3.4.

For Verilog's unary reduction operators we have UNOP *op*, which evaluates a subexpression and then returns a single expression consisting of all the scalar expressions which represent the subexpression, joined by *op*. For this we use FOLD, a slightly non-standard version of the normal fold operation, which is undefined for empty lists. FOLD f $[x_n, x_{n-1}, ..., x_1, x_0]$ is $f(x_n, (f(x_{n-1}, ...(f(x_1, x_0))...)))$.

TRUTH is an alias for the reduction or ($|$) operator. `why?`

In the definition of CONSTANT we assume a function decimal_of_bit_list which produces an integer given a list of (constant) scalar expressions. If any of the scalar expressions passed to decimal_of_bit_list are not known constant bits ($1'b0$ or $1'b1$) it is treated as a static parsing error.

Finally CONCAT is defined to produce a list of scalar expressions from a list of scalar expression lists, and MULTICAT to copy such a list multiple times. These function names are really needed for the with carries form of the algorithm and have more complicated definitions in table 2.

4.1 Modifications to Provide Carries.

So far, the functions described are suitable for manipulating the results of the naive implementation of ADD from section 3.4. The alternative is to create a set of fresh *carry wires.*

In order to extract these wires and their associated expressions from the scalarisation of an expression e, we modify the type of the result of $[\![\ e\ ]\!]$ to `integer * (integer` $\mapsto$ `(expression list * carry list)`, where *carry* is a type suitable as an argument to the functions which construct Verilog's *continuous assignments* or *immediate assignment* constructs; i.e. it contains an Lvalue and a (scalar) expression. Here we show that implementing this requires changes to some of the constructor functions already defined while the denotational definition is unaffected.

Since only the final result's datatype has changed, from `expression list` to `( expression list * carry list )`, the first few constructors of table 1 are unchanged, as mentioned earlier. However, we need to consider functions which manipulate the result type more closely.

The first new functions in table 2 are destructors over the new datatype of our final result. The scalar expression list equivalent to e is returned by ANS (SELF $[\![\ e\ ]\!]$), and the new list of carries by CARRY (SELF $[\![\ e\ ]\!]$).

```
fun  ANS (es,_)            =  es;
fun  CARRIES (_,cs)        =  cs;

fun  0EXTEND ecs p         =  ((ZERO p)@ANS(ecs)),
                               CARRIES(ecs));
fun  0PAD ecs r p          =  0EXTEND ecs (p - r);

fun  CONTEXT ee f p        =  ((f ANS(FUNC ee p)),
                               CARRIES(FUNC ee p));
fun  CONTEXTPAIR           =  ((f(ANS(FUNC ee_l p),
       (ee_l,ee_r) f p             ANS(FUNC ee_r p))),
                               (CARRIES(FUNC ee_l p)
                                @CARRIES(FUNC ee_r p)));
fun  SELFPAIR pair f       =  CONTEXTPAIR(pair)(f)
                                          (MAXINDEX pair);
fun  ARITHPAIR pair p      =  (fn scalar_sum =>
                               ( (ANS(ANS(scalar_sum))),
                                (CARRIES(scalar_sum))
                                 @(CARRIES(ANS(scalar_sum))) )
                              )(CONTEXTPAIR(pair)(ADD)(p));

fun  UNOP op ee            =  (FOLD op (ANS(SELF ee)) (1'b0),
                               CARRIES(SELF ee));
fun  TRUTH                 =  (UNOP bitor);
fun  CONSTANT ee           =  decimal_of_bit_list(ANS(SELF ee));

fun  CONCAT ecs            =  ((flat(fst(unzip ecs))),
                               (flat(snd(unzip ecs))));
fun  MULTICAT ecs n        =  ((COPYCAT(ANS(ecs)) n),
                               CARRIES(ecs));
```

Table 2: Alternative Function Definitions for Scalarising with Extra Carry Generation.

0EXTEND is redefined to accept the new result type, and return a result with the original carries unchanged. Although the form of 0PAD is unchanged, it is repeated to make clear that it uses our new 0EXTEND function.

CONTEXT and CONTEXTPAIR are similarly redefined so as to work with the new result type. Again, despite having the same form as previously, we redefine SELFPAIR here for clarity.

The major change comes in the form of ARITHPAIR, which is the only place where new carries can be introduced. Since it combines two subexpressions and then applies ADD using CONTEXTPAIR, it will have to combine three lists of carries - one from each of the subexpressions and one from the result of the arithmetic it implements. Since this third list of carries is generated from the subexpressions, it may contain uses of their carries. Thus it is important that the subexpressions' carries are assigned first, hence the carry list from the CONTEXTPAIRing of the subexpressions occurs first in the concatenation of carries. (Notice that the type of CONTEXTPAIR(*pair*)(*ADD*)(*p*) is `((expression list * carry list) * carry list)`, where the first pair is the result of ADD, and the last list of carries is a concatenation of the subexpressions' carries.)

UNOP, TRUTH and CONSTANT are all redeclared to make use of the new result type, whilst leaving the carries unaffected.

The new CONCAT makes use of flat and unzip to produce a pair of lists from a list of pairs, and MULTICAT simply copies the expressions in the result it is passed a given number of times.

4.2 Denotational Definitions over Expressions

Having described the functions used to compose the denotational definitions, we can now explain the meaning of tables 6 and 7.

Remembering that each right hand side comprises a pair *index* * *function*, we see that operators which return a single bit, such as logical operators and reduction operators, have 0 in the first place, whilst bitwise operators return some function of the index (indices) of their subexpression(s). Notice that the definition for addition adds one to the MAXINDEX of its operands since we require an extra bit for a final carry to fully represent the result.

We use 0EXTEND or CONTEXT to construct functions with a single argument (*p*) remaining unbound, which is the requested most significant index. In several places, TRUTH is used to obtain expressions for operands which represent truth values.

The `shiftop` and `compop` definitions make use of the functions from tables 3 and 4 respectively. LeftShift simply adds extra zeros to the right of a list of scalar expressions, removing an expression from the left every time it does, yielding a list of the same length as its argument. If any of these "lost" most significant bits are required in the result, the subexpression being shifted will have been evaluated to the correct length for the final result. A right shift is implemented as a reversed left shift.

The comparison operators use BusGt, CaseEqReduc and NoXsPresent to implement comparisons in terms of Verilog's ? :, &, $\wedge$, $===$ and $\sim$ operators. Case equality ("$===$") between vectors is a conjunction of case equivalences between the vectors' bits, achieved with CaseEqReduc. Logical equality ("$==$") is slightly more complicated, since it treats the result of comparisons which involve any unknown values as unknown. NoXsPresent is used to detect the presence of any unknowns in the bitwise comparison of two vectors. Its result is $1'bx$ if there is at least one and $1'b1$ if there are none. Since an exclusive-or

$$\begin{array}{llll}
\text{fun} & \text{LeftShift}\ es\ 0 & = & es \\
| & \text{LeftShift}\ e :: es\ k & = & \text{LeftShift}(es@(1'b0))(k-1);
\end{array}$$

$$\begin{array}{lllll}
[(& \text{“<<”}, & \text{fn}\ k \mapsto \text{fn}\ es \mapsto & \text{LeftShift}\ k\ es &), \\
(& \text{“>>”}, & \text{fn}\ k \mapsto \text{fn}\ es \mapsto & \text{rev}(\text{LeftShift}\ k\ (\text{rev}\ es)) &)]
\end{array}$$

Table 3: Definition of `shiftop` Functions.

$$\begin{array}{llll}
\text{fun} & \text{BusGt}\ []\ [] & = & 1'b0 \\
| & \text{BusGt}\ eg :: egs\ el :: els & = & \texttt{cond}(\texttt{bitxor}(eg, el), eg, \\
 & & & \qquad (\text{BusGt}\ egs\ els)); \\
 & & & \\
\text{fun} & \text{CaseEqReduc}\ [e_1]\ [e_2] & = & \texttt{caseeq}(e_1, e_2) \\
| & \text{CaseEqReduc}\ e_1 :: es_1\ e_2 :: es_2 & = & \texttt{bitand}(\texttt{caseeq}(e_1, e_2), \\
 & & & \qquad\qquad (\text{CaseEqReduc}\ es_1\ es_2)); \\
 & & & \\
\text{fun} & \text{NoXsPresent}\ [e_1]\ [e_2] & = & \texttt{bitand}(\texttt{bitxor}(e_1, \sim e_1), \\
 & & & \qquad\qquad \texttt{bitxor}(e_2, \sim e_2)) \\
| & \text{NoXsPresent}\ e_1 :: es_1\ e_2 :: es_2 & = & \texttt{bitand}(\texttt{bitand}(\texttt{bitxor}(e_1, \sim e_1), \\
 & & & \qquad\qquad\qquad \texttt{bitxor}(e_2, \sim e_2)), \\
 & & & \qquad\qquad (\text{NoXsPresent}\ es_1\ es_2));
\end{array}$$

$$\begin{array}{lrlll}
[(& \text{“>”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\text{BusGt}\ es_1\ es_2] &), \\
(& \text{“<”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\text{BusGt}\ es_2\ es_1] &), \\
(& \text{“}\geq\text{”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\texttt{bitnot}(\text{BusGt}\ es_2\ es_1)] &), \\
(& \text{“}\leq\text{”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\texttt{bitnot}(\text{BusGt}\ es_1\ es_2)] &), \\
(& \text{“===”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\text{CaseEqReduc}\ es_2\ es_1] &), \\
(& \text{“!==”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\texttt{bitnot}(\text{CaseEqReduc}\ es_2\ es_1)] &), \\
(& \text{“==”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\texttt{bitxor}(\texttt{bitnot}(\text{CaseEqReduc}\ es_2\ es_1), & \\
 & & & \qquad\qquad (\text{NoXsPresent}\ es_2\ es_1))] &), \\
(& \text{“!=”}, & \text{fn}\ es_1 \mapsto \text{fn}\ es_2 \mapsto & [\texttt{bitxor}((\text{CaseEqReduc}\ es_2\ es_1), & \\
 & & & \qquad\qquad (\text{NoXsPresent}\ es_2\ es_1))] &)]
\end{array}$$

Table 4: Definition of `compop` Functions.

involving an unknown is defined as unknown, the definitions for "==" and "!=" in the associativity list can produce the required $1'bx$.

We assume that the primary expressions involving identifiers and constant numbers generate empty lists of carries in the carry generating version.

The extra conditions on bit selects and part selects make use of the functions msb and lsb, which rely on the declarations of the identifier they are passed. For example, having declared `reg [7:0] r1`, msb `r1` is 7 and lsb `r1` is 0. The conditions ensure that the addresses selected are within the correct range for the identifier. Notice that dynamic bit selects, often used to implement multiplexers, are not allowed.

The definition of $[\![$ {elist} $]\!]$ is the only place where the denotation of a *list* of expressions is taken. This is defined recursively as:

$$\begin{aligned} [\![\, [] \,]\!] &= [] \\ [\![\, e :: es \,]\!] &= [\![\, e \,]\!] :: [\![\, es \,]\!] \end{aligned}$$

hence SELF is applied to every expression in the concatenation using the map function, with the results being CONCATenated together.

One final point is that the negated unary operators, $\sim \&(e)$, $\sim\!|\,(e)$ and $\sim \wedge(e)$ are treated as $\sim (\&(e))$, $\sim (|\,(e))$ and $\sim (\wedge(e))$ respectively.

fun	TwosComp_zero []	=	[]
|	TwosComp_zero $(1'b0) :: es$	=	$(1'b1) ::$ (TwosComp_zero es)
|	TwosComp_zero $(1'b1) :: es$	=	$(1'b0) ::$ (TwosComp_zero es);
fun	TwosComp_one []	=	[]
|	TwosComp_one $(1'b0) :: es$	=	$(1'b0) ::$ (TwosComp_one es)
|	TwosComp_one $(1'b1) :: es$	=	$(1'b1) ::$ (TwosComp_zero es);
fun	TWOSCOMP es	=	rev(TwosComp_one(rev es));

Table 5: Definition of TWOSCOMP Function.

$$
\begin{array}{rcl}
[\![\ \mathbf{unop}\ e \ ]\!] & = & (0, \\
 & & 0\mathrm{EXTEND}(\mathrm{UNOP}\ \mathbf{unop}\ [\![\ e \ ]\!])) \\[1em]
[\![\ !e \ ]\!] & = & (0, \\
 & & 0\mathrm{EXTEND}(\mathtt{bitnot}(\mathrm{TRUTH}[\![\ e \ ]\!]))) \\[1em]
[\![\ e_l\ \mathtt{logop}\ e_r \ ]\!] & = & (0, \\
 & & 0\mathrm{EXTEND}(\mathtt{logop}(\mathrm{TRUTH}[\![\ e_l \ ]\!], \mathrm{TRUTH}[\![\ e_r \ ]\!]))) \\[1em]
[\![\ e_l\ \mathtt{binop}\ e_r \ ]\!] & = & (\mathrm{MAXINDEX}([\![\ e_l \ ]\!], [\![\ e_r \ ]\!]), \\
 & & \mathrm{CONTEXTPAIR}([\![\ e_l \ ]\!], [\![\ e_r \ ]\!]) \\
 & & ((\mathrm{map}\ \mathtt{binop})\ o\ \mathrm{zip})) \\[1em]
[\![\ \sim e \ ]\!] & = & (\mathrm{INDEX}[\![\ e \ ]\!], \\
 & & \mathrm{CONTEXT}[\![\ e \ ]\!](\mathrm{map}\ \mathtt{bitnot})) \\[1em]
[\![\ e_c ? e_t : e_f \ ]\!] & = & (\mathrm{MAXINDEX}([\![\ e_t \ ]\!], [\![\ e_f \ ]\!]), \\
 & & \mathrm{CONTEXTPAIR}([\![\ e_t \ ]\!], [\![\ e_f \ ]\!]) \\
 & & ((\mathrm{map}(\mathrm{CONDRESOP}(\mathrm{TRUTH}[\![\ e_c \ ]\!])))\ o\ \mathrm{zip})) \\[1em]
[\![\ e\ \mathtt{shiftop}\ k \ ]\!] & = & (\mathrm{INDEX}[\![\ e \ ]\!], \\
 & & \mathrm{CONTEXT}[\![\ e \ ]\!](\mathtt{shiftop}'(\mathrm{CONSTANT}[\![\ k \ ]\!]))) \\[1em]
[\![\ e_l\ \mathtt{compop}\ e_r \ ]\!] & = & (0, \\
 & & 0\mathrm{EXTEND}(\mathrm{SELFPAIR}([\![\ e_l \ ]\!], [\![\ e_r \ ]\!])\mathtt{compop}')) \\[1em]
[\![\ +e \ ]\!] & = & [\![\ e \ ]\!] \\
[\![\ -e \ ]\!] & = & (\mathrm{INDEX}[\![\ e \ ]\!], \\
 & & \mathrm{CONTEXT}[\![\ e \ ]\!]\mathrm{TWOSCOMP}) \\
[\![\ e_l - e_r \ ]\!] & = & [\![\ e_l + (-e_r) \ ]\!] \\[1em]
[\![\ e_l + e_r \ ]\!] & = & ((\mathrm{MAXINDEX}([\![\ e_l \ ]\!], [\![\ e_r \ ]\!])), \\
 & & \mathrm{ARITHPAIR}([\![\ e_t \ ]\!], [\![\ e_f \ ]\!])) \\[1em]
[\![\ (e) \ ]\!] & = & [\![\ e \ ]\!]
\end{array}
$$

Table 6: Scalarising Verilog Expressions.

$$
\begin{array}{rcl}
\llbracket\, k \,\rrbracket & = & \llbracket\, 32'dk \,\rrbracket \\
\llbracket\, '\mathtt{B}k \,\rrbracket & = & \llbracket\, 32'\mathtt{B}k \,\rrbracket \\
\llbracket\, s'\mathtt{B}k \,\rrbracket & = & (s-1, \\
 & & \mathrm{0PAD}(\mathrm{bit\_list\_of\_based\_number}\ k)(s-1)) \\
 & & \\
\llbracket\, \mathtt{id}[e] \,\rrbracket & = & (0, \\
 & & \mathrm{0EXTEND}(\mathtt{id}[\mathrm{CONSTANT}\llbracket\, e \,\rrbracket])) \\
 & \mathrm{iff} & \quad (\mathrm{msb}\ \mathtt{id} \leq (\mathrm{CONSTANT}\llbracket\, e \,\rrbracket) \leq \mathrm{lsb}\ \mathtt{id}) \\
 & & \ \vee\ (\mathrm{msb}\ \mathtt{id} \geq (\mathrm{CONSTANT}\llbracket\, e \,\rrbracket) \geq \mathrm{lsb}\ \mathtt{id}) \\
 & = & \mathrm{COPYCAT}\ 1'bx \quad \text{otherwise} \\
 & & \\
\llbracket\, \mathtt{id}[e_1 : e_2] \,\rrbracket & = & \llbracket\, \{\mathtt{id}[e_1], \ldots, \mathtt{id}[e_2]\} \,\rrbracket \\
 & \mathrm{iff} & \quad ((\mathrm{CONSTANT}\llbracket\, e_1 \,\rrbracket < \mathrm{CONSTANT}\llbracket\, e_2 \,\rrbracket) \\
 & & \qquad \wedge\ (\mathrm{msb}\ \mathtt{id} < \mathrm{lsb}\ \mathtt{id})) \\
 & & \ \vee\ ((\mathrm{CONSTANT}\llbracket\, e_1 \,\rrbracket > \mathrm{CONSTANT}\llbracket\, e_2 \,\rrbracket) \\
 & & \qquad \wedge\ (\mathrm{msb}\ \mathtt{id} > \mathrm{lsb}\ \mathtt{id})) \\
 & & \ \vee\ \ (\mathrm{CONSTANT}\llbracket\, e_1 \,\rrbracket = \mathrm{CONSTANT}\llbracket\, e_2 \,\rrbracket) \\
 & & \\
\llbracket\, \mathtt{id} \,\rrbracket & = & \llbracket\, \mathtt{id}[(\mathrm{msb}\ \mathtt{id}) : (\mathrm{lsb}\ \mathtt{id})] \,\rrbracket \\
 & & \\
\llbracket\, \{\mathrm{elist}\} \,\rrbracket & = & (\mathrm{CATINDEX}\llbracket\, \mathrm{elist} \,\rrbracket, \\
 & & \mathrm{0PAD}(\mathrm{CONCAT}(\mathrm{map}\ \mathrm{SELF}(\llbracket\, \mathrm{elist} \,\rrbracket)))(\mathrm{CATINDEX}\llbracket\, \mathrm{elist} \,\rrbracket)) \\
 & & \\
\llbracket\, \{k, \{\mathrm{elist}\}\} \,\rrbracket & = & ((\mathrm{CONSTANT}\llbracket\, k \,\rrbracket * (\mathrm{CATSIZE}\llbracket\, \{\mathrm{elist}\} \,\rrbracket)) - 1, \\
 & & \mathrm{0PAD}(\mathrm{MULTICAT}(\mathrm{SELF}(\llbracket\, \{\mathrm{elist}\} \,\rrbracket))\mathrm{CONSTANT}\llbracket\, k \,\rrbracket)) \\
 & & (((\mathrm{CONSTANT}\llbracket\, k \,\rrbracket) * (\mathrm{CATSIZE}\llbracket\, \{\mathrm{elist}\} \,\rrbracket)) - 1))
\end{array}
$$

Table 7: Scalarising Verilog Primary Expressions.

5 Scalarising Verilog code

We now turn our attention to the process of incorporating the scalarised expressions into the Verilog syntactic categories which include them. Although both carry generating and non-carry generating forms of the algorithm have so far been shown, only the former will be described here, since without carries, scalar expression lists enclosed in Verilog's concatenations can be directly substituted. The issue of interest from now on is what to do with the carries we have generated.

Table 8 shows mostly syntactic components concerned with flow of control. In these, an expression is usually used as a condition to determine the next statement to execute, so the vector expression e_c is replaced with the scalar version obtained from ANS(TRUTH$[\![e_c]\!]$). The carries from CARRIES(TRUTH$[\![e_c]\!]$) are turned into immediate assignments with the constructor `immediate_assignment`. Each seperate carry wire must be assigned seperately, so that less significant bits' carries are set before the more significant bits' carries which depend on them. In the translation of `while` we must set the carries once before the first test, and again after executing the body of the `while`.

The rule for `repeat` performs unwinding, and the `for` and `wait` rules are recursively defined by evaluating an equivalent piece of Verilog code. The definiton of the function SUPPORT, which returns a list of the identifiers present in its argument, is not shown, as it is defined recursively on the whole structure of expressions.

Notice in the rule for `wait`, the use of a *null statement* ';' after the timing control, making the body of the `while` loop do nothing once the guard is triggered. Also, with this equivalent code, it is possible that in-between the triggering of the guard, and the evaluation of the original expression, the support is changed by a non-deterministic interleaving with another process. In such cases, a transient true value which could release the guard could be missed. Such behaviour has been noticed in the Verilog simulator.

In table 9, the context for evaluating the rvalue is taken from the INDEX of the lvalue.

5.1 Event guards

Expressions are permitted in event guards and this adds a complication because our generated carries can no longer accompany the scalarised expression. A solution is to *continually assign* the carries.

A further complication arises if we consider the final form of our language V, where only identifiers may occur in the event guards. Again, the solution is to continually assign the original scalar expressions.

Finally, the `posedge` and `negedge` guards are only sensitive to changes in their least significant bit.

```
$\llbracket$ if $e_c$ $s_t$ else $s_f$ $\rrbracket$          =  begin
                                     (map immediate_assignment)
                                         (CARRIES(TRUTH$\llbracket$ $e_c$ $\rrbracket$))
                                   end
                                   if ANS(TRUTH$\llbracket$ $e_c$ $\rrbracket$) $\llbracket$ $s_t$ $\rrbracket$ else $\llbracket$ $s_f$ $\rrbracket$

$\llbracket$ forever $s_f$ $\rrbracket$              =  forever$\llbracket$ $s_f$ $\rrbracket$

$\llbracket$ repeat $e$ $s_r$ $\rrbracket$             =  begin
                                     COPYCAT$\llbracket$ $s_r$ $\rrbracket$(CONSTANT$\llbracket$ $e$ $\rrbracket$)
                                   end

$\llbracket$ while $e_c$ $s_l$ $\rrbracket$            =  begin
                                     (map immediate_assignment)
                                         (CARRIES(TRUTH$\llbracket$ $e_c$ $\rrbracket$))
                                   end
                                   while ANS(TRUTH$\llbracket$ $e_c$ $\rrbracket$)
                                     begin
                                       $\llbracket$ $s_l$ $\rrbracket$
                                       (map immediate_assignment)
                                         CARRIES(TRUTH$\llbracket$ $e_c$ $\rrbracket$)
                                     end

$\llbracket$ for ($r_1 = e_1$; $e_c$; $r_2 = e_2$) $s$ $\rrbracket$  =  begin
                                     $\llbracket$ $r_1 = e_1$; $\rrbracket$
                                     $\llbracket$ while $e_c$ begin $s$; $r_2 = e_2$; end $\rrbracket$
                                   end

$\llbracket$ wait ($e$) $s$ $\rrbracket$              =  $\llbracket$ while ($\sim e$) @(SUPPORT $e$) ; $s$ $\rrbracket$

$\llbracket$ begin end $\rrbracket$               =  (*nothing*)
$\llbracket$ begin $ss$ end $\rrbracket$            =  begin $\llbracket$ $ss$ $\rrbracket$ end
```

Table 8: Scalarising Flow Control Statements.

```
fun  NHEADS e :: es 0   =  [e]
 |   NHEADS e :: es n   =  e :: (NHEADS es (n − 1))
fun  SNIP es p          =  rev(NHEADS(rev es) p)
```

```
⟦ l = g e; ⟧    =  begin
                     (map immediate_assignment)(CARRIES
                     (CONTEXT⟦ e ⟧(MAXINDEX(⟦ l ⟧,⟦ e ⟧))))
                   end
                   {ANS(SELF⟦ l ⟧)} = ⟦ g ⟧ {ANS
                     (CONTEXT⟦ e ⟧(MAXINDEX(⟦ l ⟧,⟦ e ⟧)))}

⟦ l <= g e; ⟧   =  begin
                     (map immediate_assignment)(CARRIES(SNIP
                       (CONTEXT⟦ e ⟧(MAXINDEX(⟦ l ⟧,⟦ e ⟧)))
                       (INDEX⟦ l ⟧)))
                   end
                   {ANS(SELF⟦ l ⟧)} <= ⟦ g ⟧ {ANS(SNIP
                     (CONTEXT⟦ e ⟧(MAXINDEX(⟦ l ⟧,⟦ e ⟧)))
                     (INDEX⟦ l ⟧))}
```

Table 9: Scalarising Assignments.

```
⟦ posedge(e) ⟧  =  (map posedge(ANS(SNIP(SELF⟦ e ⟧)0)))
                     Continuously assign(CARRIES(SNIP(SELF⟦ e ⟧)0))
⟦ negedge(e) ⟧  =  (map negedge(ANS(SNIP(SELF⟦ e ⟧)0)))
                     Continuously assign(CARRIES(SNIP(SELF⟦ e ⟧)0))
⟦ anyedge(e) ⟧  =  (map anyedge(ANS(SELF⟦ e ⟧)))
                     Continuously assign(CARRIES(SELF⟦ e ⟧))

 ⟦ @(edges) ⟧   =  @(flat(map ANS⟦ edges ⟧))
                     Continuously assign
                       (CARRIES(flat(map CARRIES⟦ edges ⟧)))
    ⟦ #e ⟧      =  #CONSTANT(⟦ e ⟧)
```

Table 10: Scalarising Event Guards.

Part II
Verilog Expressions

by Myra VanInwegen

Our work with Verilog expressions proceeded in two stages. The first stage was to work out exactly what the meaning of Verilog expressions should be. In doing this we wrote a program that evaluated expressions. Part of the problem here is to figure out width and signedness (signed vs. unsigned) of intermediate results. The second stage was to normalize Verilog expressions, transforming them into a form such that they can be evaluated exactly as they are, without doing any width calculations.

6 Verilog Expression Evaluation

How to evaluate Verilog expressions ought to be clearly specified by the Proposed IEEE Standard 1364 describing Verilog HDL [2]. However, the Standard only describes the results in the straightforward cases, not what should happen if the programmer does something unexpected. To see what to do in these cases, we referred to three simulators: Cadence Design Systems' Verilog-XL, Wellspring Solutions' VeriWell, and interHDL's ViperFree. We also used a Verilog textbook by Thomas and Moorby [6].

The semantics we chose usually follows the Standard, but if all three simulators took a different route, and they all agreed with each other, then we usually took the solution used by the simulators. If the Standard wasn't specific about some aspect, we would pick our favorite of the solutions proposed by the simulators.

First we explain the evaluation strategy (in general terms, how the program evaluates expressions), then we explain in detail the choices we made in determining exactly how expressions should be evaluated.

6.1 Evaluation Strategy

Probably the most complicated aspect of expression evaluation is determining the widths of intermediate results. If you add two vectors together, the width of the result is the maximum of the widths of the two vectors. Using computations such as this, we determine the minimum width needed to hold the final result, based on the widths of the components. Here, the components in the computation of the width are not necessarily the leaves in the syntax tree of the expression, but are just those subexpressions that have a fixed width, including identifiers, bit and part selects, expressions with unary or binary operators that have a single-bit result, and concatenations.

The width used for computing intermediate results in operations such as addition is the maximum of the minimum width needed to hold the final result and the width of the lvalue to which it is assigned. Each of the components in the expression must be expanded to this size (by adding zeros to the left of the most significant bit) before the computation is started. We must do this in order to allow operators that invert the most significant bits to operate in a sensible manner. For example, in the program fragment `-3'b010 + 5'b00101`, we are adding -2 to 5, and the result should be 3. This is in fact what we get if we expand the first component to a 5 bit width before we negate it. However, if we negate the first component while it is still 3 bits wide, then pad with zeros, then add, we get the result 11 (in binary, 01011).

Thus we must explore a good portion of the syntax of the expression in order to find the width before we can begin evaluating the expression. However, simply finding the widths of the components requires evaluating expressions: the repetitions argument in the multiple concatenation and the indices of a part select may themselves be expressions.

To break this seeming circularity we adopted the approach used by Daryl Stewart, in which expression evaluation is done in one recursive examination of the expression. The result of an expression evaluation is a pair of an SML integer and an SML function. The integer is the minimum width needed to hold the expression. The function can be considered a partial evaluation. It takes a integer, which is expected to be greater than or equal to the minimum width returned, and completes the evaluation at that width. We get the final result by applying the function to the maximum of the returned width and the width of the lvalue to which it is being assigned.

6.2 Signed vs. Unsigned Computation

Since a two's complement representation is used for negative numbers, most operations (including multiplication and addition) are exactly the same for signed as for unsigned numbers. However, for some operations it makes a difference. For example, `-8 / 4` could either end up a large positive number or -2, depending on whether -8 is considered a large positive number (unsigned) or a small negative number (signed). Similarly, whether or not `-4 < 5` depends on whether we're comparing the arithmetic values, or just the bit patterns. That is, it depends on whether -4 is treated as signed or unsigned.[1]

According to the Standard, arithmetic operations on integers are to be treated as signed, whereas operations on registers or nets are treated as unsigned. With integer constants, if the constant is written with a size or base, then it is treated as unsigned, but plain numbers are treated as integers.

[1] Actually, the Standard never states explicitly that a relational operation on integers should be treated as a signed comparison, but at least one textbook (Thomas and Moorby) does, and all the simulators do, so we do as well.

What exactly is an integer anyway? The Standard says that integers are general purpose registers used for simulation, but are not meant to model hardware registers. The Standard does not say how many bits an integer register should have. In fact it suggests that an integer register may not have a fixed width (there is a note stating that implementations may limit the maximum size of an integer). However, all the simulations (and the Thomas and Moorby textbook) treat an integer as a special register with a fixed range (the most significant bit has index 31, the least 0). We adopt this latter approach, except that we can easily change the width of the integer registers by changing a constant in the implementation.

What if one signed and one unsigned argument are used in arithmetic operators? The Standard suggests that the signed vs. unsigned aspect of each argument should be considered separately. For example, `-12 / 3` results in -4, and `-10'd12 / 3` results in a large positive number, but probably `-12 / 5'd3` should result in -4. However, two of the simulators we use (namely, Verilog-XL and ViperFree) treat the computation as signed only if both operands are signed. Using this method, `-12 / 5'd3` would result in a large positive number: because second argument is unsigned, the -12 is treated as unsigned as well. This method makes for fewer different cases in implementations of operators. It also makes it easier to decide when the result is signed: the result of a binary operation is signed if both arguments are signed (with one exception, see below). Thus we have adopted this method.

What if we do something like `(a op1 b) op2 c`, where `op1` and `op2` are binary arithmetic operations, `a` and `b` are signed values, and `c` is an unsigned value? According to our approach, `op2` is evaluated using unsigned arithmetic, since `c` is unsigned, but is the evaluation of `op1` done using signed or unsigned arithmetic? Verilog-XL uses unsigned for `op1`; the other two use signed. We prefer the idea of using signed if both operands are signed, so in this situation we use signed for `op1`.

It is even less clear whether the result of non-arithmetic operations should be signed. For example, is `-6 | -8` signed? There is no suggestion one way or another in the Standard. However, all the simulators say that the result will be signed, so we follow their lead. What about `-12 << 1'b1`? This is trickier. The Standard says that the width of the result is the width of the left argument, and that the right argument is self-determined and also should be considered unsigned (thus we never end up with a negative number to shift by). Our opinion is that since the right argument is always considered unsigned, then its actual sign should not affect the sign of the result. Thus if the left argument is signed, then the result is as well. Two of the simulators (ViperFree and VeriWell) agree with me; Verilog-XL insists that both the left and right arguments be signed before the result will be.

In our semantics, the results of a part select and concatenation are always unsigned, even if the part select selects all the bits of an integer or an integer is the only item in the concatenation (in these cases the bits we get are just the original bits in the integer). Verilog-XL does things our way. With VeriWell, the result of the concatenation example is signed while the part select is unsigned. ViperFree returns a signed value in both of the special cases above.

We treat the repetitions argument in multiple concatenations as unsigned, in analogy with treating the right argument of the shift operator as unsigned.

Integers are special registers of a fixed width (see above). What if we add a negative integer and a register that has a width larger than the width of an integer? Do we extend the most significant bit of the integer with zeros (as we normally do), or do we extend with ones, thus preserving the sign of the number? The Standard suggests that whenever values are extended, they are extended with zeros. Yet extending negative signed values with ones makes logical sense. Two of the simulators (Verilog-XL and VeriWell) extend with zeros, while the third (ViperFree) extends with ones. We will go with the majority opinion (and the Standard) on this and extend with zeros.

6.3 Differences from the Standard—Widths

There are several places where our semantics differs from that in the Standard in handling the widths of expressions. In each case, all three simulators did the same thing, which was different from what the Standard says. These differences are summarized by Table 11, which is similar to Table 4-21 in the Standard. Table 11 also includes comments on signedness of operations, as discussed in the Section 6.2.

In our semantics, the length of a multiplication is the maximum length of the two operands, while in the Standard it is the sum of the two lengths. We chose to use the maximum for several reasons. First, all the simulators do it this way. Second, it uses the same philosophy as addition, in which the carry bit can be lost unless you make sure that you've got the bits available to hold it. Third, we allow signed operations only on integer-width values. If we follow the Standard's suggestion and then multiply two integers, we end up with a value that is twice the width of an integer, which we would then treat as unsigned. This does not seem sensible.

In our semantics, the operands of the relational and equality operators are context-determined. To implement this, we figure out natural widths of each of the operands, evaluate the operands to the maximum of those widths, then compare. The Standard says that the operands of all the relational and equality operators are self-determined, and then they are extended with zeros to make their lengths equal before the comparison. We chose to do it our way because all the simulators do it that way, and because it seems more logical: with the Standard's semantics, the comparison `-3'd5 == -5'd5` would result in false (a one-bit binary 0).

The Standard specifies that the width of the return value of a conditional operator will be the maximum of the widths of the second and third operand. As Table 4-21 doesn't mention that j and k are self-determined, then one can

Expression	**Bit length**	**Comments**
unsized constant number	same as integer	Result is signed
sized constant number	as given	Result is unsigned
i op j, where op is: + − ∗ % & | ^ ^~ ~^	max(L(i), L(j))	Result is signed if i and j are
op i, where op is: + − ~	L(i)	Result is signed if i is
i op j, where op is: === !== == != > >= < <=	1 bit	Operands are context-determined, length used is max(L(i), L(j))
i op j, where op is: && ||	1 bit	Operands are self-determined
op i, where op is: & ~& | ~| ^~ ~^	1 bit	Operand is self-determined
i op j where op is: >> <<	L(i)	j is self-determined and treated as unsigned Result is signed if i is
i ? j : k	max(L(j), L(k))	i is self-determined Result is signed if j and k are
{i, . . . , j}	L(i)+. . . +L(j)	Operands are self-determined Result is unsigned
{i{j, . . . , k}}	i*(L(j)+. . . +L(k))	Operands are self-determined i is treated as unsigned Result is unsigned

Table 11: Bit lengths and signs resulting from expressions

suppose that they are context-determined. However, in the section on conditionals, it is stated that if one of the operands is shorter than the other, it will be extended to match the length of the larger operand. This suggests that the second and third arguments are self-determined. We have chosen to make the second and third arguments context-determined, both because all the simulators do this and because it results in more intuitive behavior (for example, `a ? -12 : -5'd12` evaluates to an integer-width unsigned value with the twos complement representation of −12 no matter what `a` is).

6.4 The Logical Operators

The Standard's description of the meaning of the logical operators && (and) and || (or) in Section 4.1.9 leaves much to be desired. Presumably, `a && b` will be evaluated by determining the truth value of `a` and `b`, then performing the && according to some truth table. However, it is not stated exactly how one turns a vector value into a single bit logical value. Nor is it stated what truth tables to use for && and ||. What to do with true and false is obvious, but what do we do with ambiguous truth values?

In Section 4.1.13 on the conditional operator, the Standard says that if the vector has the value zero it is false, if it has a known value other than zero it is true, and if the value of the vector is x or z the truth value is ambiguous. Piecing this together with information from other parts of the Standard, we can figure out that if any bit of the vector is x or z, then the truth value is x. If the value is all zeros, then the result is 0 (false). Otherwise, it is either all ones or is a mix of zeros and ones, and the truth value is 1 (true). This is the semantics we use. Incidentally, two of the three simulators we use have a different method to compute truth values: Verilog-XL and ViperFree say that a value is true if there are any 1 bits in it, is ambiguous if there is a mix of zeros and xs, and is false if it is all zeros; only VeriWell agree with the Standard.

We still need to know the truth tables for && and ||. After factoring out their different truth valuations, all of the simulators use the same truth tables for the logical operators as for the bitwise binary operators, and so we do this as well.

6.5 Short-Circuit Expression Evaluation

Section 4.1.4 of the Standard states that short-circuit evaluation is allowed. This form of evaluation allows one to leave part of an expression unevaluated if the result of the expression can be determined early. For example, if E_1 has the truth value 0 (false), then E_1&&E_2 will be 0, no matter what E_2 is, so there is no need to evaluate E_2.

We do not implement this in the program. The main reason for this is that leaving it out makes the program code clearer. When we include function evaluation, there will be another reason not to do it. The problem is that function calls may have side effects (changes to non-local variables). Recall that one must partly evaluate an expression in order to find its width. Thus, using the example above, any function calls occurring in say, part select indices in E_1 and E_2 would be evaluated. If E_1 has the truth value 0 and we use short-circuit evaluation, then the rest of E_2 will not be evaluated. Thus some function calls in E_2 would be evaluated, while others are not. If the functions have side effects, this could be very confusing to the user, as it would be hard to predict which function calls would be evaluated and which would not.

6.6 Ranges and Bit and Part Selects

The range gives the most and least significant bits in reg or net declarations. These are given by constant expressions. The Standard does not specify what to do if there are x or z bits in the resulting values. Different simulators take different approaches. Verilog-XL seems to use x or z bits as if they were 0. ViperFree treats x bits as 0 and z bits as 1. VeriWell prints prints out an error message. Since range specifications much be constants, we feel that it is a bad idea to try to make sense of ranges with x or z bits in them, so we issue an error message if this happens.

The Standard and the Thomas and Moorby book state that the keywords **vectored** or **scalared** may be used to modify the range to specify that a vector net or trireg must be treated as a single entity, that is, bit or part selects cannot be done on it. The simulators take a strange approach to these keywords. Verilog-XL and ViperFree allow one to declare a net as vectored, but they ignore it without giving any error messages. VeriWell prints out "sorry: VECTORED keyword is not supported; all nets treated as vectored". This would leave one to think that one cannot do bit or part selects on any net, but this is not the case. It seems that the implementors of VeriWell have gotten a bit confused about what the **vectored** keyword means.

We implement the **vectored** and **scalared** keywords: if a net is declared as vectored and you try to do a bit or part select on it, an error message is printed out and the execution stops.

The Standard specifies that if a bit or part select is out of range, or if the indices are x or z, then the value returned will be x. It also says that part-select indices that evaluate to x or z may be flagged as compile-time errors, and that bit or part-select indices that are outside the declared range may be flagged as compile-time errors.

The simulators all return an x if a bit-select is out of range. They also all return x if the bit-select contains an x, and two of the three return x if the bit-select contains a z. However, ViperFree sometimes gives an "Index out of bound" error if the bit-select contains a z, and it sometimes returns an x.

For part selects, if the part-select is out of range, Verilog-XL and VeriWell seem to return 0s for the part that is out of bounds on the side of the most significant bit, but they returns a result that is *all* 0s if the part is out of bounds on the least significant bit side. For example, in the code below

```
reg [3:1] r2, r3, r4;
...
r2 = 3'b10z;
r3 = r2[4:2];
r4 = r2[2:0];
```

r3 ends up as 010, while r4 is 000. ViperFree gives a "Range index out of bound" in both cases.

If a part select contains an x or a z, Verilog-XL and VeriWell give error messages. ViperFree seems to interpret an x in a part select index as a 0 bit, and a z as a 1 bit.

This is very bizarre. We do the following. For bit selects, if there is an x or z in the index, or if the index is out of the declared range, we return x. For part selects, if there is an x or z in one of the indices, or if the part select is out of range, we give an error message. We chose different behaviors for bit and part selects because bit selects can be dynamic, while part-selects much be constant. In a dynamic value, there may be temporary situations where there are unknown or high-impedance bits in the index, or the index is out of range, so we return what we feel is a sensible value (x). Since part selects are static (constant), if the desired part has xs or zs in the indices, or if the part is out of range, then it is a mistake and the user should be notified.

6.7 Parameters

Parameters are constants. When you declare them, the form of the value assigned to them determines the type of the parameter. The following examples are from the Standard.

```
parameter msb = 7;
parameter r = 5.7;
```

The explanation given is that `msb` is a constant value 7, and `r` is a real value.[2] Thus a parameter's form determines the way it is used. Since unsized, unbased constants are treated as integers, one can suppose that when these parameters are used in expressions, they are treated as signed values with integer width. Further extending this "value determines type" notion, we can guess that based numbers will result in registers with a width appropriate for the number (integer width for unsized numbers, the given size for sized numbers). This is what all the simulators do, and thus our implementation does as well.

The Standard and the Thomas and Moorby book do not allow range specifications for parameters. All the simulators allow it, but the way they handle

[2]Recall that we don't implement real numbers; this is just an illustration.

the ranges is very unintuitive: using a range in a parameter declaration does not change the size of the resulting parameter. For example, in the following declaration

```
parameter [5:0] pa = 6'b110101, pa2 = 5, pa3 = 3'b101;
```

pa is a 6-bit value, pa2 is a 32-bit integer, and pa3 is a 3-bit value. The only thing that using a range in a parameter declaration does is change the indexes of the bits. Consider the following declaration:

```
parameter [0:1] pb = 4'b01xz;
```

The result is that pb is the 4-bit value 01xz. In all the simulators, the last two bits (x and z) have indices 0 and 1: if we select bit 1 (using `pb[1]`), the result is a 1-bit z. However, if we select an index that is in the value of the parameter but not in the declared range, each simulator returns something different. If we ask for `pb[-1]` Verilog-XL returns 1, the bit in that position. VeriWell returns an x, since -1 is out of the declared range. ViperFree considers it an error, saying "Index out of bound". Because of the strange way it is handled, we will go with the Standard and not allow ranges in parameter declarations.

The Standard is ambiguous about whether bit or part selects on parameters should be allowed. Among the listing of possible operands (just before Section 4.1 on operators), bit or part selects on parameters are not mentioned. However, just a few of paragraphs earlier, bit and part selects on parameters are listed as possible operands for constant expressions. Certainly, constant expressions should be a subset of expressions, so we will allow bit and part selects on parameters in all situations. Two of the simulators (Verilog-XL and VeriWell) do as we do; ViperFree does not allow bit or part selects on parameters.

7 Verilog Expression Normalization

The normalization of the expressions results in an expression that has been altered in three major ways. First, all the subexpressions that are required to be constant have been evaluated. Second, all the width manipulations have been made explicit. Components are padded with the use of concatenations, and if the result of an expression is wider than will fit into the lavalue to which it's assigned, a part select retrieves the least significant bits. Finally, declarations are normalized so that ranges of busses are $[width - 1 : 0]$, where *width* is the number of bits in the bus. Bit and part selects in expressions and lvalues are changed accordingly.

In order to simplify the semantics, we do not distinguish between signed and unsigned values when doing constant expression evaluation. The reasons for this are explained below in Section 7.1.

The input is the abstract syntax (Daryl Stewart's *P1364.meta* syntax) for a Verilog source file, and the output is the modified source. We can deal with concrete syntax with the use of the CLaReT tools [1]. In addition to modifying the expressions, all widths of arguments to module instantiations, task enables, and function calls are checked. The types of identifiers is checked (for example, registers cannot be used in continuous assignments), and exceptions are raised (with arguments describing roughly what went wrong) when a problem is encountered.

Here are some examples that show the results of expression normalization. The following declarations are used for all the examples.

```
reg [3:0] reg4, reg4b;
reg [5:0] reg6;
reg [11:0] reg12;
reg reg1;
parameter p = 5, p2 = 16;
```

The examples will be shown as follows:

```
before: statement1
after: statement2
```

This shows a statement (`statement1`) and what the program transforms it to (`statement2`). We use a full statement rather than just an expression because the width of the lvalue to which an expression is assigned affects the width of intermediate expressions. Occasionally the before and after lines will contain declarations in addition to statements.

The operands of a sum are "context-determined". This means that the operands both contribute to the final width of the expression and are affected by the width required by the context. For example:

```
before: reg12 = reg4 + reg6;
after: reg12 = {8'd0, reg4} + {6'd0, reg6};
```

The result is determined this way: the maximum width needed to hold the sum is 6 bits. Since the lvalue to which it's assigned has 12 bits, the expression must be evaluated to 12 bits. In order to evaluate a sum to 12 bits, we expand each operand to 12 bits, then do the sum.

The arguments of minus (−) are also context-determined, as well as for bitwise AND (&), OR (|), XOR (^), and XNOR (^~ or ~^).

If the width needed to hold an expression is larger than the width of the lvalue to which it's assigned, a part select retrieves the least significant bits of the result of the expression.

```
before: reg6 = (reg4 + reg6) - reg12;
after: (({8'd0, reg4} + {6'd0, reg6}) - reg12)[5:0];
```

Since the maximum width of the components is 12 bits, the components with width less than 12 are expanded via concatenating them with bunches of 0s.

Since the lvalue to which the expression is assigned has a width of 6 bits, the least significant bits of the result are extracted with a part select. Note that the use of a part select with a general expression rather than an identifier is non-standard Verilog. It is an extension we have made to the syntax.

The operand of unary operators for two's complement negation (−) and bitwise negation (~) are context-determined.

```
before: reg12 = -reg6 + reg4;
after: reg12 = -{6'd0, reg6} + {8'd0, reg4};
```

Note that the expansion of `reg6` via the concatenation happens before the negation—that way we will have a proper 12-bit two's complement representation of the intermediate values, and hence the result.

If we need to expand a concatenation, the extra 0 bits are included in the concatenation itself.

```
before: reg12 = {reg4, reg6};
after: reg12 = {2'd0, reg4, reg6};
before: reg12 = {reg4, reg4b} + reg6;
after: reg12 = {4'd0, reg4, reg4b} + {6'd0, reg6};
```

With shift expressions, the left operand (the item to be shifted) is context-determined, while the right operand is evaluated at its natural width (this is called "self-determined").

```
before: reg6 = reg4 << 1;
after: reg6 = {2'd0, reg4} << 1;
```

With conditional expressions, the second and third operands are context-determined, while the first operand is self-determined.

```
before: reg12 = reg1 ? reg4 : reg6;
after: reg12 = reg1 ? {8'd0, reg4} : {6'd0, reg6};
before: reg4 = reg1 ? reg6 : reg12;
after: reg4 = (reg1 ? {6'd0, reg6} : reg12)[3:0];
```

For binary operators that require a bit-by-bit comparison, but have a one-bit result, the operands are expanded to the maximum of the sizes of the two operands. The operators where this is the case are logical equality (==), logical inequality (! =), case equality (===), case inequality (! ==), less than (<), greater than (>), less than or equal (<=), and greater than or equal (>=).

```
before: reg4 = (reg4 > reg6) ? reg4 : reg4b;
after: reg4 = ({2'd0, reg4} > reg6) ? reg4 : reg4b;
```

Unary operators with one-bit results (like the reduction operators |, ~|, ^, ~^, &, ~&) have self-determined arguments.

```
before: reg6 = reg6 + |reg4;
after: reg6 = reg6 + {5'd0, |reg4};
```

The meaning of the reduction operators involving negation is that the operation is performed, then the result is negated. For example, `~|reg6` is equivalent to `~(|reg6)`.

One thing that the program does not make explicit is the truth reduction required to get a truth value out of a vector. For example, the statement

```
reg4 = (reg6) ?  reg4 :  reg4b;
```

is left exactly as it is by the program, although a one-bit truth value is needed as the first operand of the conditional. A vector is converted to the truth value in the following way: if any bit of the vector is x or z, then the truth value is x (uncertain). If the value is all zeros, then the result is 0 (false). Otherwise, it is either all ones or is a mix of zeros and ones, and the truth value is 1 (true). Since there is no clear way to represent this in Verilog expression syntax, we do not make it explicit, but leave it to whatever takes the output of this program as input to deal with this.

For logical operators (&& and ||), where the operands are reduced via truth reduction to one bit values that are then compared, both args are self-determined. Thus, for example, the statement `reg4 = (reg6 && reg12) ? reg4 :  reg4b;` is unchanged by the program.

In Verilog some expressions (for example, in ranges of declarations and in the repetitions operand of a multiple concatenation) are required to be constant expressions (that is, the only identifiers are parameters). These expressions are fully evaluated by the program.

```
before: reg6 = reg12[p + 3:p-2];
after: reg6 = reg12[8:3];
before: reg12 = {p - 3{reg4}};
after: reg12 = {4'd0, {2{reg4}}};
```

As noted below, in Section 7.1, we do not allow function calls in constant expressions. Also, we do not do signed computation in evaluating constant expressions.

We normalize declarations so that the ranges become $[width - 1 : 0]$, where *width* is the number of bits in the vector. Bit and part selects in expressions and lvalues are changed accordingly.

```
before: reg [2:9] reg8;  ...  reg4 = reg8[4:7];
after: reg [7:0] reg8; ...  reg4 = reg8[5:2];
before: reg [1:p2] short; reg [1:p2+p2] long;
after: reg [15:0] short; reg [31:0] long;
before: reg [2:9] reg8;  ...  reg8[3] = 0;
after: reg [7:0] reg8; ...  reg8 [6] = 1'd0;
```

Because we do not deal with signed numbers during evaluation of constant expressions, negative indices are not allowed. We do not allow concatenations in lvalues. We do not allow block declarations in modules. The output of the program has all the declarations placed at the top of the module, no matter where you put them in the code. All of the above are changes from standard Verilog.

7.1 Limitations

All width conversions needed to evaluate expressions are made explicit in the result produced by the program, except one: when a bus is used as a truth value, a reduction must be made to it to determine its truth value.

The rest of this section summarizes the differences between standard Verilog and the input accepted and produced by the program.

The program doesn't understand basic modules (gates such as AND, NOT, and pmos). Module, function, and task declarations must appear before their use (in function calls, task enables, or module instantiations) in the text. We don't allow arrays of module instantiations.

We do not allow concatenations as lvalues. Since connections to output and inout ports in module instantiations and arguments of task enables declared as output or inout act as lvalues, concatenations are not allowed here either. Connections to output/inout ports in module instantiations and output/inout arguments to task enables must be the same width as that used in the declaration within the module or task. Concatenations and bit and part selects are not allowed in port lists in module declarations.

The program does not handle declarations local to functions and tasks (other than its input/output parameters), or block declarations in modules. It is intended that the user be allowed to include these in his/her Verilog code, but the code should be processed first by Daryl Stewart's scope-checking code which deals with this.

All register, wire, and input/output declarations in modules are moved to the top of the module (before task or function declarations, assignments, and statements) in the output of the program. Comments are eliminated.

We do not handle **defparam** statements or parameter overrides in module instantiations. We do not handle charge or drive strength, **time**, **real** or **realtime** variables, or simulator directives such as monitor and display stamements.

The method used to evaluate constant expressions is limited in two ways. First, it cannot handle function calls. Second, it does not do signed arithmetic. In Verilog, integer variables and unsized integer constants should be treated as signed. The program does use proper two's complement representations of numbers, so it will get most operations right despite not explicitly treating integers as signed. There are a few cases where special consideration is needed to get the "right" answer. These are in comparison operators over vectors.[3] In a comparison, a small (in absolute value) negative number will be represented with the same bit pattern as a large positive number, so it will be larger than most positive numbers. Thus, one may get unexpected behavior, as in

```
before: reg [((-5 > 5) ? 10 : 20):0] regc;
after: reg [10:0] regc;
```

[3]There is a problem with the division operation as well, but since the syntax we are using doesn't have division, it doesn't show up here.

Part III
Flattening Verilog Module Hierarchies

by Daryl Stewart

Here we describe a method for producing a single module version of a hierarchical piece of verilog code. After explaining some terminology, the procedures used for this are described and the method for applying them is explained. An example of a hierarchy is given with a step by step explanation of its flattening.

8 Terminology for Instantiations

A Verilog *module-hierarchy* consists of one or more *top-level* modules containing tasks and instantiations of submodules, which may themselves contain tasks and instantiations of further modules. In order to *flatten* a module we need to substitute suitable statements for task invocations and flattened versions of submodules for module instantiations.

I use the term *module type name* to refer to the name given to a module by a module declaration. When modules are instantiated, they are given *module instance names*. A top-level module is one which is never instantiated, therefore it has no module instance name.

The syntax of an instantiation [2, section 12.1.2 "Module instantiation"] using Backus-Naur form is shown in table 12, where square brackets enclose optional items and braces enclose a repeated item. Here, the *module_*identifier in the module_instantiation category is a module type name, and the *module_instance_*identifier in the module_instance category is a module instance name.

Note that a single instantiation may instantiate more than one new instance of a particular module type. Each instance will have a unique replacement, derived from the module type definition using the instance name and port connections as explained in section 9.

8.1 IDENTIFIERs vs. identifiers

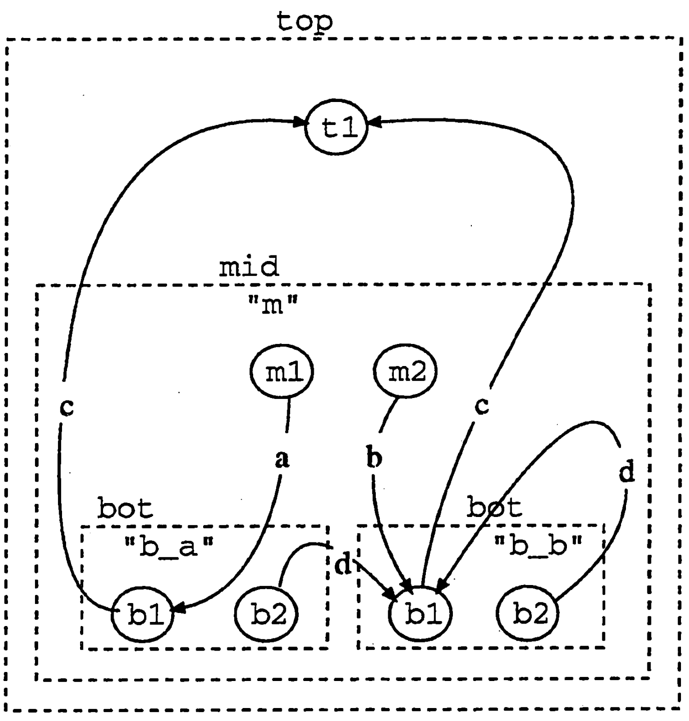
In Verilog an *IDENTIFIER* is any sequence of letters, digits, dollar signs ($) and underscore characters (_). The first character of an IDENTIFIER may not be a digit or $. IDENTIFIERs refer to unique objects within a given scope.

An *identifier* is made up of several IDENTIFIERs, separated by periods (.). Identifiers are used to refer to objects which may exist elsewhere in the module

```
  module top();
  reg t1;
  mid m();
  endmodule
```

```
  module mid();
  reg m1, m2;
  bot b_a(), b_b();
a initial m1 = bot.b;
b initial m2 = b_b.b;
  endmodule
```

```
  module bot();
  reg b1, b2;
c initial b1 = top.t;
d initial b2 = b_b.b1;
  endmodule
```



a) a downward reference by module type name, b) a downward reference by instance name, c) an upward reference by module type name, which is also a top-level reference and d) an upward reference by instance name.

Figure 1: A Simple Hierarchy with References.

hierarchy and are composed of a module type name or a module instance name followed by module instance names, ending with the object's IDENTIFIER.

There are *downward* references, *upward* references and *top-level* references, which are actually a special case of upward references. If an identifier begins with the instance name of a module instantiated in the same module that the identifier appears in, then the rest of the identifier is resolved within that instance (figure 1 b). This is a downward reference. When an identifier begins with a module type name, the rest of the identifier is resolved as a downward reference in the first instantiation of that type (figure 1 a). In either case, if no suitable module instance exists in the module where the identifier is used, then an attempt is made to resolve it in the parent modules (figure 1 c and d). This is known as an upward reference. A top-level reference begins with the module type name of a top-level module (figure 1 c) and is a *full-pathname* reference to the desired item.

Furthermore we will assume that both IDENTIFIERS and identifiers are represented as lists of ID's. An ID has the same form as an IDENTIFIER but not the same meaning. A singleton ID list represents an IDENTIFIER. Taking example identifiers and IDENTIFIERS from the unflattened code in section 10 we would represent the wire named `conx` by the ID list ["conx"] and the downward reference `sub_type.delay` by ["sub_type", "delay"].

Since, by definition, there is no hierarchy in a flattened Verilog program, we will replace all identifiers with unique IDENTIFIERs. Such unique IDENTIFIERs can be formed by concatenating the elemental IDENTIFIERs of the *full-pathname* for an object, adding some unique characters between them. These unique characters should not appear in any other IDENTIFIER, since they are in effect replacing the unique periods used in identifiers for seperating hierarchical levels. In this paper we will use a double underscore (__) as the unique characters.

module_instantiation	::=	*module*_identifier
		[parameter_value_assignment]
		module_instance {, module_instance } ;
parameter_value_assignment	::=	# (expression {, expression })
module_instance	::=	*module_instance*_identifier
		([list_of_module_connections])
list_of_module_connections	::=	ordered_port_connection
		{, ordered_port_connection }
	|	named_port_connection
		{, named_port_connection }
ordered_port_connection	::=	[expression]
named_port_connection	::=	.*port*_identifier ([expression])

Table 12: Syntax of an Instantiation.

We will assume a recursive function FlatID *id* : ID list $\mapsto$ ID list which creates these unique IDENTIFIERs from full-pathname identifiers. So, for example, FlatID["my", "child_s", "count"] returns ["my__child_s__count"], and FlatID["me"] returns ["me"].

8.2 Modelling Ports

Module ports in Verilog can exhibit quite complex behaviour and our model reflects this.

The obvious way to model a port connection is as a continuous assignment from source to sink, e.g. for an input port we would form `assign` *port expression=port connection*, and this is what the standard [2, section 5.6.6 "Port Connections"] would seem to state. Although this method works well for ports with registers as sources, experiments with Verilog-XL show that port connections involving wires as sources exhibit behaviour not covered by this approach. We are also left with the problem of modelling inout ports, which are supposedly modelled by non-strength reducing, bidirectional transistors [2].

8.2.1 Directionality

The first oddity about ports to note is that the directionality of a port is only important with respect to the type of nets connected to the port, and has no effect on the port's simulation behaviour.

Although conceptually, the designer should generate values in wires on the source side of a port and only read values from wires on the sink side, it is possible to affect a port's source by assigning values to its sink. Hence we ignore the directionality of a port completely in our model - it's only relevence is in generating a restriction forbidding `reg` types from being sinks.

8.2.2 Delays

If we consider a simple port with wires on either side we see there are several delays we can specify. Remembering that wires have two sorts of inertial delays , a *driver delay* for every driver connected to it and a *net delay* which affects signals which have passed the driver delay section, we find four delays associated with a port, two from the source and two from the sink.

The net delay for a wire is specified in its declaration. If none is given it defaults to #0, and if the wire declaration contains an assignment declaration then there is no net delay.

The driver delays are specifed with each continuous assignment used to set up a driver for the wire, and if unspecified then there is no delay associated with that driver.

My experiments have shown that in all three directional types of port where solely wires are involved, the internal net delay is ignored, and no delay is perceivable between a change in the actual elements of the port expression and the matching elements of the port connection and vice versa. Their values match so tightly that even if they are used in the same expression of wire assignment with no delay (e.g. `wire A = port_connection === port_expresion`) no difference is detected - the example would have a constant value of true, whereas `wire A = B === C` with `wire C = B` actually allows an interleaving where A is false.

Where a port expression contains a concatenation of registers and wires, the register dominates and the internal net delay given to the wire in the concatenation *does* affect the delay across the port.

8.2.3 Rules for Flat Model

Given a hierarchic model, we can produce flat code with the same behaviour by applying the following rules:

1. For a port involving registers anywhere:
   (a) All declarations and assignments are unchanged.
   (b) A continuous assignment with no driver delay is formed, assigning the source to the sink.
2. For a port involving no registers anywhere:
   (a) All declarations and assignments affecting the port connection are unchanged.
   (b) The declaration(s) of the wire(s) in the port expression are removed.
   (c) All uses of parts of the port expression become uses of the matching part of the port connection.

9 Flattening Verilog code

Flattening involves six procedures:

1. *Task Substituting*
   This is applied to each module type declaration, and textually replaces task invocations with the body of the task, leaving us with a *task-free module type declaration.*
2. *IDENTIFIER Prefixing*
   This provides a unique *prefixed module type declaration* for each instance of a module type by prefixing the instance's name to all IDENTIFIERs in the task-free module type declaration.

3. *Parameter Overriding*
   This provides an *overriden module type declaration* for each instance, by substituting parameter overrides into the prefixed module type declaration.

4. *Port Assignments*
   Produces *port code* consisting of continuous assignments and/or identifier renaming rules for each instance from the overriden module type declaration.

5. *Absorption*
   Replaces an instantiation command with the port code and contents of the overriden module type declaration for each instance named in the instantiation command.

6. *Identifier Imploding*
   Resolves any identifier references in the now flat module.

To flatten a piece of Verilog so that we are left with only top-level modules, containing no identifier references which can be resolved (i.e. only IDENTIFIERS), we firstly replace all module type declarations with their task free versions by using procedure 1. Then modules are flattened by using the following algorithm to flatten a module M.

I Flatten every module type named in the instantiations present in M. If there are no instantiations in M then it is already flat so skip the remaining steps.

II If we take all the instantiations in M, we will have t module types being instantiated. Call these $N_1, ..., N_t$. Let i_j be the number of instances of N_j. Then we need $\sum_{j=1}^{t} i_j$ instances in all. Call these I_j^i and we have:

$$\begin{matrix} I_1^1 & \cdots & I_1^{i_1} \\ & \vdots & \\ I_t^1 & \cdots & I_t^{i_t} \end{matrix}$$

Hence I_j^i is instance i of module type N_j.

An instance I_j^i is produced by applying procedures 2, 3 and 4 to the (flat) declaration of N_j, using the parameter overrides and port connections specified for the ith instance of N_j.

III Apply procedure 5 and procedure 6 to M using the instances I_j^i from the previous step.

Designs which contain looping instantiations can be easily detected, since at some point we will attempt to flatten a module as a descendant of itself.

Once all the modules have been flattened, any modules which were never instantiated are top-level modules. To produce code with a single top-level module, we flatten an imaginary module called `MAIN__` which consists of an instantiation for each top-level module named T of the form `T T__ ();`. The use of "__" in these IDENTIFIERs allows us to create unique names for this imaginary construction.

All other modules should be removed from the program. This ensures that our final code contains a single, albeit imaginary, top-level module and that top-level references between the (original) top-level modules are resolved.

The six procedures are now described in detail. Examples of their use can be found in section 10.

9.1 Task Substituting

First we take each task declaration and prefix all IDENTIFIERs which are declared in the task with the name of their enclosing task. Any IDENTIFIERs which are not prefixed in this way will later be interpreted in the scope of the enclosing module. Then we substitute every task invocation in a module with the result of surrounding the statement which forms the body of the task with appropriate blocking assignments to perform input and output. For example, an invocation of the task `my_task`, declared as:

```
task my_task;
    input a;
    output b;

    b = ~a & active;
endtask
```

of the form `my_task (z, y)` is replaced by

```
begin
    my_task__a = z;
    my_task__b = ~my_task__a & active;
    y = my_task__b;
end;
```

The task declaration itself is replaced with the declarations it contains, unchanged except that the keywords `input` and `output` become `reg`. In the example, the following declarations become part of the enclosing module:

```
reg my_task__a;
reg my_task__b;
```

This method does not allow tasks to be invoked by reference as is possible in Verilog-XL since the present Verilog definition [2] does not include this feature.

9.2 IDENTIFIER Prefixing

Every occurence of an IDENTIFIER in the flattened task-free module type declaration of m is replaced with an IDENTIFIER formed by prefixing it with the instance name of the instance we are producing. The result is a prefixed module type declaration for a single instance of m.

Note that every *IDENTIFIER* is affected, but not unresolved *identifiers*. This includes IDENTIFIERs in port expressions, declarations, statements etc. There are, of course, no instantiations in the flattened version of m to be affected by prefixing.

9.3 Parameter Overriding

A pre-requisite of parameter overriding is that the number of override expressions in a module instantiation of module type m does not exceed the number of parameters declared in the module type declaration of m. It is important that this check is made against the original module type declaration of m, since flattened versions of m may include parameter declarations absorbed from children in an arbitrary order.

We produce an overriden module type declaration for each instance of m by replacing the right hand side of the parameter declarations in the prefixed module type declaration with the override expressions in the instantiation command in the same order listed there.

9.4 Port Assignments

Each port in a module type declaration is "in", "out" or "inout".

If every net used in a port expression is declared as an `input` then the port is "in". If every net used in a port expression is declared as an `output` then the port is "out". If the nets used in the port expression are declared as a mix of `input`, `output` or `inout` types, then the port is "inout".

Note also that certain side conditions on the input/output declarations exist:

- Every identifier declared as an `input`, `output` or `inout` must be used in a port expression.
- Every identifier used in an "in" or "inout" port must be declared as a `wire` type of the same `range` as its `input`, `output` or `inout` declaration.
- Every identifier used in an "out" port must be declared as a `reg` or `net` type of the same `range` as its `output` declaration.

Each port is connected by pairing the port connections in the instantiation command with the port expressions in the overriden module type declaration.

If connections are by order then they are paired with port expressions in the same order that they are listed in the instantiation. Any null port connections will not make a pair.

If connections are by name, then port connections pair with the port expression of the same name. If a port expression consists of a single identifier, its name is the same as that identifier. A port's name can be explicitly given using the named_port_connection form (see table 12). Ports left unconnected are treated as though they had null port connections given (see above).

Each pair made is then processed according to the rules in section 8.2.3.

9.5 Absorption

Here we textually replace all instantiations in a module with the port code from 9.4 and the body of the overriden module type declaration from 9.3 for the required instances.

During absorption we produce a function $\mathcal{I} : \text{ID} \rightarrow \text{ID}_\perp$:

$$\mathcal{I}\, i = \begin{cases} i \text{ cat ``\_''} & \text{if } i \text{ is the name of an instance or task we replaced} \\ j \text{ cat ``\_''} & \text{if } j \text{ is the first instance of the module type } i \text{ we replaced} \\ \text{``''} & \text{if } i \text{ is the module type of the absorbing module} \\ \perp & \text{otherwise} \end{cases}$$

Here the infix function "cat" concatenates strings. Note that where both a module type and an instance name match i the instance name takes precedence.

9.6 Identifier Imploding

Finally, we replace every identifier in the module whose instantiations have been absorbed away with the result of applying $\mathcal{I}'$: ID list $\rightarrow$ ID list:

$$\mathcal{I}'\ id = \begin{cases} id & \text{if } \mathcal{I}\ (\text{hd } id) = \perp \\ (\mathcal{I}\ (\text{hd } id)) \text{ cat FlatID } (\text{tl } id) & \text{otherwise} \end{cases}$$

The first case leaves an identifier unchanged if it cannot yet be resolved. Hopefully such identifiers are upward or top-level references, and will be resolved later. Any identifiers left in the code after all flattening has occured are erroneous.

The second case allows us to determine which instance the reference should be resolved down from. If the identifier is correct, the IDENTIFIER produced will be that of a declared object. IDENTIFIERS are unaffected at this point.

10 An example of flattening

Suppose we have a pair of modules:

```
module top();
   reg D, clk;      wire conx, Q;

   sub_type alpha(D, conx, );
   sub_type #(sub_type.delay) beta(.result(Q), .data(conx));

   initial begin
       clk = 0;
       forever #100 clk = ~clk;
   end
endmodule

module sub_type (.data(D), .result(Q), clk);
   parameter delay = 0;
   input D, clk;   output Q;
   wire  D, clk;   reg    Q, data;

   always @(posedge (top.clk) or posedge(clk))
       begin
           Q = data;
           data = #delay D;
       end
endmodule
```

Note that sub_type has no children so flattening it has no effect but we must flatten top.

1. *Task Substituting*
   There are no tasks anywhere, so the *task free module type declarations* are the same as the module type declarations in this case.

2. *IDENTIFIER Prefixing*
   We produce two *prefixed module type declarations* for sub_type. The first is prefixed with alpha, the second with beta. For example:

   ```
   module sub_type
          (.data(alpha__D), .result(alpha__Q), alpha__clk);

   parameter alpha__delay = 0;
   input alpha__D, alpha__clk;
   output alpha__Q;
   ```

```
wire alpha__D, alpha__clk;
reg alpha__Q, alpha__data;

always @(posedge (top.clk) or posedge(alpha__clk))
    begin
        alpha__Q = alpha__data;
        alpha__data = #alpha__delay alpha__D;
    end

endmodule
```

3. *Parameter Overriding*
   We produce one *overriden module type declaration* from each of the two prefixed module type declarations.

   For the first instantiation we have no overrides, so we do not change the prefixed module type declaration of **alpha**. For the second we replace the parameter declaration with `parameter beta__delay = sub_type.delay`.

4. *Port Assignments*
   For **alpha** we connect by order, and form the three continuous assignments:

   ```
   assign alpha__D = D;
   assign conx = alpha__Q;
   ```

   Note that the IDENTIFIERs alpha__D and alpha__Q are taken from the port_list of the *overriden* module type declarations.

   For **beta** we connect by name, and form the three continuous assignments:

   ```
   assign Q = beta__Q;
   assign beta__D = conx;
   ```

5. *Absorption*
   The two instantiation commands in **top** are replaced by the port codes and the bodies of the overriden module type declarations we have. We also create $\mathcal{I}$:

$$\begin{aligned}
\mathcal{I}\,(\text{“alpha”}) &= \text{“alpha\_\_”}\\
\mathcal{I}\,(\text{“beta”}) &= \text{“beta\_\_”}\\
\mathcal{I}\,(\text{“sub\_type”}) &= \text{“alpha\_\_”}\\
\mathcal{I}\,(\text{“top”}) &= \text{“”}
\end{aligned}$$

The firt two cases are from the instance names of the instances we absorb. The third case returns the first named instance of the instance type we absorbed. The third and fourth cases are for references which refer by module type, either to a direct parent, in this case "top", or a sibling module type, "sub_type".

6. *Identifier Imploding*
   In this one level hierarchy we rather hope that all unresolved identifiers resolve in the top-level module, which they do.

   Note particularly the reference to `subtype.delay` in the override of parameter `delay` in instance `beta`. Once the overriden module type definition for beta is absorbed $\mathcal{I}$("sub_type") provides "alpha", the correct prefix in this case.

   The resulting flat module is:

```
module top();

// top's original declarations are:
reg D, clk;
wire conx, Q;

// sub_type alpha(D, conx, );
// is replaced by port code:
assign alpha_D = D;
assign conx = alpha__Q;
// and overriden module type declarations:
parameter alpha__delay = 0;
wire alpha_D, alpha__clk;
reg alpha__Q, alpha__data;

always @(posedge (clk) or posedge(alpha__clk))
    begin
        alpha__Q = alpha__data;
        alpha__data = #alpha__delay alpha_D;
    end

// sub_type #(sub_type.delay) beta(.result(Q), .data(conx));
// is replaced by port code:
assign Q = beta__Q;
assign beta_D = conx;
// and overriden module type declarations:
parameter beta__delay = alpha__delay;
wire beta_D, beta__clk;
reg beta__Q, beta__data;

always @(posedge (clk) or posedge(beta__clk))
    begin
        beta__Q = beta__data;
        beta__data = #beta__delay beta_D;
    end

// top's original code is:
initial begin
    clk = 0;
    forever #100 clk = ~clk;
end

endmodule
```

If `top` were instantiated elsewhere, then this declaration would be used in flattening its parents. Since it is not we now have a flat version of a top-level module. This is instantiated in our imaginary `MAIN__` module like so:

```
module MAIN__();
top top__ ();
endmodule
```

Flattening this will give us the fully flattened version of the code.

References

[1] R.J. Boulton. Computer Language Reasoning Tool. Available at http://www.dai.ed.ac.uk/daidb/staff/personal_pages/rjb/claret/.

[2] IEEE Standards Department. *P1364 IEEE Draft Verilog HDL Reference Manual.* IEEE, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331., October 1995. Draft standards document.

[3] M.J.C. Gordon. The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, San Diego, California, June 26-29 1995. http://www.cl.cam.ac.uk/users/mjcg/Verilog.

[4] D.J. Greaves and M.J.C Gordon. Checking equivalence between synthesised logic and non-synthesisable behavioural prototypes. A three year EPSRC research project On http://www.cl.cam.ac.uk/users/mjcg/Verilog.

[5] Open Verilog International. *Verilog Hardware Description Language Reference Manual (LRM)*, version 1.0 edition, November 1991.

[6] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer, Second edition, 1995.

[7] The Verilog-XL simulator. Available from Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, email: talkverilog@cadence.com.

[8] The Veriwell simulator from Wellspring Solutions. Available for downloading on ftp://iii.net:/pub/pub-site/wellspring/.

[9] The Viper/free simulator. Available free from interHDL, Inc., 4984 El Camino Real, Suite 210, Los Altos, CA. 94022-1433, email: info@interhdl.com.