

A Mathematical Level/Strength Model for Synthesizing STD_LOGIC_1164 Values

James H. Vellenga

Viewlogic Systems, Inc., 293 Boston Post Road West
Marlboro, MA 01752-4615, USA

Abstract

STD_LOGIC_1164 values let a VHDL user represent known and unknown signal values based on two levels (0 and 1) and three strengths. This paper formalizes a two-part level/strength model for STD_LOGIC_1164 values to develop strength-related transformations that can be used in synthesis. We represent unknowns as a set of choices among possible well-defined ("real") values. This lets us formally define "implementation" as the process of narrowing the set of choices. Defining "strong equivalence" (levels and strengths) and "weak equivalence" (levels only) then allows one to determine, for example, under what conditions "orland" logic can be considered (logically) equivalent to a bus with weak and high-impedance drivers. The formalism is used to study the composition of resolution functions, and to compare them to equivalent hardware implementations.

Introduction

In the early 1980s, switch-level MOS logic simulators began modeling digital logic signals as mixtures of levels (high, low) and strengths (strong, weak, and high-impedance) ([1], [2], [3]). These models allowed the user to create and simulate buses with multiple drivers, some of which could be switched off (high-impedance) or provide a pull-up or pull-down (weak driver). The mechanism for determining the signal values that results from the multiple drivers is called a *resolution function*.

The VHDL simulation semantics provides a way of representing resolution functions, but the STANDARD library in VHDL defines only two binary enumeration types (BIT and BOOLEAN) for representing bit values, and neither type represents unknown values or signal strengths [4]. Practitioners quickly began proposing multi-valued types to meet this need (see, for example, [5]), finally agreeing on the definition of the STD_LOGIC_1164 nine-state standard value system [6].

Meanwhile, users and vendors of logic synthesis have been converging on an agreement to use assignments to the STD_LOGIC_1164 high-impedance value 'z' to infer the existence of three-state drivers. An unpublished posi-

tion paper from DASC's VHDL Synthesis SIG proposes that "In a write reference, the assignment of the scalar value 'z' ... would imply a three-state buffer that was disabled by the condition controlling the assignment" [7].

Synthesis may also introduce three-state drivers in the process of mapping generic logic to specific technologies. In our own work developing the SilcSyn synthesis system [8], we have found that in certain technologies it is beneficial to implement a complex multiplexer as a bus with multiple three-state drivers that are conditionally enabled.

The increasing use of three-state drivers and high-impedance logic within logic synthesis raises at least three questions:

- 1) How can one represent common hardware resolution functions (wired-or, wired-and, and ordinary three-state) in VHDL input for synthesis?
- 2) How should one handle the composition of resolved signals (one signal driving another) either within a design entity or between levels of a hierarchy of design entities?
- 3) When can a synthesis tool treat ordinary combinational logic as a bus of three-state drivers, and vice versa?

This paper develops a formalism based on logic levels and strengths to represent the STD_LOGIC_1164 values, and defines unknown values as a set of choices among values that a circuit may actually achieve. It introduces the concept of "implementation" as the process of selecting among (constraining) the set of choices, and defines two levels of equivalence (strong and weak) that allow us to address the questions listed above.

Representation of single bits

We represent a *bit* as an ordered pair $a = [a^l, a^s]$. The level a^l is either

- -1 (denoting a "low," "false," or "0" value) or
- 1 (denoting "high," "true," or "1").

The strength a^s of the bit is either

- 2 for a signal source that has a normal ("forcing") drive,
- 1 for a weak source such as a pull-up or pull-down, or

- 0 for a high-impedance source such as a disabled three-state driver.

For simulation purposes, STD_LOGIC_1164 represents certain values as “unknowns” or “don’t cares.” The formalism of this paper represents such values as a set of choices: that is, as a collection of real values that the unknown value can have. The symbol “ \otimes ” denotes a choice; thus

$$[(-1 \otimes 1), 2]$$

is either $[-1, 2]$ or $[1, 2]$ (strong unknown), while

$$[-1, 2] \otimes [1, 1]$$

is either $[-1, 2]$ or $[1, 1]$ (mixed-strength unknown). Note that “ \otimes ” is overloaded so that it can apply when both operands are bits or when both operands are levels—or for that matter, when both operands are more complex types of values.

With these conventions, we can represent the STD_LOGIC_1164 values as shown in Table 1:

$0 = [-1, 2]$ '0'	$x = [-1 \otimes 1, 2] = 0 \otimes 1$ 'x', 'u', '-'	$1 = [1, 2]$ '1'
$l = [-1, 1]$ 'L'	$w = [-1 \otimes 1, 1] = l \otimes h$ 'w'	$h = [1, 1]$ 'H'
	$z = [-1 \otimes 1, 0]$ 'z'	

Table 1: STD_LOGIC_1164 Values

For synthesis purposes, the formalism treats 'x', 'u', and '-' as equivalent, even though they represent different intents on the part of the designer. This is consistent with their treatment as arguments to Boolean logic functions in the STD_LOGIC_1164 package [6], and consistent with the proposals of the VHDL Synthesis SIG [7].

The italicized values 0 , 1 , l , and h are literals representing the known (real) strong and weak values. The italicized literals x , w , and z represent values with unknown levels, and hence are equivalent to choices between corresponding real values. However, the values $[-1, 0]$ and $[1, 0]$ have no real meaning (since they represent a high-impedance source, the source cannot drive the signal to any particular level) and hence no STD_LOGIC_1164 values. (Alternatively, one could regard them as representing the actual levels that a signal is left in when all its drivers are disconnected.)

A different way to look at these values is to regard them as a set of possible actual values, with each known value being represented by a set containing exactly one member. An unknown value is then a set with more than one member—that is, the set contains the values that might occur at this point in the actual circuit. In this way, a choice may be regarded as the union of two sets.

Because (in this view) the choice operator is equivalent to the union operator, it is clear that the choice operator is

commutative, associative, and idempotent (makes multiple equal choice values equivalent to a single instance of that value). That is,

$$a \otimes a = a$$

$$a \otimes b = b \otimes a$$

$$(a \otimes b) \otimes c = a \otimes (b \otimes c) = a \otimes b \otimes c$$

Thus,

$$1 \otimes 0 \otimes 1 = 0 \otimes 1 = x$$

$$x \otimes z = (0 \otimes 1) \otimes z = 0 \otimes 1 \otimes z$$

We define the choice operator as distributive over other functions:

$$f(a \otimes b) = f(a) \otimes f(b) \quad (\text{EQ 1})$$

The reason for this is that, in a real circuit at any one instant in time, the argument takes on one and only one of the choice values, and the function then returns the corresponding “output” value. So, for example,

$$\begin{aligned} \min(-1 \otimes 1, 1) &= \min(-1, 1) \otimes \min(1, 1) \\ &= -1 \otimes 1 \end{aligned}$$

$$\begin{aligned} \min(-1 \otimes 1, -1) &= \min(-1, -1) \otimes \min(1, -1) \\ &= -1 \otimes -1 = -1 \end{aligned}$$

A function of multiple arguments which are independent choice sets returns a value which is the product of distributing the function over each of the choice sets:

$$f(a \otimes b, c \otimes d) = f(a, c) \otimes f(a, d) \otimes f(b, c) \otimes f(b, d)$$

If the arguments are not independent, then not all the function values occur in reality. For example, if $a = -b$ and $b = -1 \otimes 1$, then

$$\begin{aligned} a \times b &= (1 \otimes -1) \times (-1 \otimes 1) \\ &= -1 \otimes 1 \otimes -1 \otimes 1 = -1 \otimes 1 \end{aligned}$$

whereas in reality $a \times b$ is always -1 , which is a subset of the values $-1 \otimes 1$ resulting from treating the arguments a and b as independent. This is a general rule: if one computes a set of function return values by treating unknown arguments as independent, the values that can actually occur are a subset (but not necessarily a proper subset) of the computed values.

Using the values -1 and 1 to represent logic levels leads to a particularly compact arithmetic representation of the basic logic functions, as shown in Table 2. Each of the logic functions produces a “forcing strength” result, corresponding to what a CMOS logic gate would do (and also corresponding to the behavior of the functions defined in STD_LOGIC_1164). We have included the buffer function TO_X01 to represent a logic gate that acts as a buffer.

VHDL	Functional Form	Definition
not A	$not(a)$	$[-a^l, 2]$
A and B	$and(a, b)$	$[min(a^l, b^l), 2]$
A or B	$or(a, b)$	$[max(a^l, b^l), 2]$
A xor B	$xor(a, b)$	$[-a^l \times b^l, 2]$
TO_X01(A)	$buffer(a)$	$[a^l, 2]$

Table 2: Level/Strength Representation of Logic Functions

The arithmetic form provides a convenient proof of the DeMorgan Laws:

Functional Form	Arithmetic Form
$not(and(a, b)) = or(not(a), not(b))$	$-min(a^l, b^l) = max(-a^l, -b^l)$
$not(or(a, b)) = and(not(a), not(b))$	$-max(a^l, b^l) = min(-a^l, -b^l)$
$not(xor(a, b)) = xor(not(a), b) = xor(a, not(b))$	$-(-a^l \times b^l) = -(-a^l) \times b^l = a^l \times b^l$

Because choices are distributive over functions, the DeMorgan Laws apply to unknowns in this representation as well.

The result of the logic functions depends only on the levels of the operand values, and not on their strengths. That is, each logic function converts operand values that are *weakly equivalent* (have the same levels) into result values that are *strongly equivalent* or equal (have the same levels and same strengths).

Vectors and sequences

In order to deal with resolution functions, we also have to represent vectors or sequences of bits. We define a *bit vector* as a sequence

$$A = \{a_i\}, i = 1, \dots, |A|$$

where $|A|$ is the number of elements in A . We can also represent a bit vector as an explicit sequence of elements such as

$$\{1, 1, x, 0\}$$

A subsequence $\{a_i | c(a_i)\}$ is a sequence made up of those elements of $\{a_i\}$ that satisfy the condition $c(a_i)$. If an element of $\{a_i\}$ is a choice, and not all elements of the choice satisfy $c(a_i)$, then $\{a_i | c(a_i)\}$ represents a choice of subsequences:

$$\{a, b, c \otimes d, e | f(x)\} = \{a, b, c, e | f(x)\} \otimes \{a, b, d, e | f(x)\}$$

The operator \parallel concatenates either two sequences, or a bit to a sequence

$$\{a_i\} \parallel \{b_j\}$$

$$z \parallel \{a_i\}$$

We also define a *signal* as a (continuous) sequence of bit values over time:

$$f(t)$$

The logical functions also apply to signals. For example,

$$not(a(t)) = [-a^l(t), 2]$$

and so on.

For the purposes of this paper, we use the term *circuit* to mean an arbitrary collection of signals ($\{a_i(t)\}$).

Resolution functions

In order to represent resolution functions, we define the functions *strongest* and *resolved* operating on sequences of bits. The usefulness of the definitions depends on the function being independent of the order of its arguments—or in other words, that the resolution functions are commutative and associative in the elements of their argument sequences.

For a sequence in which every element has an unmixed strength (that is, in which every element is either a known value or is a choice of values that all have the same strength), *strongest* is the subsequence made up of the strongest elements of the sequence:

$$strongest(\{a_i\}) = \{a_i | a^s_i = max(\{a^s_i\})\}$$

If one or more elements of the sequence have mixed strengths, we separate the sequence into a choice of subsequences, each of which contains elements of unmixed strengths. *strongest* is then the choice of the *strongest* of the subsequences:

$$\begin{aligned} strongest(\{h, l, z \otimes l, z\}) &= \\ strongest(\{h, l, z, z\} \otimes \{h, l, l, z\}) &= \\ strongest(\{h, l, z, z\}) \otimes strongest(\{h, l, l, z\}) &= \\ &= \{h, l\} \otimes \{l\} \end{aligned}$$

The function *resolved* emulates the `RESOLVED` function from `STD_LOGIC_1164`. If the strongest elements of the function argument have the same value, *resolved* returns that value; otherwise, it returns an unknown value of the same strength as that of the strongest elements. Formally, $resolved(\{a_i\})$ returns a level which is the choice of all the levels of $strongest(\{a_i\})$ and a strength equal to the maximum strength of any element of $\{a_i\}$. Thus, if

$$\{b_j\} = strongest(\{a_i\})$$

then

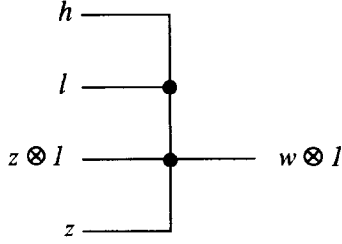


Figure 1: Resolution Function with Element of Mixed Strength

$$\begin{aligned} \text{resolved}(\{a_i\}) & \\ &= b_1 \quad \text{if } (b_j = b_1), \forall b_j \\ &= [-1 \otimes 1, b^s_1] \quad \text{otherwise} \end{aligned}$$

Here again, if the argument sequence contains elements of mixed strength, *resolved* is defined by applying *resolved* to each of the sequences created by expanding the choices among the elements:

$$\begin{aligned} \text{resolved}(\{h, l, z \otimes 1, z\}) &= \\ (h \otimes l) \otimes 1 &= w \otimes 1 \end{aligned}$$

This (electrically unsound) situation is shown in Figure 1.

The `RESOLVED` function in `STD_LOGIC_1164` is defined on elements of unmixed strength (because it is defined on `STD_ULOGIC` values), so that its domain is a subset of the domain of *resolved*.

Implementation

Any real circuit synthesized from a description in which some signal values are unknown over periods of (simulated) time will in fact have real values for those signals at any given instant in real time. For this reason, we formally define the concept of *implementation* to represent the process of replacing uncertain values from a specification of a circuit by a subset of real values that the uncertain values can take on. Our assumption is that, if a user is satisfied with a circuit in which a primary output $p(t)$ has an unknown value (such as x) over an interval of time:

$$p(t) = x = 0 \otimes 1, t_0 < t < t_f$$

then he or she will also be satisfied with a circuit in which the value of that primary output is 0 or 1 over that same interval:

$$\begin{aligned} p'(t) &= 1, t_0 < t < t_f \\ p'(t) &= p(t) \text{ otherwise} \end{aligned}$$

for example.

Because the logic functions, as noted above, convert weakly equivalent operand values (equal levels) into strongly equivalent result values (equal levels and strengths), it is useful to define the concepts of strong and weak implementation.

A bit value a' *strongly implements* (“ \Leftarrow ”) a bit value a iff every choice in a' is also a choice in a . For example,

$$1 \Leftarrow x \text{ because } x = 0 \otimes 1.$$

but it is also true, for example, that one set of choices can implement another, as in

$$0 \otimes 1 \Leftarrow 0 \otimes 1 \otimes z$$

Similarly, a signal s' *strongly implements* a signal s iff $s'(t) \Leftarrow s(t)$ for every instant t .

Weak implementation is like strong implementation, except that it doesn't preserve the strengths of the choice values. That is, a value a' *weakly implements* (“ \Leftarrow ”) a value a

$$a' \Leftarrow a$$

iff for every choice in a' , there is a choice in a that has the same level, but not necessarily the same strength. For example,

$$0 \otimes 1 \Leftarrow h \otimes 0$$

And, as with strong implementation, a signal s' *weakly implements* a signal s iff $s'(t) \Leftarrow s(t)$ for every instant t .

Either kind of implementation may reduce an “unknown” level (set of choices) to a “known” (single) level.

Strong implementation implies weak implementation, but not necessarily vice versa¹:

$$(a' \Leftarrow a) \subseteq (a' \Leftarrow a) \quad (\text{EQ 2})$$

Both forms of implementation are transitive:

$$(a'' \Leftarrow a') \wedge (a' \Leftarrow a) \subseteq (a'' \Leftarrow a)$$

$$(a'' \Leftarrow a') \wedge (a' \Leftarrow a) \subseteq (a'' \Leftarrow a)$$

but also

$$(a'' \Leftarrow a') \wedge (a' \Leftarrow a) \subseteq (a'' \Leftarrow a)$$

because of (EQ 2).

Two circuits are weakly equivalent

$$\{a'_i(t)\} \leftrightarrow \{a_i(t)\}$$

iff $\{a'_i(t)\} \Leftarrow \{a_i(t)\}$ and $\{a_i(t)\} \Leftarrow \{a'_i(t)\}$. Two circuits are strongly equivalent if their output choices are equal at all times. That is, a strong equivalence $\{a'_i(t)\} = \{a_i(t)\}$ is itself equivalent to

$$(\{a'_i(t)\} \Leftarrow \{a_i(t)\}) \wedge (\{a_i(t)\} \Leftarrow \{a'_i(t)\}).$$

The logic functions defined above convert weak implementation into strong implementation. That is, if $a' \Leftarrow a$ and $b' \Leftarrow b$, then

$$\text{not}(a') \Leftarrow \text{not}(a)$$

$$\text{buffer}(a') \Leftarrow \text{buffer}(a)$$

$$\text{and}(a', b') \Leftarrow \text{and}(a, b)$$

$$\text{or}(a', b') \Leftarrow \text{or}(a, b)$$

$$\text{xor}(a', b') \Leftarrow \text{xor}(a, b)$$

¹ We use “ \Leftarrow ” to mean “implies” and “ \wedge ” to represent a logical “and.”

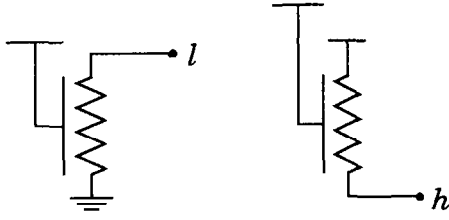


Figure 2: Passive Pull-Down and Pull-Up

This is so because the values output by a logic gate depend on the levels of the inputs, but are independent of their strengths. So a subset of choices of input values generates a corresponding subset of choices of output values, but the strength of the output value is always 2.

What this means for synthesis is that if a user wants a strong implementation for the output of a logic gate, the synthesis tool can use a weak implementation for either of the inputs to the logic gate. And weakly equivalent inputs to a logic gate produce strongly equivalent outputs.

The concept of strong implementation means that an assignment to an unknown value such as 'x' is effectively an assignment to a "don't care" value. The concept of weak implementation effectively applies the concept of "don't care" values to strengths as well.

Representation of hardware elements

With the formalism of this paper, one can represent hardware elements used in three-state design as functions, or in some cases, as constants.

The passive pull-down and pull-up elements shown in Figure 2 provide weak constant sources, and are represented as l and h respectively.

A three-state driver (Figure 3a) can be represented as the function

$$o = 3st(i, e)$$

where

$$\begin{aligned} 3st(i, e) &= [i^l, 2] = \text{buffer}(i), & e^l &= 1 \\ &= z, & e^l &= -1 \end{aligned}$$

so that when the enable has an unknown level,

$$e^l = -1 \otimes 1,$$

then by (EQ 1), the output is a choice between a high impedance value z and a strength-2 normal value:

$$3st(i, e) = z \otimes [i^l, 2], e^l = -1 \otimes 1$$

This choice cannot be represented by a single `STD_LOGIC_1164` value, but we maintain it for convenience in representing the actual resolution of hardware values.

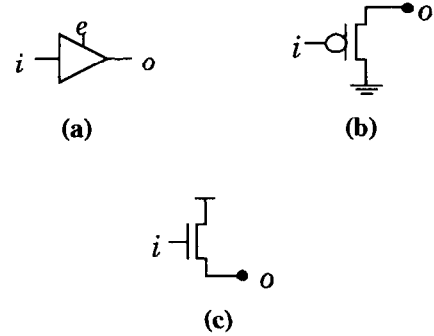


Figure 3: Active Three-State Elements

Figures 3b and 3c represent active pull-downs and pull-ups—pull-downs and pull-ups that can be enabled and disabled. These are logically equivalent to three-state drivers for which the input i is tied to 0 or 1. Thus, the equation for an active pull-down is

$$o = 3st(0, \text{not}(i))$$

while the equation for the active pull-up is

$$o = 3st(1, i)$$

Like the logic functions, the $3st$ function converts a weak implementation on its inputs into a strong implementation on its output. That is, if $i' \leftarrow i$ and $e' \leftarrow e$, then $3st(i', e') \leftarrow 3st(i, e)$.

Equivalence of hardware implementations

Figure 4 shows one schematic representation of a hardware implementation of a wired-and resolution function. If any input i_j is low, the corresponding active pull-down is enabled and the resulting signal is low. Otherwise, all the active pull-downs are disabled, and the pull-up brings the output signal high. Formally,

$$\begin{aligned} \text{wired_and}(\{i_j\}) & \\ &= \text{resolved}(h \parallel \{3st(0, \text{not}(i_j))\}) \end{aligned}$$

and $\text{wired_and}(\{i_j\})$ is

- h if every $i_j^l = 1$ (high);
 - 0 if any $i_j^l = -1$ unambiguously;
- and hence by (EQ 1)
- $h \otimes 0$ if some $i_j^l = -1 \otimes 1$ (unknown) and no $i_j^l = -1$.

In the same way, one can represent an n -input "and" gate as an and function defined on a vector of length n . That is, $\text{and}(\{i_j\})$ is

- 1 if every $i_j^l = 1$ (high);

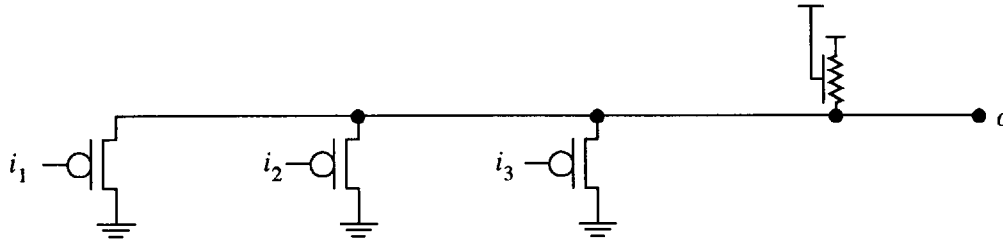


Figure 4: Wired-And Resolution Function

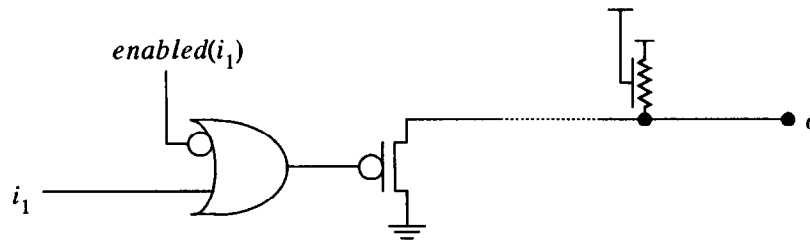


Figure 5: Wired-And with Disconnection

- 0 if any $i_j^l = -1$ (low);
and hence

- x if some $i_j^l = -1 \otimes 1$ (unknown) and no $i_j^l = -1$.

Comparing output levels for any combination of input levels shows that $wired_and(\{i_j\})$ and $and(\{i_j\})$ are weakly equivalent:

$$and(\{a_j\}) \leftrightarrow wired_and(\{a_j\})$$

so that they can be used interchangeably as inputs to, say, a logic function or three-state driver.

How can this be used in synthesizing circuits specified by VHDL source code? To answer this question, we have to make some assumptions about how a resolved signal is represented in VHDL. Our assumptions are as follows (some of these assumptions are VHDL requirements):

- The same signal (of a resolved type) can be written by multiple VHDL processes (or by concurrent signal assignment statements that are equivalent to processes) or by component instantiation statements.
- Each process or component instance acts as at most one source for the signal.
- Each source emits only STD_LOGIC_1164 values. That is, each source can emit a known value such as 1, 0, h, or l, or it can emit a single-strength unknown such as x or z, but it cannot emit a mixed-strength unknown

such as $h \otimes 0$ because STD_LOGIC_1164 doesn't include such values.

- Writing a 'z' from a process is like disconnecting the process as a source. (This interpretation is consistent with the VHDL definition for signals of kind `buffer`, but not for signals of kind `register`.)
- Each resolution function is commutative and associative in the elements of its argument sequence.

In order to represent the possibility of disconnecting sources via assignment, we define the function $enabled(i)$ such that

$$enabled(i) = 0, \quad i^s = 0 \\ 1, \quad \text{otherwise}$$

That is, the value of $enabled$ is 1 iff the input value is not z.

With this function, we can now represent a wired-and that can be disconnected as shown in Figure 5. Figure 5 shows a single source with a value i_1 that can be disconnected, along with a pull-up that supplies the weak level-1 value h . When i_1 has the value z, the synthesizer must disconnect the source as represented by the following enabled wire-and function:

$$o = wired_and^{(e)}(\{i_i\}) \\ = resolved(h \mid \{o_i\}),$$

where

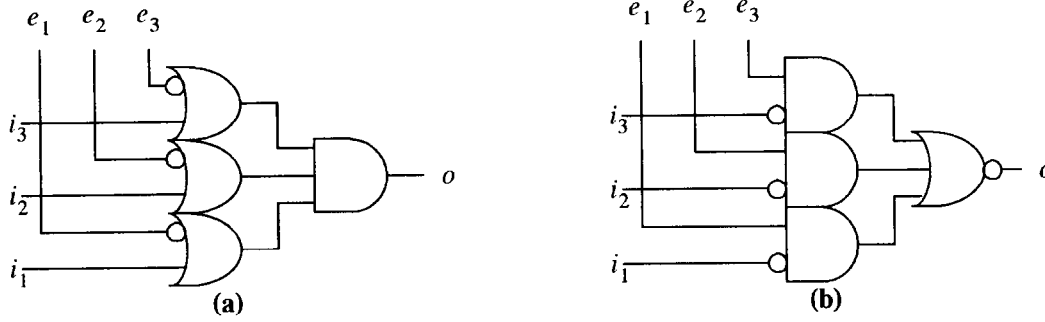


Figure 6: Weak Equivalents of Wired-And with Disconnection

$$o_i = 3st(0, not(or(i_i, not(enabled(i_i))))))$$

By the same kind of analysis used to show that $wired_and(\{i_j\})$ and $and(\{i_j\})$ are weakly equivalent, one can show that the combinational logic gates of Figure 6a are weakly equivalent to the wired-and with disconnect. That is,

$$wired_and^{(c)}(\{i_j\}) \leftrightarrow wired_and^{(e)}(\{i_j\})$$

where

$$\begin{aligned} &wired_and^{(c)}(\{i_j\}) \\ &= and(\{or(i_i, not(enabled(i_i)))\}) \end{aligned}$$

By DeMorgan's laws, also

$$\begin{aligned} &wired_and^{(c)}(\{i_j\}) \\ &= not(or(\{and(not(i_i), enabled(i_i))\})), \end{aligned}$$

so that Figure 6a and Figure 6b, which are strongly equivalent to each other, are weakly equivalent to $wired_and^{(e)}(\{i_j\})$. That is, either set of combinational logic gates could be used as a weak implementation of a wired-and with disconnect.

A possible VHDL representation of a wired-and resolution function is

```
function WIRED_AND(I: STD_ULOGIC_VECTOR)
  return STD_ULOGIC is
  variable RESULT: STD_ULOGIC := 'H';
begin
  for J in I'RANGE loop
    case X01Z'(TO_X01Z(I(J))) is
      when '0' => return '0';
      when 'X' => RESULT <= 'X';
      when others => null;
    end case;
  end loop;
  return RESULT;
end WIRED_AND;
```

In terms of our formalism, $WIRED_AND$ returns

- 0 if any input has unmixed level 0;

- x if no input has unmixed level 0, and some input has a mixed-level (unknown) value other than z; and
- h otherwise (all inputs are z, 1 or h, or disconnected).

Unfortunately, $WIRED_AND$ does not behave like other functions in our formalism in that it does not distribute over choices on its inputs. For example, when other inputs are z, an input i that has the value $x = 0 \otimes 1$ does not return a choice that is either the result when i is 0 (namely, 0) or the result when i is 1 (namely, h), but rather returns $0 \otimes 1$. However, in a universe in which every source has a single STD_LOGIC_1164 value (as required by our assumptions), both $wired_and^{(e)}(\{i_j\})$ and $wired_and^{(c)}(\{i_j\})$ are weak implementations of $WIRED_AND$, while $wired_and^{(e)}(\{i_j\})$ gives the same results as $WIRED_AND$ for all known and high-impedance source values. But any real circuit (that is, any circuit that in fact propagates actual values in a well-defined fashion) provides only a weak implementation of $WIRED_AND$ under conditions that produce an 'X' output.

We can also investigate the relationships between resolution functions that are less closely related. Figure 7 shows an implementation of a simple three-state bus without pull-ups or pull-downs. This function has the representation

$$\begin{aligned} &three_state(\{i_j\}) \\ &= resolved(\{3st(i_i, enabled(i_i))\}) \end{aligned} \quad (EQ\ 3)$$

so that

$$\begin{aligned} &three_state(\{i_j\}) \\ &= z \quad \text{if every } i_j = z. \\ &= 1 \quad \text{if there are one or more } i'_j = 1, \\ &\quad \text{and all remaining } i_j = z. \\ &= 0 \quad \text{if there are one or more } i'_j = -1, \\ &\quad \text{and all remaining } i_j = z. \\ &= x \quad \text{otherwise.} \end{aligned}$$

By analyzing the various combination of inputs, one can show that

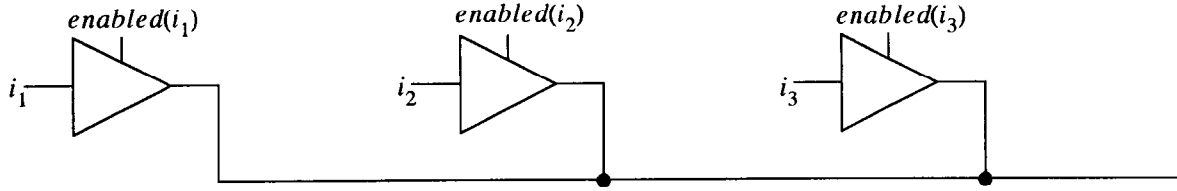


Figure 7: Three-State Bus

$$wired_and^{(e)}(\{i_j\}) \leftarrow three_state(\{i_j\})$$

That is, for any $\{i_j\}$ for which $three_state(\{i_j\})$ emits (unambiguously) a 1 or a 0, $wired_and^{(e)}(\{i_j\})$ emits an h or a 0. However, the reverse is not true: there are cases (such as when all inputs are disconnected or z) in which the level of $three_state(\{i_j\})$ is ambiguous but the level of $wired_and^{(e)}(\{i_j\})$ is not. It follows also that

$$wired_and^{(c)}(\{i_j\}) \leftarrow three_state(\{i_j\}).$$

That is, the wired-and resolution functions are weak implementations of the three-state resolution function, but not vice versa.

The same kind of analysis can be extended to wired-or resolution functions. Wired-and and wired-or resolution functions are both weak implementations of the three-state function, but they are not weak implementations of each other.

Composition

The formalism can also be used to analyze and transform the composition of resolution functions. Resolution functions are composed when the output of one feeds the input of another. We do not have a comprehensive theory of such transformations, but a few examples are given to show how the formalism may be used.

With some effort, it can be shown that

$$resolved(A \parallel B) = resolved(A \parallel \{resolved(B)\})$$

It follows, for example, that

$$\begin{aligned} & three_state(\{a_i\} \parallel \{three_state(\{b_j\})\}) \\ &= three_state(\{a_i\} \parallel \{b_j\}) \end{aligned} \quad (EQ\ 4)$$

What this means is that one can wire the output of one three-state bus as an input to another. However, since the hardware realization of the three-state function requires separate data and enable lines, we must first study the composition of enables. From the definition of the enabled function:

$$\begin{aligned} enabled(i) &= 0, & i &= z \\ &1, & & \text{otherwise} \end{aligned}$$

and from the definition of the $three_state$ function (EQ 3), it follows that

$$three_state(\{a_i\}) \neq z$$

if and only if there exists an a_i such that $enabled(a_i) = 1$. Hence,

$$\begin{aligned} & enabled(three_state(\{a_i\})) \\ &= or(\{enabled(a_i)\}) \end{aligned} \quad (EQ\ 5)$$

That is, the equivalent enable for the output of a three-state bus is the logical “or” of the enables corresponding to the inputs.

Figure 8 shows a transformation corresponding to this for composing three-state resolution functions across a hierarchical boundary. In order to preserve the high-impedance condition when using the component instance port as a source for a three-state driver in its own right, one has to collect up the enables of the contributing elements. In this case, it is obvious that the three-state driver T and the “or” of enables could be omitted altogether—and that is the significance of (EQ 4).

One can also analyze what happens when the output of one resolution function is used as input to a different one. Consider the case in which a signal defined with a three-state resolution function acts as a driver to a signal with a wired-and resolution function, as shown in Figure 9. Formally,

$$\begin{aligned} & wired_and^{(e)}(\{a_i\} \parallel \{three_state(\{b_j\})\}) \\ &= resolved(h \parallel \{x_i\} \parallel y) \end{aligned}$$

where

$$x_i = 3st(0, not(or(a_i, not(enabled(a_i))))),$$

where

$$y = 3st(0, not(or(three_state(\{b_j\}), not(w))))),$$

and where

$$\begin{aligned} w &= enabled(three_state(\{b_j\})) \\ &= or(\{enabled(b_j)\}) \end{aligned}$$

where the last equality follows from (EQ 5). This composition is illustrated in Figure 9.

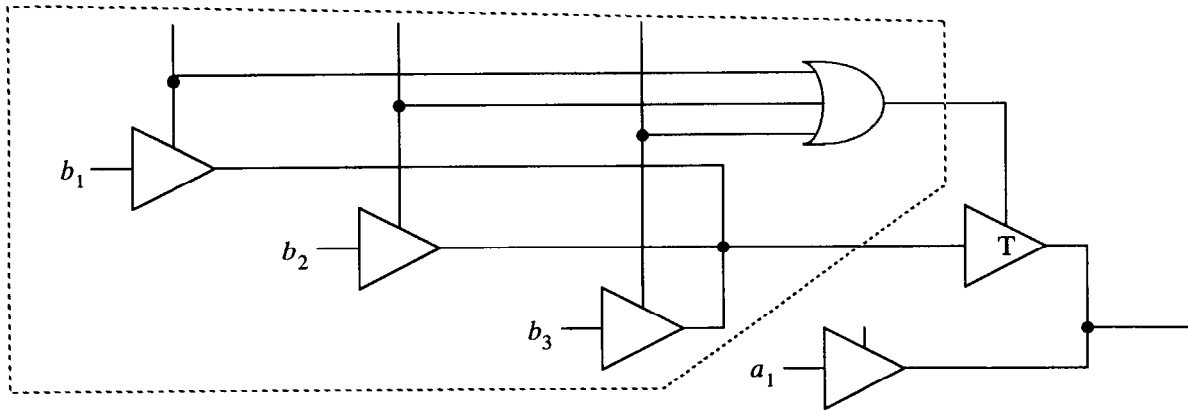


Figure 8: Composition of Three-State Resolution Functions

Conclusions

We have developed a formalism that represents STD_-LOGIC_1164 values as ordered pairs of levels and strengths. The formalism permits formal validation of transformations not only of conventional Boolean logic functions, but also of “three-state” type operations such as pull-ups, pull-downs and three-state drivers.

In the formalism, an unknown value is represented as a choice among known values. This makes possible the concept of “implementation,” which is defined as selecting a subset of the possible known values. Weak implementation implements a subset of the levels, while strong implementation selects a subset that also preserves strengths.

The idea of weak implementation shows that one can use the concept of “don’t care” values not only to synthesize levels but also to synthesize strengths. Moreover, an assignment to an unknown value such as ‘x’, because of the concept of implementation, becomes effectively an assignment to a “don’t care” value.

Weak and strong implementation implies the corresponding concepts of weak and strong equivalence, to which the formalism gives a precise definition. Because many logic elements (such as the logical functions and three-state drivers) convert weakly equivalent inputs into strongly equivalent outputs, we have shown that it is possible in many cases to convert buses with high-impedance inputs into ordinary combinational logic, and vice versa.

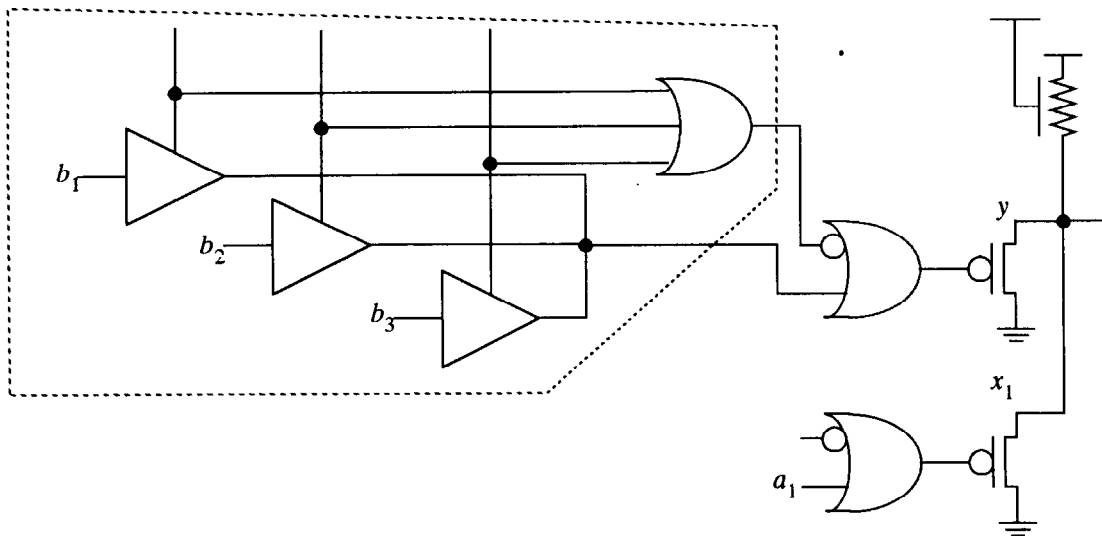


Figure 9: Composition of Differing Resolution Functions

without sacrificing strong equivalence of the overall circuit.

Finally, the formalism provides a precise meaning for the composition of the same or different resolution functions, and allows one to analyze and transform the corresponding hardware implementations in a formal fashion.

References

- [1] Randal E. Bryant, "An Algorithm for MOS Logic Simulation," *Lambda*, Fourth Quarter 1980, pp. 46-53.
- [2] H. De Man, "Mixed-Mode Analysis and Simulation Techniques for Top-Down MOSVLSI Design," *Proceedings 1981 European Conference on Circuit Theory and Design*, R. Boite and P. Dewilde, eds., North-Holland, Amsterdam, 1981, pp. 5-10.
- [3] Dündar Dumlogöl, Hugo J. De Man, Piet Stevens, and Guido G. Schrooten, "Local Relaxation Algorithms for Event-Driven Simulation of MOS Networks Including Assignable Delay Modeling," *IEEE Trans. on Computer-Aided Design*, vol. CAD-2, no. 3, July 1983, pp. 193-202.
- [4] IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronic Engineers, Inc., New York, 1988.
- [5] Ghulam M. Nurie and Paul J. Menchini, "VHDL Model Portability," *High Performance Systems*, vol. 10, no. 7, July 1989, pp. 76-85.
- [6] IEEE Computer Society, IEEE Std 1164-1993, *IEEE Standard Multivalued Logic System for VHDL Model Interoperability (Std logic_1164)*, The Institute of Electrical and Electronic Engineers, Inc., New York, 1993.
- [7] "Interpretation of the Standard Logic Type for Synthesis," DASC VHDL Synthesis SIG, fall 1993, unpublished.
- [8] Racal-Redac, Inc., *SilcSyn VHDL Synthesis Reference Guide*, UM-14-331-U-03, Mahwah, NJ, 1993.