

Be Careful with Don't Cares

Daniel Brand, Reinaldo A. Bergamaschi and Leon Stok

IBM Research Division, T.J. Watson Research Center, Yorktown Heights, N.Y., U.S.A.

Abstract

It is commonly expected that any correct implementation can replace its specification inside a larger design without violating the correctness of the whole design. This property (called replaceability) is automatically satisfied in the absence of don't cares because "correctness" by definition implies that specification and implementation compute the identical function. However, don't cares allow an implementation to compute a different function, and thus make it difficult to ensure replaceability. Whether this problem occurs depends on the exact meaning of "don't care" and the associated definition of "correctness". We will consider three meanings of "don't care" and for each give conditions under which correct implementations may replace their specifications.

1. Introduction

The concept of a "don't care" is one of the most frequently discussed in logic synthesis literature with various meanings [1],[3],[5]. In this paper we will talk only about those meanings that imply some choices in the implementation function. A typical example of a don't-care situation, which we will refer to throughout the paper, is described below.

Instruction decoder example: Figure 1 shows a design (CPU) with an instruction decoder (*Instr_decode*), which takes an instruction as input and has two outputs – *ALU_control* and *is_legal*. The latter is a one bit signal saying whether the instruction is legal or not. While the logic inside *Instr_decode* is to be synthesized in isolation, the designer is familiar with the rest of the design (CPU) and therefore wants to tell synthesis that he does not care about the value of *ALU_control* when *is_legal* is false.

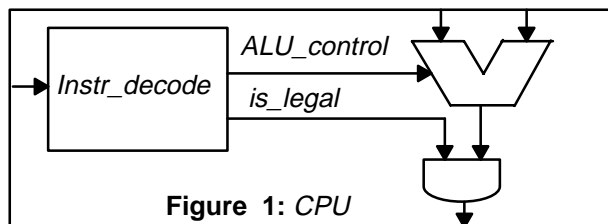


Figure 1: CPU

There are several problems associated with don't cares. The first problem is how to specify them. We will consider three common don't care specification methods in Section 3.

The second problem is defining the meaning of "don't care", that is, any don't care specification method has to define which implementations are considered "correct" (Section 2).

The third problem is synthesis, which has been the subject of most of the literature on don't cares ([1],[2],[3],[10]). This paper does not address the synthesis problem; in fact everything we say is applicable whether synthesis is done automatically or manually. Therefore the word "implementation" will refer only to a function rather than an actual realization. In this paper two implementations will be considered the same if they have the same function irrespective of their area, delay, or other properties.

The fourth problem is implementation verification. Existing verification methods are able to take don't cares into account, but that is not the subject of this paper. Nevertheless, throughout the paper we will assume that synthesis is always performed correctly.

The fifth problem is design verification (usually simulation). Throughout the paper we will assume that a design is divided into several partitions, and there is some don't-care information associated with both the design and each partition. While synthesis is performed one partition at a time, design verification is done on the whole design, and must include verifying the don't-care information. This paper is independent of how design verification is performed, but we will assume that the design's specification has been verified.

The sixth problem, which is the main subject of the paper, involves the formation of the design implementation out of the implementations for the partitions. This paper gives conditions for "replaceability", that is, conditions under which a correct implementation of a partition can replace its specification without violating the correctness of the whole design.

In the presence of don't cares the replaceability question can be asked in two contexts. "Safe replaceability"[8][11] refers to the context of a specific design, where we have to ensure correctness of only that one given design. While that is an important context, it is not the most desirable one. First of all, even in a situation where we are indeed given a specific design, we must be prepared for it to undergo frequent functional changes. It would be very disruptive if any change in one partition required resynthesis of the others just because safe replaceability does not guarantee correctness of the modified design. Second, we want to be able to reuse previously designed partitions in new designs. Safe replaceability would not allow us to do that. Therefore we need a stronger condition – "universal replaceability". If an implementation is a universal replacement for a specification then the replacement will not violate the correctness of any design.

Throughout the paper we will restrict ourselves to combinational logic; that is, memory elements are considered primary inputs and outputs. Due to lack of space, definitions and proofs are omitted and can be found in [4].

2. Formalism

The generally accepted way of specifying the meaning of don't cares is based on some formalism of representing logic (the whole design as well as each partition, specification as well as implementation). Implementations are always represented by total boolean functions. The traditional formalism for representing a specification is a partial function. In our example of the instruction decoder the specification would be given by two partial functions — one for ALU_control and one for is_legal. The don't-care information can be represented by making the ALU_control function undefined for illegal instructions. Given the partial function formalism, an implementation is defined to be correct if it has the same output as the specification whenever the latter is defined.

Partial functions are inadequate to describe some don't-care situations, and therefore Boolean relations were introduced [7]. Boolean relations are also inadequate to describe some don't-care situations [4], and therefore we will not restrict ourselves to either partial functions or Boolean relations. Any don't-care specification method has to define its formalism for representing specifications. Then the method must define correctness of an implementation. Assuming a given notion of correctness, we can now define safe and universal replaceability.

Definition 1: An **embedding** of a function f_U is a pair $[Design, Partition]$, where *Design* is a network with a distinguished node *Partition* having the function f_U .

In the example of Figure 1, let f_U be a function specifying an instruction decoder. Then $[CPU, Instr_decode]$ will be an embedding of f_U if the node *Instr_decode* is assigned f_U .

Definition 2: Let f and f_U be two functions with the same number of inputs and outputs, and let $[Design, Partition]$ be an embedding of f_U . Let F_U be the function computed by *Design* when *Partition* is given the function f_U , while F be the function computed by *Design* when *Partition* has the function f . We say that f is a **safe replacement** for f_U in the embedding $[Design, Partition]$ if F is correct with respect to F_U .

Definition 3: We say that f is a **universal replacement** for f_U iff f is a safe replacement for f_U in any embedding of f_U .

In our example of the instruction decoder assume a specification f_U and let f be one correct implementation. Consider a particular embedding of f_U , as in Figure 1. Suppose that the designer is satisfied with the behavior of f_U in the embedding. Will he be satisfied with the behavior of f in the embedding? If f is a universal replacement for f_U then the answer is "yes", regardless of the embedding. If f is only a safe replacement then the answer depends on the particular embedding.

3. Specifying don't cares

Don't-care situations arise because the designer has some information about the environment of the partition to be synthesized. There are two general strategies for describing that information. The simplest approach is to actually provide synthesis with the entire design (Section 3.1). The other approach is to describe all the correct implementation functions of a partition in isolation from the rest of the design (Section 3.2 and 3.3). The meaning of each don't care specification method is given by defining its notion of correctness. We

then examine under what conditions correctness implies safe or universal replaceability.

3.1. The whole design provided to synthesis

Consider our example of the instruction decoder. The designer provides to synthesis the whole design with one possible implementation f_U of the instruction decoder; for instance f_U might set ALU_control arbitrarily to 0 for illegal instructions. Synthesis is allowed to look at the whole design (Figure 1), and that information is in principle sufficient to perform optimization on the instruction decoder without any other don't-care information from the designer.

In order to define correctness of this don't-care specification method, implementations as well as specifications are represented by total boolean functions. An implementation f is defined to be **correct** with respect to f_U if f can replace f_U without changing the functionality of the whole design. Thus correctness is defined exactly to give safe replaceability.

While this method of specification is both convenient and powerful, it offers only safe replaceability, not universal replaceability. We do not get universal replaceability because the don't care specification is external to the partition being synthesized. Therefore the next two methods make the don't care specification an integral part of f_U .

3.2. Non-Boolean values

In some design languages it is suggested that don't cares be expressed using special non-boolean values, e.g. X. In our instruction decoder example the ALU_control would be assigned X in case of an illegal instruction. In this section we show that this form of don't care specification allows safe and universal replaceability under conditions that are too severe for any existing design language to satisfy.

A simple example of a partition using an X value is the following:

Partition1: if ($a = 1$) then $b = 0$; else $b = X$;

with primary input a and primary output b . When $a = 0$ then the designer does not care what b is.

The meaning of such multi-valued logic is normally defined by the simulator. In this paper we assume a conservative simulator, as is used for instance in VHDL, where simulation is defined for a network, whose nodes have been assigned VHDL operators. Therefore we assume the specification of a partition to be given by a network computing some function $f_U : S^N \rightarrow S^M$. We assume that the set S contains the boolean values 0,1, among others. In the the above example of *Partition1*, $S = \{0,1,X\}$.

To our knowledge there is no general agreement on how to define correctness of an implementation (which is a boolean function) with respect to a multi-valued specification. In this paper we will assume that it is done by the design language providing a rule as to which boolean values may replace non-boolean values appearing on primary outputs. For example, consider the following values $S = \{0, 1, L, H, X\}$; one might specify that L appearing on primary output must be implemented by 0, H must be implemented by a 1 and X may be implemented by either 0 or 1. For notational convenience we will assume that this information is given as a partial order $<$ (pronounced "less specified than") on the set S of values. In our example, the partial order would be $(L < 0, H < 1, X < 0, X < 1)$. Note that an implementation is

not allowed to generate 1 where specification generated 0 (or vice-versa) because in our partial order 0 and 1 are unrelated.

We will write \leq to mean “ $<$ or equal”. The order $<$ can be extended in the usual way to a vector of values component-wise. We will call a value t a **synthesis don’t-care value** provided that $t \leq 0$ and $t \leq 1$. If a function can output a synthesis don’t-care value for at least one input pattern then the function has more than one correct implementation function (Definition 4). But note that a mere use of a synthesis don’t-care value t inside a function definition will not allow multiple correct implementation functions, unless that value t can actually propagate to a partition output.

Definition 4: A function $f: S^N \rightarrow S^M$ is correct with respect to a function $f_U: S^N \rightarrow S^M$ iff for any $\bar{s} \in S^N$, $f_U(\bar{s}) \leq f(\bar{s})$.

For example, with the above partial order there are two correct implementation functions of *Partition1* (in addition to *Partition1* itself)

if $(a = 1)$ then $b = 0$; else $b = 0$;

if $(a = 1)$ then $b = 0$; else $b = 1$;

To illustrate the issue of replaceability in the context of non-boolean values consider

Partition2: { if $(b = 0)$ then $c = 0$; else $c = 1$;
if $(b = 1)$ then $d = 0$; else $d = 1$; }

with primary input b and primary outputs c, d . Since only Boolean values are involved, *Partition2* has only one correct implementation function, namely itself.

Consider a design (Figure 2) consisting of *Partition1* feeding *Partition2*. The design has primary input a and primary outputs c, d . Let us assume that the predicate “=” is extended to X as done in VHDL, in particular, both “ $X=0$ ” and “ $X=1$ ” are false. Then the function F_U computed by the design can be expressed by

if $(a = 1)$ then { $c = 0$; $d = 1$;}
else { $c = 1$; $d = 1$;}
endif

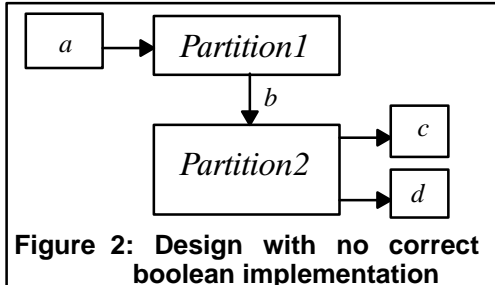


Figure 2: Design with no correct boolean implementation

Suppose that *Partition1* is replaced by its first correct implementation. Then the new design computes the function

if $(a = 1)$ then { $c = 0$; $d = 1$;}
else { $c = 0$; $d = 1$;}
endif

And if *Partition1* is replaced by its second correct implementation then the design computes the function

if $(a = 1)$ then { $c = 0$; $d = 1$;}
else { $c = 1$; $d = 0$;}
endif

We see that by replacing *Partition1* by either of the two correct implementations we get an incorrect design. That means *Partition1* is not safely replaceable by either implementation in the design containing both *Partition1* and *Partition2*. That implies that it is not universally replaceable either. The fact that correctness does not imply replaceability is a problem because today there is no easy way for a designer to know whether his specification is replaceable by any correct implementation. Even if design verification says that the specification is correct, and implementation verification says

that all his partitions are implemented correctly, the combination of those correct implementations may yield an incorrect design.

We now develop necessary and sufficient conditions for universal replaceability. The word *AllF* will refer to the set of all possible functions that can be computed by any network. That means, *AllF* consists of all the primitive (multi-valued) operators of a design language (NAND, COMPARE, etc.) plus all the functions that can be built out of them. For example, if a design language allows only the boolean values 0 and 1 and only the primitive operators AND and OR, then *AllF* consists of positive functions only. If a design language allows the values $\{0, 1, X\}$ and the operators AND, OR, NOT, then *AllF* consists of all the boolean functions plus some three valued functions; exactly which three valued functions would be included depends on how the primitive operators are defined for X .

Definition 5: A function $g: S^P \rightarrow S^Q$ is **monotonic on a set** $T \subseteq S$ iff $(g(\bar{s}_U) \leq g(\bar{s}))$ for any \bar{s}_U, \bar{s} satisfying the following two conditions

a) $\bar{s}_U \leq \bar{s}$

b) $\bar{s}_U \in T^P$ and $\bar{s} \in T^P$

For example, assume that a design language defines $X < 0$, $X < 1$ and provides the operators NAND and COMPARE, where $\text{COMPARE}(0,0) = \text{COMPARE}(1,1) = 1$, and $\text{COMPARE}(0,1) = \text{COMPARE}(1,0) = 0$. Suppose that the two primitive operators are extended to the value X by outputting X whenever any input is X . Then the two primitive operators are monotonic and hence all functions in *AllF* are monotonic. In contrast, suppose that in extending COMPARE from boolean logic to ternary logic we insisted that it must always return a boolean value (as most VHDL packages do). Then it is not possible to make COMPARE monotonic. To see that, consider the question of how to define $\text{COMPARE}(X,0)$. Let $g(s) = \text{COMPARE}(s,0)$. If we define $g(X) = 0$ then we violate monotonicity because $X < 0$ but it is not the case that $g(X) \leq g(0)$. A similar violation occurs if we define $g(X) = 1$.

It is important for functions to be monotonic because as the next two theorems show monotonicity is a necessary and sufficient condition for correctness to imply replaceability.

Theorem 1: Assume that every function in *AllF* is monotonic on a set S , and let $f: S^N \rightarrow S^M, f_U: S^N \rightarrow S^M$ be two functions in *AllF*. Then f is a universal replacement for f_U iff the following are true.

1) f is correct with respect to f_U .

2) If the i -th output of f_U does not depend on the j -th input, then the i -th output of f does not depend on the j -th input either.

Condition 2) is needed to prevent the following situation. A function $f_U: S \rightarrow S$ has input a and output b , and is defined by $b=X$ independently of a . An implementation f is defined by $b = a$. The implementation is correct, but if the partition were embedded in a design where the output b is connected to the input a through an inverter then the design would oscillate.

Theorem 1 gives us a necessary and sufficient condition for universal replaceability under the assumption that *AllF* is monotonic. The monotonicity condition is necessary (Theorem 2) if we want a general assurance that correctness implies universal replaceability.

Theorem 2: Suppose that for every pair of functions $f: S^N \rightarrow S^M$ and $f_U: S^N \rightarrow S^M$ satisfying conditions 1) and

2) of Theorem 1, f is a universal replacement for f_U . Then every function in $AIIF$ is monotonic on the set S .

To achieve universal replaceability synthesis must ensure condition 1) of Theorem 1 as usual. Synthesis must also ensure condition 2), which is normally not a problem. The design language definition must guarantee monotonicity of all operators, which is a problem; we are not aware of any design language that allows non-boolean values and also has monotonic operators. The most common cause of non-monotonic operators are statements like "if (a = b) ...", which existing design languages require to select either the then-clause or the else-clause. As we discussed above, as long as comparison for equality is required to yield a boolean value even for non-boolean inputs, monotonicity cannot be achieved.

Consider what we can do if we want to use existing design languages, but also want to synthesize partitions separately. First note that Theorem 1 requires $AIIF$ to be monotonic only on the set S , which is the set of values propagated between partitions. We would get replaceability if communication between partitions was restricted to values on which all operators are monotonic. Unfortunately existing languages like VHDL are monotonic on the set $\{0, 1\}$ only, which would imply that no partition may output any non-boolean value, which would prevent us from expressing don't care situations using synthesis don't care values.

Theorem 1 is contingent on Definition 4 of correctness. We could get replaceability if we defined correctness differently, for example, as follows. For every input and output port of the specification f_U , a correct implementation would have to have several binary ports. These binary ports would encode the values that can pass the single port of the multi-valued function f_U . This way we would get correctness to imply universal replaceability, but we would not be able to express any don't cares. (Implementations would not be free to replace X with 0 or 1.) Moreover, the expense of encoding the non-boolean values would be prohibitive.

3.3. Assertions

An assertion is a notation allowing a designer to express that certain conditions (states or input combinations) are impossible. For example, the assertion *assert a or b* would cause design verification to fail if it were possible for a and b to be 0 at the same time. In this section we explain that by using assertions we get correctness to imply universal replaceability.

Functions f_U computed by partitions as well as the function F_U computed by the whole design are partial functions, undefined for input patterns that would violate any assertion. For synthesis and verification the assertion gives a don't-care set — all implementations that agree with the specification on the care set are considered **correct**.

Assertions are a special case of the don't-care situations handled in the previous section. The assertion that an expression E is always true is equivalent to assigning a synthesis don't-care value to all primary outputs whenever E is false. Therefore Theorem 1 does apply to assertions; however, the monotonicity precondition is unnecessary.

Thus assertions give us what designers always expected of "correctness", namely that it is equivalent to universal replaceability. However, assertions are strictly weaker than synthesis don't-care values; our initial example of the

instruction decoder cannot be expressed using assertions because all input patterns are possible.

4. Conclusions

It is commonly assumed that once we have a correct implementation of a partition, we can use it as part of any design. We showed that this property (which we called universal replaceability) is not automatically satisfied in the presence of don't cares. We considered three kinds of don't-care specifications, each having its notion of implementation correctness, and we showed under what conditions correctness implies universal replaceability.

When don't cares are expressed by giving the whole design, in addition to the partition being synthesized, then correctness can imply only safe replaceability, not universal replaceability. When don't cares are expressed using non-boolean values then correctness can imply universal replaceability, but only under conditions too stringent for any existing design language to satisfy. That means that in existing design languages if we use non-boolean values to express don't cares and if we synthesize individual partitions of a design separately, then we cannot rely on the whole design to be correct. Thus it appears that simulation of the whole design after synthesis will continue to be necessary. On the other hand, with assertions, which are a more restrictive form of don't cares, we do get correctness to imply universal replaceability.

Acknowledgements

We thank A. Kuehlmann, D. Kung, V. Tiwari, L. Trevillyan for valuable discussions and reading of the manuscript.

5. References

- [1] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R.L. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "Multilevel Logic Minimization Using Implicit Don't Cares", *IEEE TCAD*, Vol. 7., No. 6, pp. 723-740, June 1988.
- [2] R.A. Bergamaschi, D. Brand, L. Stok, "Efficient Use of Large Don't Cares in High-Level and Logic Synthesis", *Proc. ICCAD*, Nov. 1995.
- [3] D. Brand, "Redundancy and Don't Cares in Logic Synthesis", *IEEE TC*, Vol. C-32, No. 10, Oct. 1983, pp. 947-952.
- [4] D. Brand, R. Bergamaschi, L. Stok, "Don't Cares in Synthesis: Theoretical Pitfalls and Practical Solutions", *RC 20127*, IBM T.J. Watson Research Center, September 1995.
- [5] R.K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic publishers, 1984.
- [6] R.K. Brayton, R.L. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "MIS: A Multi-Level Logic Optimization System", *IEEE TCAD*, Vol. 6., No. 6, pp. 1062-1081, November 1987.
- [7] R.K. Brayton and F. Somenzi, "An Exact Minimizer for Boolean Relations", *Proc. ICCAD*, Nov. 1989, pp.316-319.
- [8] B. Lin, G. de Jong, T. Kolks, "Modeling and Optimization of Hierarchical Synchronous Circuits", *Proc. EDTC*, March 1995, pp. 144-149.
- [9] C. Pixley, V. Singhal, A. Aziz, R.K. Brayton, "Multi-level synthesis for safe replaceability", *Proc. of ICCAD*, Nov. 1994, pp. 442-449.
- [10] H. Savoj, R.K. Brayton, "The Use of Observability and External Don't Cares for the Simplification of Multi-level Networks", *Proc. DAC*, June 1990.
- [11] Y. Watanabe, R.K. Brayton, "The Maximum Set of Permissible Behaviors for FSM Networks", *Proc. ICCAD*, Nov. 1993, pp. 316-320.