Don't Care Analysis of Verilog Designs

Rafael Paniago
UFMG
Belo Horizonte, Brazil
rafaelpaniago@dcc.ufmg.br

Maria Fernanda Oliveira Guimarães
UFMG
Belo Horizonte, Brazil
maria.guimaraes@dcc.ufmg.br

Augusto Mafra
Cadence
Belo Horizonte, Brazil
augusto@cadence.com

Mirlaine Crepalde
Cadence
Belo Horizonte, Brazil
mirlaine@cadence.com

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

Abstract

This paper presents a static Don't Care Analysis for Verilog designs. The analysis identifies bits within signals that do not influence any observable output or persistent state of a hardware design. Our approach operates on a static single-assignment (SSA) representation of hardware programs and propagates bit-level observability information through a backward may data-flow analysis. Each signal is associated with a care-mask that captures which bits may affect observable behavior, and these masks are computed through a set of compositional transfer functions defined over a finite lattice. We formalize the analysis on top of a small Verilog-like intermediate language, prove termination through monotonicity and finite-height lattice arguments, and establish soundness with respect to the operational semantics of the language. We implemented the analysis in the CIRCT compiler infrastructure targeting the `hw` and `comb` dialects. An evaluation on 21,361 Verilog designs from the ChiBench benchmark suite shows that the analysis scales linearly in practice and incurs low compilation overhead. Among the analyzed designs, 2,056 contain detectable don't care bits, with affected designs exhibiting a median of 32.84% semantically irrelevant bits. Finally, we demonstrate that materializing the inferred masks can simplify datapaths and reduce the runtime of formal verification tools.

Keywords

Don't Care Analysis of Verilog Designs

1 Introduction

Verilog is a widely used hardware description language (HDL) for modeling and implementing digital systems [9]. It plays a central role in modern hardware design flows, being the input language of numerous electronic design automation (EDA) tools used for synthesis, simulation, and formal verification. As with traditional programming languages, these tools rely on static analyses to reason about program behavior, enabling optimizations and correctness guarantees [7]. However, the scale and complexity of contemporary hardware designs make the development of efficient and precise analyses increasingly important [4].

As recently mentioned by Chen et al. [3], analyzing Verilog designs poses challenges that differ from those found in conventional software. In particular, Verilog operates at the level of bit-precise computations, where the semantics of a program depends not only on values but also on how individual bits propagate through circuits [10]. A key observation is that not all bits in a design necessarily contribute to observable behavior: some bits may be computed but never influence outputs or state elements. Despite this, existing open-source infrastructures lack formalized analyses capable of identifying such *don't care* bits in a principled and scalable way.

Contributions. This paper presents a static *Don't Care Analysis* for Verilog designs. This analysis operates on a static single-assignment (SSA) [6] representation of hardware programs and propagates bit-level observability information in a *backward*, *may* data-flow framework. It assigns to each signal a mask indicating which bits may influence observable outputs, and computes these masks through a set of compositional transfer functions. Section 3 formalizes the analysis over a finite lattice, proves that its transfer functions are monotonic, and shows that the analysis converges to a fixed point. Furthermore, it establishes that any bit identified as *don't care* does not affect the observable behavior of the design.

Summary of Results. We have implemented our analysis in the CIRCT compiler infrastructure, a hardware-oriented framework built on top of MLIR. Our implementation targets the `hw` and `comb` dialects, enabling integration with existing compilation and verification flows. We evaluate the analysis on a corpus of 21,361 Verilog designs from the ChiBench benchmark suite, of which 2,056 contain detectable don't care bits. The analysis is efficient, accounting for only a small fraction of total compilation time, and scales linearly in practice.

2 Overview

Hardware designs frequently manipulate signals whose full bit-width is never semantically required. Although a circuit may compute a wide intermediate value, only a subset of its bits may ultimately influence observable behavior, such as top-level outputs or state-holding elements. This situation arises naturally in several contexts, including truncation, masking, partial extraction, and cross-module communication. In these cases, some bits are computed and propagated through the circuit even though their values never affect the externally visible behavior of the design. Identifying

such redundant computations is important because they unnecessarily increase datapath width, circuit complexity, and the cost of downstream analyses such as formal verification.

2.1 Don't Care Bits

In this work, we refer to semantically irrelevant bits as *don't care bits*. Intuitively, a don't care bit is a bit whose value can be arbitrarily changed without altering the observable behavior of the hardware design. Definition 2.1 formalizes this intuition. To make this definition more intuitive, Example 1 illustrates how don't care bits naturally emerge across module boundaries in Verilog designs.

Definition 2.1 (Don't care bit). A *don't care bit* is a specific bit within a signal, register, or wire whose value does not affect any observable output or persistent state of the hardware design.

Example 1. *Consider the Verilog design shown in Figure 1. The top-level module* `top_cross_boundary` *receives three 32-bit inputs and instantiates the module* `generic_mult_add`*, which computes a 32-bit result. However, the top-level assignment* `assign out = full_result[15:0];` *retains only the lower 16 bits of this value. Consequently, bits 16 through 31 of the wire* `full_result` *are don't care bits: although they are produced by the instantiated module, they never influence the observable output of the design.*

```
01 module generic_mult_add (
02   input  [31:0] a, b, c,
03   output [31:0] result
04 );
05   wire [31:0] mult = a * b;
06   assign result = mult + c;
07 endmodule

08 module top_cross_boundary (
09   input  [31:0] in1, in2, in3,
10   output [15:0] out
11 );
12   wire [31:0] full_result;
13   generic_mult_add my_hw_block (
14     .a(in1),
15     .b(in2),
16     .c(in3),
17     .result(full_result)
18   );
19   assign out = full_result[15:0];
20 endmodule
```

Figure 1: Example of don't care bits emerging across module boundaries.

2.2 Backward Propagation of Don't Care Bits

To identify don't care bits, Section 3 will formalize a backward data-flow analysis based on *demand masks*. The analysis starts from observable sinks of the circuit, such as top-level outputs and stateful elements, assigning them masks in which a bit value of '1' denotes a demanded bit and '0' denotes a don't care bit. The analysis then propagates these masks backwards through the intermediate representation, abstractly evaluating each operation and inferring which input bits are required to compute the demanded outputs. Rather than propagating concrete bit values, the analysis propagates bit-level observability information. This propagation occurs through arithmetic operations, logical gates, slicing and concatenation operations, and also across module instantiation boundaries, as illustrated in Example 2.

Example 2. *Applying the analysis to the design in Figure 1, the truncation at the output induces the demand mask* `0x0000FFFF` *for the signal* `full_result`*. When this mask is propagated backwards across the* `my_hw_block` *instantiation boundary, the analysis infers that only the lower 16 bits of the port* `result` *are observable. This information then propagates into the body of the* `generic_mult_add` *module, revealing that the upper 16 bits produced by the internal addition and multiplication operations are also don't care bits.*

The identification of don't care bits enables several optimization opportunities. These bits can be exploited to simplify arithmetic datapaths, remove dead logic, normalize structurally distinct computations, and reduce the complexity of formal verification tasks. In particular, replacing don't care bits with deterministic constants (such as zeros) can increase the structural similarity between hardware subgraphs, enabling more effective redundancy elimination and canonicalization.

Example 3. *After identifying don't care bits, we apply a transformation pass that materializes the inferred demand masks using bitwise* and *operations. This pass explicitly clears irrelevant bits in the intermediate representation. As illustrated in Figure 2, subsequent compiler optimizations can then narrow the arithmetic operations from 32 bits to 16 bits. In this example, the original 32-bit multiplier is effectively reduced to a 16-bit multiplier, yielding a smaller datapath while preserving the observable behavior of the design.*

```
01 module generic_mult_add(
02   input  [31:0] a, b, c,
03   output [31:0] result
04 );
05   assign result = {16'h0, a[15:0] * b[15:0] + c[15:0]};
06 endmodule
07
08 module top_cross_boundary(
09   input  [31:0] in1, in2, in3,
10   output [15:0] out
11 );
12   wire [31:0] _my_hw_block_result;
13   generic_mult_add my_hw_block (
14     .a      (in1),
15     .b      (in2),
16     .c      ({16'h0, in3[15:0]}),
17     .result (_my_hw_block_result)
18   );
19   assign out = _my_hw_block_result[15:0];
20 endmodule
```

Figure 2: Optimized design after applying don't care masks.

3 Formalization of the Analysis

The don't care analysis is a backward may data-flow analysis that operates on Verilog designs. Its goal is to determine which bits of each signal may influence observable behavior, such as primary outputs or state-holding elements. The analysis is implemented over the CIRCT `hw` and `comb` dialects. It associates each SSA value with an abstract element drawn from a bitwise lattice, described in Section 3.2. These abstract values are propagated through the program by iteratively applying transfer functions, as described in Section 3.3, until a fixed point is reached.

3.1 The μ-CIRCT Toy Language

We formalize the don't care analysis on top of μ-CIRCT, a toy Verilog-like language that captures the essential bit-level structure of hardware descriptions while abstracting away syntactic and semantic details irrelevant to our development. The design of μ-CIRCT follows the philosophy of distilling hardware programs into a small set of compositional, side-effect-free operations over fixed-width bit-vectors, expressed in static single-assignment (SSA) form [6]. This choice allows us to model data dependencies explicitly and reason locally about how individual bits propagate through computations. By focusing on a core fragment that includes slicing, concatenation, bitwise operations, multiplexing, and arithmetic, we obtain a language that is simple enough to admit a precise formal treatment while remaining expressive enough to reflect the behavior of the CIRCT `hw` and `comb` dialects targeted by our implementation.

Syntax. Figure 3 defines the syntax of μ-CIRCT, a minimal intermediate language inspired by the `hw` and `comb` dialects of CIRCT. The language preserves the bit-precise, SSA-based nature of these dialects while abstracting away features that are not directly relevant to our analysis, such as attributes, types, and structural declarations. μ-CIRCT focuses exclusively on the core combinational operations necessary to formalize the propagation of bit-level observability information. A μ-CIRCT program is a linear sequence of instructions. This representation reflects the acyclic dataflow structure of combinational hardware and simplifies the presentation of both the operational and abstract semantics.

P	::=	$I;\ P \mid \epsilon$	Program
I	::=	$x \leftarrow \texttt{extract}(y, \ell, w)$	Bit extraction (slice)
	|	$x \leftarrow \texttt{concat}(y_1, y_2)$	Concatenation
	|	$x \leftarrow \texttt{and}(y_1, y_2)$	Bitwise AND
	|	$x \leftarrow \texttt{or}(y_1, y_2)$	Bitwise OR
	|	$x \leftarrow \texttt{mux}(c, y_1, y_2)$	Multiplexer
	|	$x \leftarrow \texttt{add}(y_1, y_2)$	Arithmetic addition

Figure 3: Syntax of μ-CIRCT.

Operational Semantics. We define a small-step operational semantics for μ-CIRCT programs. As shown in Figure 3, a program is a finite sequence of instructions: $P ::= I_1; I_2; \ldots; I_n$. A *configuration* is a pair (σ, P), where σ is an environment mapping variables to bit-vectors, and P is the remaining sequence of instructions to execute. The small-step relation

$$(\sigma, P) \to (\sigma', P')$$

describes the execution of a single instruction. We write ϵ for the empty program. Figure 4 defines the transition rules. Each rule consumes the first instruction in the sequence, updates the environment, and proceeds with the remaining program.

The execution of a program is given by the reflexive–transitive closure of the step relation $\to^*$. Formally:

$$\frac{}{(\sigma, P) \to^* (\sigma, P)} \quad (\textsc{Refl})$$

$$(\textsc{Extract}) \quad \frac{\sigma(y)=v \quad v'=v[\ell+w-1:\ell]}{(\sigma,\ x\leftarrow\texttt{extract}(y,\ell,w);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

$$(\textsc{Concat}) \quad \frac{\sigma(y_1)=v_1 \quad \sigma(y_2)=v_2 \quad v'=v_1::v_2}{(\sigma,\ x\leftarrow\texttt{concat}(y_1,y_2);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

$$(\textsc{And}) \quad \frac{\sigma(y_1)=v_1 \quad \sigma(y_2)=v_2 \quad v'[i]=v_1[i]\wedge v_2[i]}{(\sigma,\ x\leftarrow\texttt{and}(y_1,y_2);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

$$(\textsc{Or}) \quad \frac{\sigma(y_1)=v_1 \quad \sigma(y_2)=v_2 \quad v'[i]=v_1[i]\vee v_2[i]}{(\sigma,\ x\leftarrow\texttt{or}(y_1,y_2);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

$$(\textsc{Mux}) \quad \frac{\sigma(c)=b \quad \sigma(y_1)=v_1 \quad \sigma(y_2)=v_2 \quad v'=\begin{cases} v_1 & \text{if } b=1 \\ v_2 & \text{if } b=0 \end{cases}}{(\sigma,\ x\leftarrow\texttt{mux}(c,y_1,y_2);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

$$(\textsc{Add}) \quad \frac{\sigma(y_1)=v_1 \quad \sigma(y_2)=v_2 \quad v'=v_1+v_2 \bmod 2^{W(x)}}{(\sigma,\ x\leftarrow\texttt{add}(y_1,y_2);\ P)\to(\sigma[x\mapsto v'],\ P)}$$

Figure 4: Small-step operational semantics of μ-CIRCT.

$$\frac{(\sigma, P) \to (\sigma', P') \quad (\sigma', P') \to^* (\sigma'', P'')}{(\sigma, P) \to^* (\sigma'', P'')} \quad (\textsc{Trans})$$

A program P evaluates from an initial environment σ_0 to a final environment σ_f if:

$$(\sigma_0, P) \to^* (\sigma_f, \epsilon)$$

This semantics models μ-CIRCT as a deterministic sequential execution of combinational instructions, making explicit how each operation updates the environment and enabling a precise connection with the backward data-flow analysis defined in the next section.

3.2 Lattice and Abstract Domain

Let v be a signal of bit-width W. The abstract value associated with v is a bit-vector (mask) $M \in \{0,1\}^W$, called a *care-mask*. Each bit in M indicates whether the corresponding bit in v is observable:

- $M[i] = 1$ (care): the i-th bit of v may influence some observable output.
- $M[i] = 0$ (don't care): the i-th bit of v has no impact on observable behavior.

The abstract domain associated with a signal of width W is the set of all bit-masks of width W, namely $\{0,1\}^W$. Each abstract element represents the observability status of the corresponding bits in a signal: a bit value of 1 denotes a *care* bit, whereas a value of 0 denotes a *don't care* bit. We order these masks pointwise:

$$M_1 \sqsubseteq M_2 \iff \forall i \in [0, W-1],\ M_1[i] \le M_2[i].$$

Intuitively, $M_1 \sqsubseteq M_2$ means that every bit demanded by M_1 is also demanded by M_2. Thus, moving upward in the lattice corresponds to increasing the amount of observable information associated with a signal.

This domain forms a finite distributive lattice with:

- **Top** ($\top$): the mask $11\ldots1$, indicating that every bit of the signal is observable.
- **Bottom** ($\bot$): the mask $00\ldots0$, indicating that no bit of the signal contributes to observable behavior.
- **Join** ($\sqcup$): bitwise OR, combining observability requirements from different uses of a signal.

- **Meet** ($\sqcap$): bitwise AND, representing the intersection of observability requirements.

Because the analysis is backward, joins arise naturally from the fan-out structure of the dataflow graph. A signal may be consumed by multiple operations, each imposing different observability requirements on its bits. The analysis combines these requirements using the join operator: if any downstream use requires a bit, then that bit must be marked as *care*. Consequently, the join operation corresponds to the least upper bound in the lattice.

Example 4. *Figure 5 shows the Hasse diagram of the lattice associated with four-bit signals. Each node represents a distinct observability mask. For example, the lattice element* `1100` *indicates that the two most significant bits are observable, whereas the two least significant bits are don't care. Such a mask naturally arises in designs that use only the upper portion of a signal. For instance, consider the assignment:*

```
assign out = x[3:2];
```

In this case, only bits 3 and 2 of the signal `x` *influence the observable output, yielding the demand mask* `1100`*. Conversely, the mask* `0011` *would arise if only the lower two bits of* `x` *were observable.*

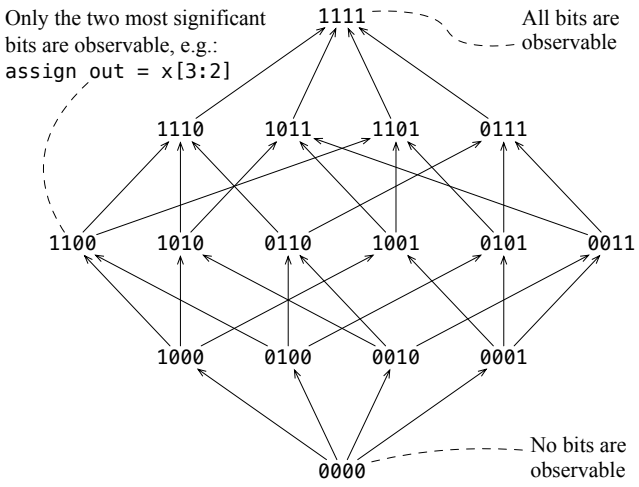

Figure 5: Lattice of don't care masks for four-bit signals.

3.3 Transfer Functions

We define the abstract semantics of μ-CIRCT as a small-step relation over abstract environments. An abstract environment Σ maps each variable to a care-mask in $\{0,1\}^{W(x)}$. Therefore, the judgment

$$(\Sigma, P) \rightsquigarrow (\Sigma', P')$$

denotes one step of the abstract interpreter. The rules in Figure 6 show how care-masks are propagated from outputs to inputs.

Initialization. The analysis begins by assigning the abstract value $\bot$ (that is, a mask in which every bit is set to "0") to every variable in the intermediate representation of the target design. This initialization represents the most conservative assumption possible:

$$\text{(Extract)}\quad \frac{\Sigma(x)=M \quad M'=M\ll \ell}{(\Sigma,\ x\leftarrow \texttt{extract}(y,\ell,w);\ P)\rightsquigarrow(\Sigma[\,y\mapsto \Sigma(y)\vee M'\,],\ P)}$$

$$\text{(Concat)}\quad \frac{\Sigma(x)=M \quad M_1=M[W(x)-1{:}k] \quad M_2=M[k-1{:}0]}{(\Sigma,\ x\leftarrow \texttt{concat}(y_1,y_2);\ P)\rightsquigarrow(\Sigma[\,y_1\mapsto \Sigma(y_1)\vee M_1,\ y_2\mapsto \Sigma(y_2)\vee M_2\,],\ P)}$$

$$\text{(And)}\quad \frac{\Sigma(x)=M}{(\Sigma,\ x\leftarrow \texttt{and}(y_1,y_2);\ P)\rightsquigarrow(\Sigma[\,y_1\mapsto \Sigma(y_1)\vee M,\ y_2\mapsto \Sigma(y_2)\vee M\,],\ P)}$$

$$\text{(Or)}\quad \frac{\Sigma(x)=M}{(\Sigma,\ x\leftarrow \texttt{or}(y_1,y_2);\ P)\rightsquigarrow(\Sigma[\,y_1\mapsto \Sigma(y_1)\vee M,\ y_2\mapsto \Sigma(y_2)\vee M\,],\ P)}$$

$$\text{(Mux)}\quad \frac{\Sigma(x)=M}{(\Sigma,\ x\leftarrow \texttt{mux}(c,y_1,y_2);\ P)\rightsquigarrow(\Sigma[\,c\mapsto \top,\ y_1\mapsto \Sigma(y_1)\vee M,\ y_2\mapsto \Sigma(y_2)\vee M\,],\ P)}$$

$$\text{(Add)}\quad \frac{\Sigma(x)=M \quad i=\max\{j\mid M[j]=1\}}{(\Sigma,\ x\leftarrow \texttt{add}(y_1,y_2);\ P)\rightsquigarrow(\Sigma[\,y_1\mapsto \Sigma(y_1)\vee \texttt{prefix}(i),\ y_2\mapsto \Sigma(y_2)\vee \texttt{prefix}(i)\,],\ P)}$$

Figure 6: Abstract small-step semantics (transfer functions) of μ-CIRCT.

before the analysis propagates any information, every bit is considered a don't care and therefore irrelevant to the computation. Every *output* bit is considered relevant and is therefore initialized with $\top$. These initial non-$\bot$ masks serve as seeds for the backward propagation process, enabling the analysis to infer relevant bits throughout the remainder of the circuit. Example 5 illustrates this initialization process.

Example 5. *Figure 7 shows a simplified fragment of the intermediate representation derived from the design previously introduced in Figure 2. Initially, every variable is assigned the abstract value $\bot$, indicating that all bits are assumed non-observable. At this stage, the analysis has not yet inferred any relevant bits. The only exception is the output assignment, which extracts the lower 16 bits of the signal* `full_result`*. From this operation, the analysis immediately infers the mask* `0xFFFF`*, indicating that the lower 16 bits are observable. This mask becomes the starting point for the backward propagation of observability information across the remaining operations in the design.*

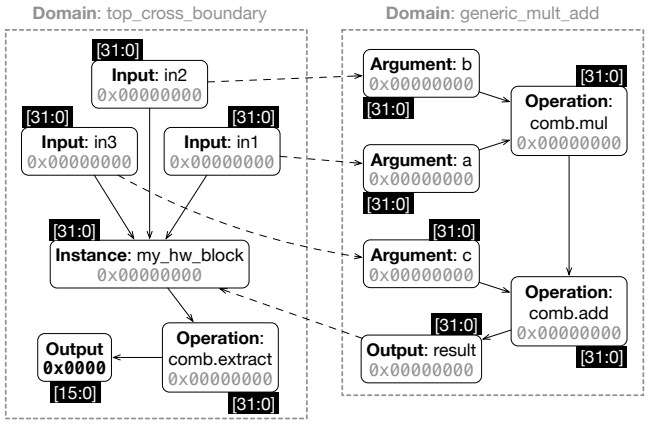

Figure 7: Initial observability masks assigned to the variables of the running example.

Backward Propagation. After initialization, the analysis iteratively propagates observability constraints backward through the dataflow graph until reaching a fixed point. Propagation occurs

through the iterative re-evaluation of the transfer functions in Figure 6. Every bit that can influence some output bit is marked as observable. Example 6 illustrates this process.

Example 6. *Figure 8 illustrates the backward propagation of demand masks in the running example introduced in Section 2. The analysis starts at the observable output of the module* `top_cross_boundary` *(Step 1), which is initialized with the mask* `0xFFFF`*. This demand propagates backward through the* `comb.extract` *operation (Step 2). Because the extraction selects only the lower 16 bits of the signal* `full_result`*, the transfer function in Figure 6 associates the result of the extraction with the mask* `0x0000FFFF`*. The same mask is then propagated to the output port* `result` *of the instantiated module* `my_hw_block` *(Step 3), establishing the connection between the observable output of the parent module and the internal computations of the instantiated module.*

Interprocedural Propagation. The analysis propagates observability information not only within individual modules but also across module instantiation boundaries. For module instances represented as `hw.instance` operations, the analysis matches actual arguments in the caller module with the corresponding formal parameters in the callee module. Demand masks inferred for output ports are therefore propagated into the instantiated module, allowing the analysis to identify don't care bits across hierarchical boundaries, as Example 7 illustrates.

Example 7. *Continuing from Example 6, the mask associated with the output port* `result` *of* `my_hw_block` *(Step 3 in Fig. 8) is propagated across the instantiation boundary into the module* `generic_mult_add`*. Inside the callee module, this demand first reaches the* `comb.add` *operation (Step 4), whose transfer function propagates the mask to both operands: the result of* `comb.mul` *(Step 5) and the argument* `c` *(Step 6). The demand propagated to* `comb.mul` *is then further propagated to its operands* `a` *and* `b` *(Steps 7 and 8). Eventually, the analysis reaches the primary inputs* `in1`*,* `in2`*, and* `in3`*, inferring that only their lower 16 bits contribute to observable behavior. Because the design is acyclic, observability information propagates monotonically through the dataflow graph, and the analysis reaches a fixed point after a single backward traversal.*

Interprocedural propagation is necessary because many hardware designs are modularized. A truncation or masking operation performed in a parent module may imply that large portions of the computations performed inside a submodule are semantically irrelevant. When the definition of an instantiated module is unavailable, the analysis falls back to a conservative approximation by assigning $\top$ to all output ports of the instance. In this case, every output bit is assumed observable, ensuring that the analysis remains sound in the presence of incomplete designs or external IP blocks.

Termination. The backward propagation process is guaranteed to terminate (Theorem 4.2) because the transfer functions defined in Figure 6 are monotonic and operate over the finite-height lattice introduced in Section 3.2. Furthermore, for purely combinational acyclic designs, propagation stabilizes after a single backward traversal of the dataflow graph. In such cases, every operation is visited at most once after the observability requirements of its users become available. Example 6 illustrates this propagation process.

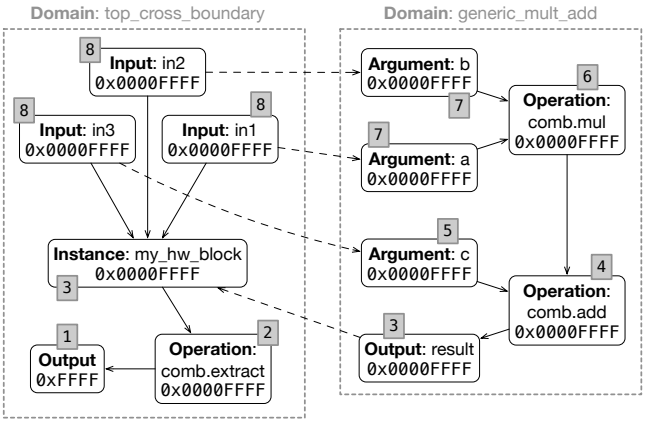

Figure 8: Backward propagation of demand masks across arithmetic operations and module instantiation boundaries.

4 Correctness

This section establishes two results. First, Section 4.1 shows that the Don't Care Analysis terminates. Second, Section 4.2 shows that it provides a conservative over-approximation of don't care bits.

4.1 Termination

Termination of the Don't Care Analysis follows from two observations. First, every transfer function defined in Figure 6 is monotonic with respect to the lattice order $\sqsubseteq$ (Lemma 4.1). Second, the lattice introduced in Section 3.2, which serves as both domain and codomain of each transfer function, has finite height. Together, these two properties guarantee convergence to a fixed point by the Knaster–Tarski theorem.

Lemma 4.1 (Monotonicity of Transfer Functions). *For every transfer function f of the Don't Care Analysis and every pair of care-masks $M_1 \sqsubseteq M_2$, we have $f(M_1) \sqsubseteq f(M_2)$.*

Proof: The transfer functions in Figure 6 are built from the following primitive operations on bit-vectors, each of which is monotonic with respect to the pointwise order $\sqsubseteq$:

- **Bitwise OR and AND.** For any $M_1 \sqsubseteq M_2$ and mask M, we have $M_1 \vee M \sqsubseteq M_2 \vee M$ and $M_1 \wedge M \sqsubseteq M_2 \wedge M$, since both operations act independently on each bit.
- **Left shift.** For any $M_1 \sqsubseteq M_2$ and shift amount $\ell \geq 0$, we have $M_1 \ll \ell \sqsubseteq M_2 \ll \ell$, since shifting inserts zeros at the low end and preserves the relative order of the remaining bits.
- **Bit slicing and concatenation.** Projecting a mask onto a contiguous range of bit positions, or concatenating two masks, both preserve $\sqsubseteq$ component-wise, as neither operation combines bits from distinct positions.
- **Prefix propagation.** The function $\texttt{prefix}(i)$, used in the transfer function for addition, returns the mask $1^{i+1}0^{W-i-1}$ (i.e., ones in all positions 0 through i). Since $\texttt{prefix}(i)$ depends only on the index of the highest set bit of $\Sigma(x)$, and this index is non-decreasing as $\Sigma(x)$ grows under $\sqsubseteq$, prefix propagation is monotonic.

Each transfer function is a composition of the above operations joined by bitwise OR (to accumulate contributions from multiple uses). Since each primitive is monotonic and

composition preserves monotonicity, every transfer function is monotonic. □

Because the lattice height is bounded by the bit-width of each signal and every transfer function is monotonic, the analysis is guaranteed to converge to a fixed point. In practice, as shown in Section 5.1, this convergence is fast: the analysis runs in time approximately linear in the size of the input design. Theorem 4.2 formalizes the convergence guarantee.

Theorem 4.2 (Convergence to a Fixed Point). *Let Σ_0 be the environment that maps every SSA value v of width $W(v)$ to $\bot = 0^{W(v)}$. The sequence $(\Sigma_k)_{k\geq 0}$ obtained by iteratively applying the global transfer function of the Don't Care Analysis to Σ_0 is an ascending chain that stabilizes at a least fixed point Σ^*.*

Proof: The set of environments $\mathcal{E} = \prod_v \{0,1\}^{W(v)}$, where the product ranges over all SSA values v in the program, forms a finite lattice under the pointwise extension of $\sqsubseteq$. Its height is bounded by $\sum_v W(v)$, the total bit-width of all values, which is finite.
Define the global transfer function $F\colon \mathcal{E} \to \mathcal{E}$ as the simultaneous application of all per-instruction transfer functions. By Lemma 4.1, each per-instruction function is monotonic; hence F is monotonic as well.
Starting from $\Sigma_0 = \bot_{\mathcal{E}}$ (the least element), the sequence $\Sigma_0 \sqsubseteq F(\Sigma_0) \sqsubseteq F^2(\Sigma_0) \sqsubseteq \cdots$ is an ascending chain in $\mathcal{E}$. Since $\mathcal{E}$ has finite height, this chain must eventually stabilize: there exists k^* such that $F(\Sigma_{k^*}) = \Sigma_{k^*}$. Setting $\Sigma^* = \Sigma_{k^*}$ yields the desired least fixed point. □

4.2 Soundness

This section establishes that the Don't Care Analysis is *sound*: any bit identified as don't care truly does not affect the observable behavior of the design. We first define what it means for a bit to be observable (Definition 4.3), then argue that termination of μ-CIRCT programs is guaranteed by construction, and finally state and prove the main soundness theorem (Theorem 4.4).

Definition 4.3 (Observable outputs). Let P be a μ-CIRCT program. A variable v is an *observable output* of P if it is either (i) a primary output of the module, or (ii) an operand of a stateful operation (e.g., a register input). Given a final environment σ_f, the *observable behavior* of P under an initial environment σ_0 is the restriction of σ_f to the set of observable outputs.

Termination. A key prerequisite for the soundness theorem is that programs actually produce a well-defined final environment. In μ-CIRCT, this is not an assumption but a *consequence of the language design*: every program is a finite, straight-line sequence of combinational instructions $I_1; I_2; \ldots; I_n$ with no loops or branching (Figure 3). Each instruction steps deterministically to the next, and the small-step relation of Figure 4 therefore always reaches the empty program ϵ in exactly n steps. Hence, for every program P and every initial environment σ_0, there exists a unique final environment σ_f such that $(\sigma_0, P) \to^* (\sigma_f, \epsilon)$. The soundness theorem below can therefore be stated without a separate termination hypothesis.

Theorem 4.4 (Soundness of the Don't Care Analysis). *Let $P = I_1; \ldots; I_n$ be a μ-CIRCT program and let Σ^* be the fixed-point environment computed by the Don't Care Analysis. For any initial environment σ_0, let σ_f be the unique final environment such that $(\sigma_0, P) \to^* (\sigma_f, \epsilon)$. For any SSA value v and bit index i, if $\Sigma^*(v)[i] = 0$, then for every initial environment σ_0' that differs from σ_0 only in bit i of v, the observable behavior of P under σ_0 and σ_0' is identical.*

Proof: The proof proceeds by backward induction on the position of the instruction that defines v in the sequence $I_1; \ldots; I_n$.

Base case. If v is an observable output, then the analysis seeds $\Sigma(v) = \top$, so $\Sigma^*(v)[i] = 1$ for all i. Hence, the hypothesis $\Sigma^*(v)[i] = 0$ is vacuously false, and there is nothing to prove.

Inductive step. Suppose v is defined by instruction I_k and assume the theorem holds for all variables defined by instructions $I_{k+1}, \ldots, I_n$. We case-split on the form of I_k.

- **Extract.** $I_k \equiv x \leftarrow \texttt{extract}(y, \ell, w)$. The transfer function sets $\Sigma(y)$ += $\Sigma(x) \ll \ell$. If $\Sigma^*(x)[j] = 0$ for some j, then $\Sigma^*(y)[\ell + j] = 0$. By the induction hypothesis applied to x, flipping bit j of x does not affect any observable output. Since the concrete semantics sets $x \leftarrow y[\ell + w - 1 : \ell]$, flipping bit $\ell + j$ of y flips exactly bit j of x, so it too has no effect on observables.
- **Concat.** $I_k \equiv x \leftarrow \texttt{concat}(y_1, y_2)$. The transfer function splits $\Sigma(x)$ into a high part M_1 for y_1 and a low part M_2 for y_2. If $\Sigma^*(y_j)[i] = 0$, then the corresponding bit of $\Sigma^*(x)$ is also 0, and the induction hypothesis on x gives the result.
- **And / Or.** Both transfer functions propagate $\Sigma(x)$ unchanged to each operand. If $\Sigma^*(y_j)[i] = 0$, then $\Sigma^*(x)[i] = 0$, and by induction, flipping bit i of x is unobservable; since flipping bit i of y_j flips at most bit i of x, the result follows.
- **Mux.** The transfer function assigns $\top$ to the selector c, so $\Sigma^*(c)[0] = 1$ always and the hypothesis $\Sigma^*(c)[0] = 0$ is vacuously false. For the data inputs y_1 and y_2, the argument is identical to the And/Or case.
- **Add.** Let $i = \max\{j \mid \Sigma^*(x)[j] = 1\}$. The transfer function propagates $\texttt{prefix}(i)$ to both y_1 and y_2, marking all bits up to position i as care. If $\Sigma^*(y_j)[k] = 0$ for some k, then $k > i$, meaning that bit k lies strictly above the highest care bit of x. Because carry propagation in binary addition can only affect bits at positions $\leq k$ when flipping bit k of an addend, and all bits of x above position i are don't care by hypothesis, flipping bit k of y_j cannot affect any observable output.

In all cases, if $\Sigma^*(v)[i] = 0$, then flipping bit i of v propagates only through bits already marked as don't care, and therefore cannot reach any observable output. □ □

5 Evaluation

This section evaluates the following research questions:

RQ1: What is the asymptotic running-time behavior of the proposed Don't Care Analysis?
RQ2: What is the prevalence of don't care bits in real-world benchmarks?

Before presenting the empirical results that answer these questions, we describe the experimental setup used throughout the evaluation.
Hardware: All experiments were conducted on an AMD Ryzen 9 5900X processor with 32 GB of RAM, two L1 caches (384 KB each),

one L2 cache (6 MB), and one L3 cache (64 MB), running Ubuntu Linux 22.04.5 LTS.

Software: The Don't Care Analysis was implemented on top of CIRCT Release 192 (Oct-25).

Benchmarks: The experiments use the ChiBench collection of Verilog designs [12]. ChiBench contains 49,599 designs mined from open-source repositories distributed under permissive licenses. Among these designs, CIRCT successfully parses 21,361. Out of this set, 2,056 designs contain don't care bits, as discussed in Section 5.2.

Correctness: To validate the correctness of the proposed analysis, we masked all bits classified as don't care using logical AND operations. We then used Jasper Sequential Equivalence Checking (SEC) to verify the semantic equivalence between the original designs and the transformed designs with masked bits. Semantic equivalence was successfully established for all 2,056 designs that contained don't care bits.

5.1 RQ1: Asymptotic Behavior

Our implementation of the Don't Care Analysis appears to run in time linearly proportional to the size of the Verilog designs. This section supports this claim with empirical evidence. When referring to the *analysis time*, we consider only the execution time of the pass implementing the proposed analysis, including the iterations required to reach a fixed point. The time required for parsing, lowering, and printing results is not included.

Discussion: Figure 9 shows the analysis time as a function of the number of lines of code in each Verilog design. The linear correlation is very strong ($R^2 = 0.99$), indicating that the analysis scales linearly in practice. Furthermore, the analysis is generally very fast. It processes the vast majority of designs (99.1%) in less than 0.1 seconds. Our largest benchmark, containing approximately 384K lines of code, is analyzed in about 0.06 seconds.

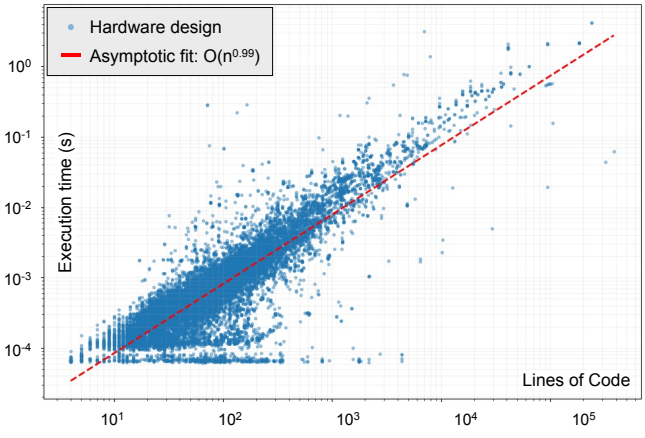

Figure 9: Empirical asymptotic behavior: number of lines of code per Verilog design versus the execution time of the Don't Care Analysis.

This efficiency becomes even more apparent when compared with other phases of the compilation pipeline. Figure 10 provides this perspective by comparing the runtime distributions of three CIRCT phases: parsing, canonicalization, and common subexpression elimination. Overall, the Don't Care Analysis accounts for approximately 7.103% of the total compilation time of the CIRCT toolchain. In absolute terms, it takes 148.87 seconds to analyze all 21, 360 designs. The tight distribution and low median indicate that the proposed pass introduces only minimal overhead for the vast majority of hardware designs.

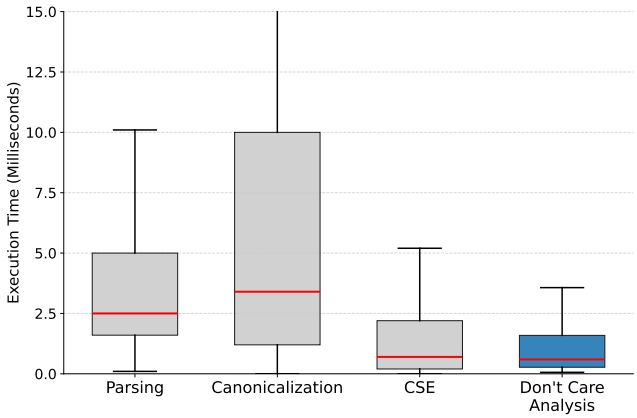

Figure 10: Comparison of different phases of the CIRCT pipeline, highlighting the relative cost of the Don't Care Analysis.

5.2 RQ2: Prevalence

The analysis presented in this paper is able to identify a non-trivial number of don't care bits in the evaluated benchmarks; however, the number of affected designs remains relatively small compared with the total number of analyzed designs. As previously discussed, among the 21,361 analyzed designs, 2,056 contain confirmed don't care bits. This result is expected, as don't care bits represent wasted hardware resources, which designers generally try to avoid when writing Verilog specifications. This section investigates the prevalence and structural origins of these bits.

Discussion. We evaluate both the relative proportion of don't care bits and the operations that most frequently contribute to their emergence. On average, 38.8% of the bits computed in the affected designs are irrelevant. The median proportion is 32.84%, meaning that in half of these designs nearly one third of the computed bits are unobserved. The interquartile range (IQR) further shows that the middle 50% of these designs exhibit waste proportions between 11.11% and 64.64%, reaching as high as 99.98% in extreme cases.

Figure 11 shows that the proportion of don't care bits tends to decrease as design size increases. This behavior is expected because larger hardware designs usually exhibit higher connectivity and more complex data dependencies among signals. As modules become more interconnected, a larger fraction of the bits computed throughout the design eventually contributes, either directly or indirectly, to observable outputs or state-holding elements. In contrast, smaller designs often contain localized computations, unused upper bits, or partially utilized datapaths that are easier for the analysis to

classify as non-observable. Consequently, although large designs may still contain a substantial absolute number of don't care bits, these bits represent a smaller fraction of the overall design.

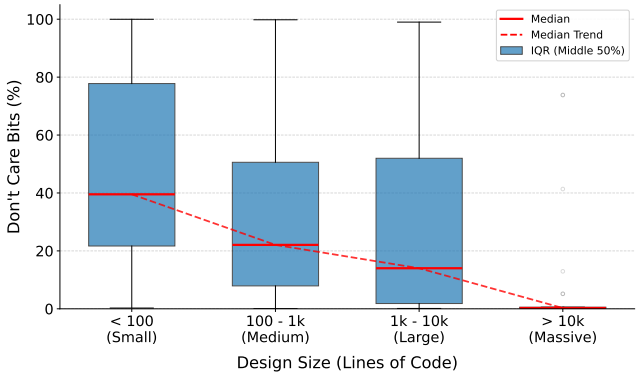

Figure 11: Comparison of the proportion of identified don't care bits across different design sizes.

Structural Origins of Don't Cares: We also examined how the frequency of specific intermediate operations correlates with the absolute number of inferred don't care bits within a design. Figure 12 presents the Pearson correlation coefficients for key operations, including bit extraction, logical AND, multiplexing, addition, module instantiation, and concatenation. The data suggests that certain operations, particularly multiplexing and concatenation, are more strongly associated with the emergence of don't care bits. In contrast, operations such as addition and module instantiation appear to contribute less significantly to the generation of non-observable signals.

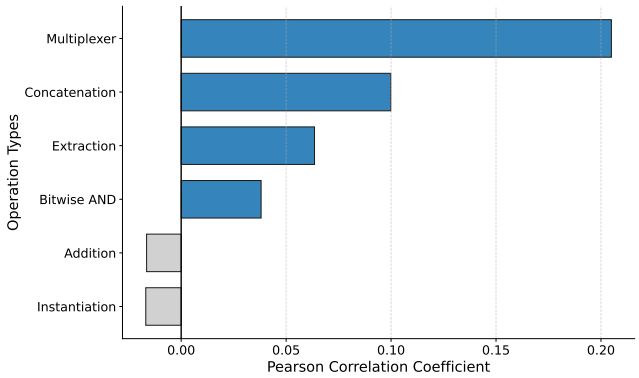

Figure 12: Pearson correlation between the frequency of specific hardware operations and the absolute number of inferred don't care bits.

6 Related Work

The Don't Care Analysis presented in this paper sits at the intersection of three research communities: logic synthesis and electronic design automation, bit-level compiler analyses, and static analysis frameworks for hardware description languages. We survey the most closely related work in each area and discuss how our contribution differs.

Don't Cares in Logic Synthesis and EDA. The notion of observability don't cares has a long history in logic synthesis. Savoj et al. introduced a formal treatment of *controllability* and *observability* don't cares for Boolean networks in the context of the SIS synthesis system [11], establishing the conceptual foundation that our analysis builds upon: a bit is a don't care if and only if its value cannot be observed at any primary output. The ESPRESSO logic minimizer [2] exploits don't cares for two-level Boolean minimization, while the ABC synthesis and verification system [1] uses observability don't cares in optimization passes such as resubstitution and refactoring. Our work can be seen as a principled, scalable reimagining of observability don't care computation for a modern SSA-based hardware intermediate representation, replacing the classical BDD and SAT machinery with a lightweight bitwise dataflow analysis that integrates directly into the CIRCT compiler infrastructure.

Bit-Level Analyses in Compilers. From the programming languages perspective, our analysis is an instance of the backward must dataflow framework formalized by Kildall [8] and placed on rigorous abstract interpretation foundations by Cousot and Cousot [5]. The bitwise lattice we employ is closely related to what compiler writers call *demanded bits* or *bit-level liveness*: the set of bits of a value that are actually consumed by downstream operations. Both GCC and LLVM implement variants of this idea to drive strength reduction, dead code elimination, and integer narrowing. Whereas these compiler analyses target general-purpose software and operate on machine-word-sized integers, our analysis is designed for the bit-precise semantics of hardware designs, where signal widths are arbitrary and every bit position carries independent physical significance. The SSA form of μ-CIRCT allows us to connect cleanly to this tradition while handling hardware-specific operations such as concatenation, slicing, and carry-propagating addition.

Static Analysis Frameworks for HDLs. Closest to our work in terms of target language are recent efforts to bring principled static analysis to Verilog and related HDLs. Chen et al. present an operational semantics for Verilog and use it as the basis for a general-purpose static analysis framework [4], demonstrating that the bit-precise nature of hardware programs demands analyses that differ fundamentally from those used in software. Earlier work by Salama et al. uses dependent types to check consistency of Verilog wire interconnects [10], and Herklotz et al. apply formal methods to verify the correctness of high-level synthesis tools [7]. Our contribution complements these efforts by providing the first formalized, bit-level *observability* analysis for Verilog, implemented in the CIRCT infrastructure and evaluated at scale on the ChiBench benchmark suite [12]. Unlike general-purpose frameworks, our analysis is specialized to the must dataflow setting, admitting a simple lattice, a small set of compositional transfer functions, and a proof of soundness that connects directly to the operational semantics of μ-CIRCT.

7 Conclusion

This paper has presented a don't care analysis for Verilog designs implemented on top of the CIRCT infrastructure. The analysis models observability as a backward may data-flow problem over a bitwise lattice, enabling the identification of non-observable bits in hardware computations. We formalized the analysis over the μ-CIRCT intermediate language, proved its termination and soundness, and evaluated it on more than 21 thousand real-world Verilog designs from the ChiBench benchmark suite. Our results show that the analysis scales approximately linearly with design size, introduces only modest compilation overhead, and is able to identify substantial amounts of semantically irrelevant computation in a significant subset of the evaluated benchmarks.

Despite these encouraging results, the current work still has important limitations. The transfer functions implemented in this work remain conservative for several operations, particularly arithmetic operators, where more precise reasoning about carry propagation could identify additional don't care bits. Furthermore, the analysis currently operates as a standalone observability pass and does not yet integrate with optimization or synthesis transformations capable of exploiting the inferred information.

These limitations open several opportunities for future work. One promising direction is the development of more precise transfer functions for arithmetic and bit-manipulation operations, potentially incorporating symbolic techniques. Finally, integrating the proposed analysis with optimization passes in CIRCT could enable practical reductions in area, power consumption, and compilation effort by automatically eliminating semantically irrelevant hardware computations.

Data Availability

The full implementation of the don't care analysis, plus the benchmarks in Section 5 are publicly available at https://github.com/lac-dcc/manticore.

Acknowledgment

This project was supported by a grant from Cadence Design Systems. The authors also acknowledge the support of FAPEMIG (Grant APQ-00440-23), CNPq (Grant #444127/2024-0), and CAPES (PrInt). The authors acknowledge the use of generative AI tools (Gemini 3.1 and GPT-4o) to improve the clarity and readability of the manuscript. Usage was solely for language revision; all technical content, ideas, and conclusions are the responsibility of the authors.

References

[1] Robert Brayton and Alan Mishchenko. Abc: an academic industrial-strength verification tool. In *CAV*, page 24–40, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] Robert K Brayton, Gary D Hachtel, Curt McMullen, and Alberto Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis*, volume 2. Springer Science & Business Media, 1984.

[3] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Jiacai Cui, Tian Tan, Xiaoxing Ma, Chang Xu, Jian Lu, and Yue Li. Qihe: A general-purpose static analysis framework for verilog, 2026.

[4] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. The essence of verilog: A tractable and tested operational semantics for verilog. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[8] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[9] Samir Palnitkar. *Verilog HDL: a guide to digital design and synthesis*, volume 1. Prentice Hall Professional, Hoboken, New Jersey, U.S., 2003.

[10] Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. Static consistency checking for verilog wire interconnects: using dependent types to check the sanity of verilog descriptions. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, page 121–130, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Hamid Savoj and Robert K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, page 297–301, New York, NY, USA, 1991. Association for Computing Machinery.

[12] Rafael Sumitani, João Victor Amorim, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. Chibench: a benchmark suite for testing electronic design automation tools, 2024. https://arxiv.org/abs/2406.06550.