Handling Don't-Care Conditions in High-Level Synthesis and Application for Reducing Initialized Registers

Hong-Zu Chou[†], Kai-Hui Chang[‡], and Sy-Yen Kuo[†]
[†]Electrical Engineering Department, National Taiwan University, Taipei, Taiwan
[‡]Avery Design Systems, Inc., Andover, MA, USA
sykuo@cc.ee.ntu.edu.tw

ABSTRACT

Don't-care conditions provide additional flexibility in logic synthesis and optimization. However, most work only focuses on the gate level because it is difficult to handle such conditions accurately at the behavior and register transfer levels, which is problematic since the trend is to move toward high-level synthesis. In this work we propose innovative methods to handle such conditions accurately at high-level designs. In addition, we propose two novel algorithms based on our new methods to minimize the number of registers that need to be initialized at the architecture level, which can reduce the routing resources used by the reset signals and alleviate the routing problem. Our results show that we can identify 53% of the registers that can be uninitialized in a 5-stage pipelined processor within 5 minutes, demonstrating the effectiveness of our approach.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Optimization*

General Terms

Design, Verification

Keywords

Don't-Care (DC), RTL symbolic simulation, Synthesis

1. INTRODUCTION

Don't-care conditions have been widely exploited by synthesis tools to improve netlist quality. In order to enhance the efficiency and accuracy of such synthesis techniques, a great deal of effort has been invested [3,6,9,11,12,15]. However, most existing techniques focus on optimizing an existing gate-level netlist only. Not being able to utilize such DCs at a higher-level of abstraction, such as the Register Transfer Level (RTL), greatly reduces the optimization power of high-level synthesis tools [11]. In particular, once a netlist has been generated, DCs often provide local optimization opportunities only and cannot change the netlist structure significantly. On the other hand, if DCs are known at the RTL, the synthesis tool can potentially generate quite different netlists for better optimizations. Not being able to utilize don't-cares in the RTL will undoubtedly become a serious limitation of such synthesis tools in the future.

One major reason why DCs are rarely utilized at the RTL for logic synthesis is that it is difficult to handle such DCs correctly at this level. As Haufe and Rogin suggested [7], DC (or X) is one major source of RTL and gate-level mismatch. One previous research suggests to eliminate X-values throughout the design using a two-state methodology [2]. However, this methodology can significantly reduce synthesis quality because DCs are no longer available for optimization. Our first contribution in this work is to use high-level[1] symbolic simulation to accurately handle the propagation of X values. In addition, we propose a SAT formulation that can prove whether or not the X values will be propagated to any observation points, such as primary outputs or registers. Our techniques not only provide more optimization opportunities in high-level synthesis but also solve many verification problems [1, 14]. In this way, designers can use X values whenever appropriate and verify whether those Xs are indeed don't cares using our methods, producing more optimization opportunities for synthesis tools.

Being able to handle X conditions accurately at higher levels enables many logic optimizations. For instance, with the miniaturization of transistors, routing has become a serious problem that attracts much attention [10]. By reducing the number of registers which need to be initialized, the routing problem can be alleviated because routing resources used by the reset signal can be reduced. Our second contribution is two algorithms for finding registers that can be uninitialized in a design. The first algorithm is optimal and can find the maximum set of such registers; however, it is computation intensive. The second algorithm is a faster heuristic that finds approximate solutions. Our empirical results show that the heuristic algorithm can produce good results within significantly shorter time compared with the optimal one. Moreover, when the algorithm is applied to a 5-stage pipelined processor, it found that approximately 53% of the control registers do not need to be initialized for a reset period that is 5 cycles long. These results show that our techniques can improve synthesis quality dramatically and help alleviate routing problems.

The rest of this paper is organized as follows. Our symbolic-based X-value checking algorithm is presented in Section 2. In Section 3, we propose new methodologies to find the minimum number of registers that need to be initialized. Empirical results are shown in Section 4, and Section 5 concludes this paper.

2. HANDLING X-VALUES AT THE RTL

In this section, we describe how symbolic simulation can be used to accurately handle X-propagation at the RTL and check the observability of those X-values. The reason why we chose symbolic simulation is because it is a combination of hardware and software verification methods, thus it can natively handle high-level code without the synthesis step [5]. Furthermore, handling X-values at higher levels can also provide additional benefits for synthesis and verification [11].

[1]We only use RTL in the text for simplicity, but our techniques can also be applied to ESL as long as the design can be symbolically simulated.

DAC'09, July 26-31, 2009, San Francisco, California, USA

2.1 Handling X-Propagation at the RTL

Although many logic simulators support 3-value (0/1/X) simulation, the handling of X-values is often inaccurate. Consider a simple example shown in 1(a), the RTL code includes an uninitialized register i which controls the value of output o. In this example, if i has X-value, output o can be either 0 or 1. However, logic simulation can only take one branch and makes $o = 1$ according to the Verilog standard. Consequently, simulation mismatch between RTL code and the synthesized netlist could occur. Unlike logic simulation, symbolic simulation treats the X-value in i as a symbol and produces output expressions in terms of the symbol. Since a symbol represents both 0 and 1, symbolic simulation can handle all possible values of i simultaneously, as shown in Figure 1(c). Therefore, we can employ symbolic simulation to accurately handle X-propagation at the RTL.

(a)
```
module X (o);
output reg o; reg i;
always @(i)
  if (i == 1)  o = 0;
  else         o = 1;
endmodule;
```

(b) Logic simulation results:
$o = 1$

(c) Symbolic trace:
o = multiplex(i) –
1: 1'b0;
0: 1'b1;

Figure 1: Simulating X using logic and symbolic simulation. Assume i=1'bx, symbolic simulation produces more accurate results.

2.2 Checking the Observability of X-Values

By using symbolic simulation to handle the propagation of X-values, those values can be handled accurately at the RTL. However, even if the symbols representing X propagated to primary outputs, it does not necessarily mean that those X-values will affect primary outputs. In this section we formulate the X-value checking problem as a SAT problem. In particular, we use symbolic simulation to generate logic expressions to form a SAT instance, and rely on SAT engines to solve the problem. Note that although similar SAT constructions have been proposed for error diagnosis and equivalence checking [4, 13], the SAT instance formulation proposed in this paper is still different.

2.2.1 Problem Formulation

The main objective of X-value checking is to determine whether any X will propagate to any observation points (e.g., primary outputs at each cycle, register values at the end of the reset period, etc.), and the problem is formulated as follows. Given a design containing N primary inputs PI_1, PI_2, ..., PI_N, M registers R_1, R_2, ..., R_M and K observation points O_1, O_2, ..., O_K, check whether any O_i can be affected by an uninitialized R_j within a given number of cycles C, where $1 \le i \le K$ and $1 \le j \le M$. C is the number of reset cycles for the design, and it is often defined in the specification. Alternatively, our experiments suggest that one can incrementally prolong the reset period until the number of uninitialized registers no longer increases. Note that although in our current formulation we only consider X-values in registers, our methods can be applied easily to any signal in the design.

2.2.2 Converting the Problem to a SAT Instance

To solve the problem formulated in Section 2.2.1, we propose a new method that converts the problem to a SAT instance. The built instance is shown in Figure 2, and the method works as follows.

1) Duplicate design A to create an identical copy A'. Use symbol X_{Ai} to represent the initial state of register R_i in design A. Similarly, use symbol $X_{A'i}$ for design A'.
2) Inject a symbol $PI_{i@c}$ to both design's primary inputs PI_i at cycle c, perform symbolic simulation for C cycles.
3) Observation points $OA_{1@c}$, ..., $OA_{K@c}$ and $OA'_{1@c}$, ..., $OA'_{K@c}$ are connected to a miter. When the miter's output is 1, it means there exists a pair of observation points, $OA_{i@c}$ and $OA'_{i@c}$, whose values are different; otherwise, the equivalency of these observation points is proved.

Note that Figure 2 only shows a snapshot at cycle c for PI and O. The SAT instance should include PI and O from cycle 1 to cycle C. Finally, a SAT solver is called to find if a solution exists to make miter = 1. If the solver can find a solution, then "X"s can propagate to one or more observation points. The reason is that if we have a circuit and apply the SAT solutions to the primary inputs, then we will have two different initial states that can make the values at observation points different, which means the "X" can be propagated out. On the other hand, if the solver proves that the SAT instance is unsatisfiable, then we know that none of the "X" can be propagated to the observation points.

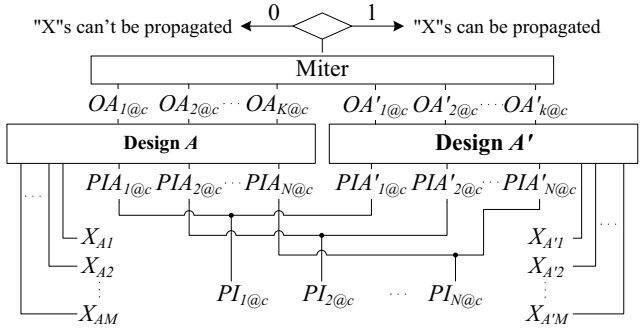

Figure 2: Illustration of symbolic X-propagation checking ($PI_{i@c}$, $PIA_{i@c}$ and $PIA'_{i@c}$ denote primary inputs, X_{Ai} and $X_{A'i}$ denote registers' initial states, and $OA_{i@c}$ and $OA'_{i@c}$ denote observation points)

3. REDUCING INITIALIZED REGISTERS

In this section we propose two algorithms to minimize the number of registers based on the techniques described in Section 2. The first one can find the minimum set of registers that need to be initialized, while the second one can only get an approximate result but requires lower computation cost.

3.1 Finding Optimal Solution

Our first method that finds the minimal number of initialized registers is based on the SAT-instance construction described in Section 2.2.2. To serve the new purpose, however, two components are added to the SAT-instance formulation, including 1) multiplexers and 2) cardinality constraints [13]. A multiplexer is added to every register to model the initial state in the SAT formulation. More specifically, when the select line S_i is 1, the initial states of X_{Ai} and $X_{A'i}$ are both 0, which means register R_i is initialized to 0. When S_i is 0, X_{Ai} and $X_{A'i}$ are regarded as free variables and become the primary inputs of the SAT instance; i.e., they are uninitialized. An example of the multiplexer is given in Figure 3(a), where symbol "X" denotes a free variable that can take any value. Cardinality constraints, as shown in Figure 3(b), restrict the number of select lines that can be set to 1 simultaneously to m. This number also represents the number of registers that need to be initialized. The cardinality constraints are implemented by an adder that performs a bitwise addition of the select lines and outputs the sum, m.

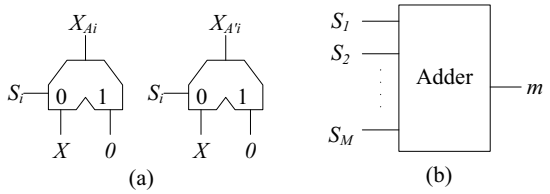

Figure 3: Additional constructs added to Figure 2 for finding the minimum set of initialized registers

The algorithm is presented in Figure 4. Variable ss denotes a set of registers to be initialized, and SS_m denotes the solution space that contains different combinations of exactly m registers that should be initialized, where $0 \leq m \leq M$ (M=*total number of registers*). Function *check_X*($miter$, m, SS) denotes the procedure to check the observability of X-values in solution space SS under the constraints $miter$ and m using the constructs shown in Figure 2 and Figure 3. Initially, m is set to 0, and a SAT solver is called to find if a solution exists to make $miter = 1$ under constraint m. If the problem is satisfiable and solution ss is returned, it means that initializing the combination of m registers returned by *check_X* in the solution ss will propagate X-values to observation points, and ss should be excluded from the solution space SS_m. The procedure is repeatedly performed until no solution is returned. At this point, two situations should be considered: 1) SS_m is empty, which means that if only m registers are initialized, no matter which combination, X values in one or more of the uninitialized registers will be propagated to observation points. When this happens, we increment m to look for solutions with one more register to initialize. 2) SS_m is not empty, which means initializing m registers using any combination of registers that remain in SS_m can make sure no X in the uninitialized registers can be propagated to observation points, and the solution is returned. Note that observation points typically include two types of signals: primary outputs at every cycle and registers at the end of the reset period. In practice, users can decide which signals to observe based on their requirements. For instance, if the Xs at primary outputs during reset can be ignored, users do not have to make them observation points.

```
01 for (m = 0; m <= M; m++)
02     ss = check_X(miter = 1, m, SS_m);
03     while (ss)
04         SS_m = SS_m \ {ss};
05         ss = check_X(miter = 1, m, SS_m);
06     if (SS_m is not empty) return SS_m;
```

Figure 4: Optimal method for finding a minimum set of registers that need to be initialized.

The above formulation uses the constructs shown in Figure 2 and 3 to find and exclude all combinations of m registers that when only those registers are initialized, the X values in the uninitialized variables will be propagated to observation points. When *check_X* can no longer find a solution under constraint m, it means choosing any combination of m registers that still exists in SS_m will not propagate X-values to the observation points. Since we start our search from $m = 0$, we can find the minimal number of registers that should be initialized. This problem is at least NP-complete (the proof is omitted due to space limitations).

3.2 Fast Heuristic Algorithm

In order to reduce the computation cost, we design a heuristic algorithm that finds a large set of registers that can be uninitialized. The pseudo-code of the algorithm is presented in Figure 5. Initially, *check_X* is used to find sets of registers to initialize so that no X can be propagated to the observation points under constraint $m = M - 1$ (i.e., only one register is uninitialized), and these solutions are added to a set $seed_{M-1}$. After that, solution space SS_m is created by spanning the solutions comprised in $seed_{m+1}$, where $0 \leq m \leq M - 2$. This is achieved in the *span* function by adding each register in $seed_{M-1}$ to each combination of registers in $seed_{m+1}$ in line 4 to form a new set of candidate combinations of registers for *check_X* in SS_m. To avoid significant cost for checking all possible solutions in SS_m, the algorithm randomly selects only T solutions, $ss_1, ss_2, \ldots, ss_T$, for X-propagation checking, where T is a pre-defined threshold determined empirically. If ss_i cannot propagate X-values, it will be added to $seed_m$. The same procedure is performed repeatedly with a smaller m until $seed_m$ is empty. At this point, $seed_{m+1}$ includes the solutions that $M - (m + 1)$ registers can be uninitialized.

```
01 seed_{M-1} = {ss_i | ss_i ∈ SS_{M-1} &&
02         check_X(miter = 1, m = M − 1, ss_i) == null};
03 for (m = M − 2; m >= 0; m − −)
04     SS_m = span(seed_{m+1}, seed_{M-1});
05     {ss_1, ss_2, ... ss_T} = randomly_select(SS_m);
06     foreach (ss_i)
07         if (check_X(miter = 1, m, ss_i) == null)
08             seed_m = seed_m ∪ ss_i;
09     if (seed_m is empty)
10         return seed_{m+1};
```

Figure 5: Heuristic method for finding a large set of registers that can be uninitialized.

Two efficient methods can be employed to improve the performance of the heuristic algorithm. The first one is that for the registers that should be initialized, we can actually make them stay at 0/1 for all cycles, not just the first cycle. The reason is that since they are going to have the reset signal, they can be forced to 0/1 during the reset period. The second one is that the primary inputs can be constant scalar values instead of symbols. Since people usually have direct access to primary inputs, and we are only concerned about the final state after the reset period, we can apply a known sequence to the primary inputs during the reset period. These scenarios can reduce the solution space and make SAT solver run faster. Obviously, the sequence used may affect the number of registers that need to be initialized. Finding a good sequence is our future work.

4. EXPERIMENTAL RESULTS

Designs PIPE and DLX are used in our experiments to evaluate the performance of optimal and heuristic algorithms. Since the heuristic algorithm is based on a random scenario, we ran each configuration of experiment 30 times to obtain their average. Only register values at the end of the reset period are considered as the observation points, and the experiments are conducted with a commercial symbolic simulator called Insight [16] that supports high-level code.

Comparing Optimal and Heuristic Algorithms Using PIPE: To compare the performance of the optimal and the heuristic algorithms, we developed a simple design called PIPE containing 12 word-level registers, and each register can be uninitialized after certain cycles. The results are shown in Table 1. The optimal algorithm can always find the minimum set of registers that need to be initialized as expected; however, its runtime is long due to its optimal nature (more than 1000 seconds when the number of cycles is smaller than 5). Note that the runtime of optimal algorithm increases rapidly as the number of registers increases, making it difficult to be applied to practical designs. Comparatively, the heuristic algorithm is a greedy approach that checks T possible combinations per iteration only. As shown in Table 1, when $T = 2$, the heuristic algorithm can get the same results as the optimal algorithm. We also observe that as the number of cycle increases, SAT solver also requires more time to produce the result. However, the maximum runtime is still shorter than 4 seconds, suggesting that our heuristic algorithm can produce good results within a significantly shorter time.

Evaluating Heuristic Algorithm on DLX: DLX is a 32-bit RISC microprocessor with 5-stage pipeline designed by Hennessy and Patterson [8]. The BugUnder Ground project from Michigan [17] provides a DLX implementation, and it is used to evaluate the performance of our heuristic approach. The DLX implementation has 75 control registers, 32 general

Table 2: The performance of heuristic algorithm on DLX (C=1 to 5, T=2 to 6)

	$T=2$		$T=3$		$T=4$		$T=5$		$T=6$	
Cycles C	Uninit. Reg.	Runtime(s)	Uninit. Reg.	Runtime(s)	Uninit. Reg.	Runtime(s)	Uninit. Reg.	Runtime(s)	Uninit. Reg.	Runtime(s)
1	18.3	101.98	21.07	117.8	24.9	154.01	25.37	196.04	26.33	207.88
2	30.6	126.22	32.93	156.12	32.83	209.34	34.47	226.83	34.5	230.55
3	35.87	139.78	36.93	170.43	37.47	235.02	37.67	246.44	37.43	251.45
4	37	141.91	37.67	171.65	37.83	241.89	37.9	250.43	37.97	254.56
5	40	144.5	40	177.26	40	250.82	40	249.99	40	260.38

Table 1: Comparison of optimal and heuristic algorithms using PIPE

	Optimal		Heuristic ($T=2$)	
Cycles C	Init. Reg.	Runtime(s)	Init. Reg.	Runtime(s)
1	11	2414.69	11	1.91
2	10	2401.49	10	1.82
3	8	2326.83	8	1.8
4	6	1531.38	6	1.98
5	5	841.76	5	2.15
6	3	82.23	3	2.07
7	3	85.3	3	2.06
8	2	17.46	2	2.22
9	1	2.39	1	2.48
10	1	2.47	1	2.55
11	1	2.39	1	2.54
12	0	0.01	0	3.48

registers and 3 dummy registers (RDaddr_reg, Wrt_en and Control_state in cpu.v). Note that the number of registers in the synthesized netlist is 1652. It is obvious that utilizing don't-cares at the RTL is much more efficient than the gate level because only 75 word-level registers need to be handled instead of 1652 gate-level registers.

We initialize general and dummy registers by default because their X-values are known to be propagated. In addition, we applied the methods mentioned in Section 3.2 to improve the performance of SAT solving. Specifically, primary inputs Iin and Din are constrained to 0, and registers that need be initialized are forced to 0 for all cycles.

We examined the effect of varying the number of cycles C and the threshold T on the results. As shown in Table 2, as the number of cycles increases, there is a gradual increase of registers that can be uninitialized because the Xs may be masked with additional cycles. Therefore, in practice one can gradually increase C until the number of uninitialized registers stops increasing.

Since DLX is a 5-stage processor, many registers that can be uninitialized will be identified within 5 cycles. Therefore, when $C = 5$ the proposed approach found that 40 registers do not propagate their X-values. We also observe that when the threshold T becomes larger, more combinations of registers will be checked, and better results can be produced. It is interesting to note that the results of $C = 5$ are the same regardless of the value of T. The reason is that all registers contained in $seed_{74}$ can be uninitialized after 5 cycles, and the SAT solver cannot find any satisfiable solution from the constrained solution space. We also analyzed the reason why some registers must be initialized and found that they are intermediate registers to store temporary values for specific instructions. Since we bind primary inputs to 0, those instructions cannot be generated to initialize those registers.

Runtime of the above experiments are listed in Table 2. Apparently, the required runtime increases with the number of cycles C and the threshold T because of the complexity in SAT translation and satisfiability checks. However, the average maximum runtime is still smaller than 5 minutes. The empirical results show that the heuristic approach can find approximately $40/75 = 53\%$ registers that can be uninitialized; therefore, we do not need to use a large threshold which may cause significant increase in runtime.

5. CONCLUSION

Don't-care (X) conditions provide additional flexibility for synthesis optimizations. However, existing techniques typically focus on optimizing gate-level netlists only, mostly due to the difficulty to handle X accurately at the RTL. Not being able to use don't-care conditions at those levels hurts the quality of synthesized netlists, and this limitation will become more problematic in the future since the trend is to move even more toward high-level synthesis. To address this problem, we propose innovative techniques to accurately handle X-values using high-level symbolic simulation and check observability of X-values using SAT-solvers. In addition, we utilize the don't-cares for a novel optimization: identifying the registers that can be uninitialized in high-level designs for synthesis optimizations. This new optimization reduces the number of registers that need reset signals and can alleviate routing problems. Our proposed methods found that 53% of control registers in a DLX processor can be uninitialized for a reset period that is 5 cycles long, and the runtime is smaller than 5 minutes, suggesting that our methods provide a practical building block for new high-level synthesis optimizations.

Acknowledgments

This research was supported by the National Science Council, Taiwan under Grant NSC 97-224-E-002-216-MY3; Excellent Research Projects of National Taiwan University, 95R0062-AE00-05; and Small Business Innovation Project of Ministry of Economic Affairs, Taiwan, 1Z960401.

6. REFERENCES

[1] D. Brand, R. A. Bergamaschi and L. Stok, "Be Careful with Don't Cares," *DAC'95*, pp. 83-86.
[2] L. Bening, "A Two-State Methodology for RTL Logic Simulation," *DAC'99*, pp. 672-677.
[3] R. A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, and S. Prakash, "Efficient Use of Large Don't Cares in High-Level and Logic Synthesis," *ICCAD'95*, pp. 272-278.
[4] K. H. Chang, I. L. Markov, and V. Bertacco, "Functional Design Errors in Digital Circuits: Diagnosis, Correction and Repair", Springer 2008.
[5] D. W. Currie, A. J. Hu, and S. Rajan, "Automatic Formal Verification of DSP Software," *DAC'00*, pp. 130-135.
[6] M. Damiani and G. De Micheli, "OObservability Don't Care Sets and Boolean Relations," *ICCAD'90*, pp. 502-505.
[7] C. Haufe and F. Rogin, "Ad-Hoc Translations to Close Verilog Semantics Gap," *workshop on IEEE DDECS'08*, pp. 1-6.
[8] J. L. Hensessy, and D. J. Patterson, "Computer Architecture: A Quantitative Approach, 2nd edition," Morgan Kaufman, 1996.
[9] A. Mishchenko and R. K. Brayton, "SAT-Based Complete Don't-Care Computation for Network Optimization," *DATE'05*, pp. 412-417.
[10] M. D. Moffitt, J. A. Roy, and I. L. Markov, "The Coming of Age of (Academic) Global Routing," *ISPD'08*, pp. 148-155.
[11] R. Ranjan, Y. Antonioli, A. Hunter, and O. Petlin, "Formal Verification Enables Safe X Handling," Dec. 2008. http://www.scdsource.com/article.php?id=324
[12] H. Savoj and R. K. Brayton, "Observability Relations and Observability Don't Cares," *ICCAD'91*, pp. 518-521.
[13] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability," *ASPDAC'04*, pp.218-223.
[14] M. Turpin, "The Dangers of Living with an X," SNUG Boston, 2003.
[15] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't-Cares," *DAC'06*, pp. 229-234.
[16] Avery Design Systems Inc., http://www.avery-design.com
[17] Bug UnderGround, http://bug.eecs.umich.edu