FPGA Glitch Power Analysis and Reduction

Warren Shum and Jason H. Anderson

Department of Electrical and Computer Engineering, University of Toronto
Toronto, ON. Canada
{shumwarr, janders}@eecg.toronto.edu

***Abstract*—This paper presents a don't-care-based synthesis technique for reducing glitch power in FPGAs. First, an analysis of glitch power and don't-cares in a commercial FPGA is given, showing that glitch power comprises an average of 26.0% of total dynamic power. An algorithm for glitch reduction is then presented, which takes advantage of don't-cares in the circuit by setting their values based on the circuit's simulated glitch behavior. Glitch power is reduced by up to 49.0%, with an average of 13.7%, while total dynamic power is reduced by up to 12.5%, with an average of 4.0%. The algorithm is applied after placement and routing, and has zero area and performance overhead.**

Index Terms – FPGA, glitch power, don't-cares, SAT

I. Introduction

Field-programmable gate arrays (FPGAs) are highly desirable for the implementation of digital systems due to their flexibility and low time-to-market. However, their programmability comes at a cost to power consumption. Analyses show an increase of as much as 10x the power of a functionally-equivalent ASIC design [7]. This can be a barrier to FPGA use in power-sensitive applications, such as mobile devices.

Power in FPGAs can be divided into two categories: static power and dynamic power. Static power is due to current leakage in transistors. Dynamic power is a result of signal transitions between logic-0 and logic-1. These transitions can be split into two types: functional transitions and glitches. Functional transitions are those which are necessary for the correct operation of the circuit. Glitches, on the other hand, are transitions that arise from unbalanced delays to the inputs of a logic gate, causing the gate's output to transition briefly to an intermediate state. Although glitches do not adversely affect the functionality of a synchronous circuit (as they settle before the next clock edge), they have a significant effect on power consumption. The work in [8] estimates that glitch power in FPGAs comprises from 4% to 73% of total dynamic power, with an average of 22.6%. This is a significant motivator for the reduction of glitch power.

Don't-cares are an important concept in logic synthesis and are frequently used for the optimization of logic circuits. A don't-care of a logic function within a larger circuit is an input state for which the function's output can be *either* logic-0 or logic-1, without affecting the circuit's correctness. Don't-cares can come from external constraints or from within the circuit itself. An external constraint may be specified by the designer (e.g. asserting that a certain input combination will never be applied). A logic function within a circuit may also have don't-cares due to its surrounding logic, for example, if the logic feeding the function's fanins can never satisfy a certain input combination, or if the function's output does not affect the circuit's primary outputs under certain circumstances.

In this paper, we present a glitch reduction optimization algorithm based on don't-cares. It sets the output values for the don't-cares of logic functions in such a way that reduces the amount of glitching. This process is performed *after* placement and routing, using timing simulation data to guide the algorithm. Relative to prior published FPGA glitch reduction techniques, our approach is entirely new, and leverages the ability to re-program FPGA logic functions without altering the placement and routing. Since the placement and routing are maintained, this optimization has zero cost in terms of area and delay, and can be executed after timing closure is completed.

The paper is organized as follows: Section II provides related work and background on glitches and don't-cares. Section III presents the motivation for this work and an analysis of the glitch characteristics of circuits in a commercial $65nm$ FPGA, which to our knowledge is the first study of its kind. Section IV describes our glitch reduction algorithm. Section V describes our experimental methodology and results, and Section VI concludes the paper.

II. Background

A. FPGA Architecture

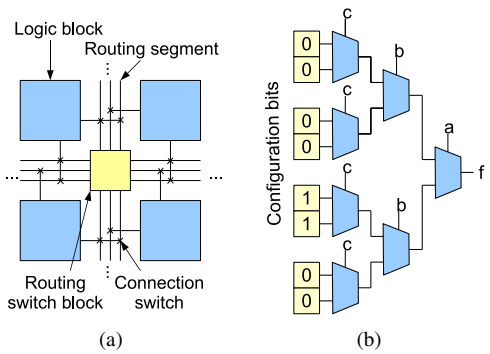
Fig. 1(a) shows a section of a typical island-style FPGA architecture. It is composed of logic blocks connected to one another through a programmable routing network. Programmable routing switches (shown as x's in Fig. 1(a)) allow pins on logic blocks to be programmably connected to pre-fabricated metal wire segments, and also allow wire segments to be programmably connected with one another to form routing paths.

Inside the logic blocks, logic functions are implemented using look-up-tables (LUTs). An example is shown in Fig. 1(b). A k-input LUT can implement any logic function of up to k variables. In essence, a LUT is a hardware implementation of a truth table, where the output value for each minterm is held in an SRAM configuration cell (bit). A k-input LUT requires 2^k configuration bits. For this work, we target an FPGA that contains 6-input LUTs, which are typical of modern commercial FPGA architectures [2], [17].

B. Glitch Power in FPGAs

The dynamic power consumed by an FPGA can be modeled by the formula

978-1-61284-660-6/11/$26.00 © 2011 IEEE



(a) (b)

Fig. 1. (a) Logic blocks and routing in an island-style FPGA architecture. (b) Example of a 3-input LUT (look-up-table) with truth table in Table I.

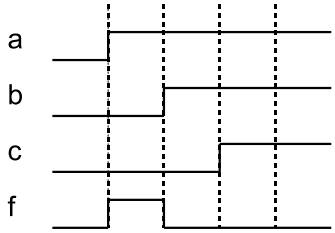


Fig. 2. Example waveform showing a glitch on the output of a LUT f with truth table given in Table I.

$$P_{dyn} = \frac{1}{2}\sum_{i=1}^{n} S_i C_i f V_{dd}^2 \quad (1)$$

where n is the number of nets in the circuit, S_i is the switching activity of net i, C_i is the capacitance of net i, f is the frequency of the circuit, and V_{dd} is the supply voltage. The glitch reduction algorithm presented in this paper aims to lower the switching activity as a means of reducing dynamic power.

As a result of the differences in delays through the routing network and LUTs themselves, signals arriving at LUT inputs may transition at different times, leading to glitches. An example is shown in Fig. 2. This LUT implements the 3-input function given in Table I. Consider the case where the inputs transition from $000 \rightarrow 111$. Ideally, the output f would remain constant at 0. However, varying arrival times on the inputs may cause an input transition sequence such as $000 \rightarrow 100 \rightarrow 110 \rightarrow 111$, causing f to make a $0 \rightarrow 1 \rightarrow 0 \rightarrow 0$ transition rather than remaining at 0. This leads to extra power consumed by the LUT and any of its fanouts that propagate the glitch. Furthermore, the glitch is propagated through the FPGA interconnect which presents a high capacitive load due to its long metal wire segments and programmable (buffered) routing switches. Prior work has shown, in fact, that interconnect accounts for 60% of total FPGA dynamic power [14].

C. Glitch Reduction in FPGAs

Glitch reduction techniques can be applied at various stages in the CAD flow. Since glitches are caused by unbalanced path delays to LUT inputs, it is natural to design algorithms that attempt to balance the delays. This can be done at the technology mapping stage [3], in which the mapping is chosen based on glitch-aware switching activities. Another approach operates at the routing stage [5], in which the faster-arriving inputs to a LUT are delayed by extending their path through the routing network. Delay balancing can also be done at the architectural level. The work in [8] inserts programmable delay elements to balance the arrival times of signals at LUT inputs. However, these approaches all incur an area or performance cost.

Some works use flip-flop insertion or pipelining to break up deep combinational logic paths which are the root of high glitch power. Circuits with higher degrees of pipelining tend to have lower glitch power because they have fewer logic levels, thus reducing the opportunity for delay imbalance [16]. Flip-flops with shifted-phase clocks can be inserted to block the propagation of glitches [9]. Another work in [4] uses negative edge-triggered flip-flops in a similar fashion, but without the extra cost of generating additional clock signals. It is also possible to apply retiming to the circuit by moving flip-flops to block glitches [6].

Our work draws inspiration from hazard-free logic synthesis techniques for asynchronous circuits, such as [10]. In asynchronous circuits, glitches (hazards) cannot be tolerated because they may produce incorrect behavior (consider, for example, the disasterous effect of a glitch on a handshaking signal). Our work is different in that while hazards are tolerable from a functionality standpoint, it is beneficial to remove them to reduce power consumption.

A key feature of the work presented here is that it has no impact on the rest of the design flow. It is applied after placement and routing, and as a consequence, the algorithm has no cost in terms of performance or area. Other methods incur additional area/delay from the inclusion of delay elements, registers and extra routing resources, as well as disrupting the synthesis and layout of the circuit in an unpredictable way. Our approach maintains the results of the existing compilation while only making changes to the don't-cares within LUT truth table configuration bits. This zero-overhead method is a highly desirable quality not shared by previous glitch reduction approaches.

abc	f	Care
000	0	Y
001	0	Y
010	0	Y
011	0	Y
100	1	N
101	1	Y
110	0	N
111	0	Y

TABLE I
GLITCH EXAMPLE TRUTH TABLE FOR A LOGIC FUNCTION WITH INPUTS abc AND OUTPUT f. A POSSIBLE EXAMPLE OF CARES IS GIVEN (*care = Y, don't-care = N*)

D. Don't-Cares in Logic Circuits

To prevent glitches, we take advantage of *don't-cares*. These are entries in the truth table where a LUT's output can be set as either logic-0 or logic-1 without affecting the correctness of the

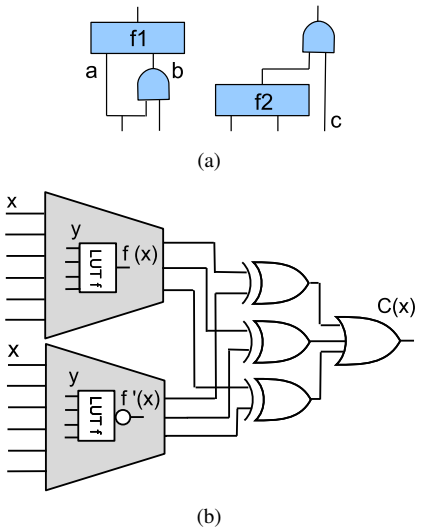


Fig. 3. (a) Example of SDCs (left) and ODCs (right). (b) Miter circuit used in don't-care analysis [11].

circuit. Don't-cares fall into two categories: satisfiability don't-cares (SDCs) and observability don't-cares (ODCs) [12]. SDCs occur when a particular input pattern can never occur on the inputs to a LUT. In the example shown in Fig. 3(a), the inputs $a = 0, b = 1$ will never occur. ODCs occur when the output of a LUT cannot propagate to the circuit's primary outputs. In the example, the output of $f2$ has no effect when $c = 0$.

In this work, we leverage the don't-care analysis capabilities of the ABC logic synthesis network developed at UC Berkeley [15]. ABC incorporates Boolean satisfiability (SAT)-based complete don't-care analysis that can be used to determine the don't-care minterms for a given LUT in a technology mapped FPGA circuit [11]. To find the don't-cares for a given LUT, f, ABC uses a *miter* circuit, as illustrated in Fig. 3(b). As shown, two instances of LUT f and (some of) its surrounding circuitry are created – the surrounding circuitry is shown as a shaded region in the figure. In one instance, f's output is in true form; in the other instance, f's output is inverted. The outputs of the two instances are exclusive-OR'ed with one another, with the XOR gate outputs being fed into a wide OR gate. The final OR gate produces an output logic signal $C(x)$ for a given input vector x.

For an input vector x to the miter in Fig. 3(b), one can compute a *local* input vector y to LUT f. For any such x where $C(x)$ is logic-1, y is a *care* minterm of LUT f; that is, LUT f affects the circuit outputs for input vector x. The basic approach taken in [11] is to use a fast vector-based simulation as well as SAT to find all vectors, x, where $C(x)$ evaluates to logic-1, yielding the complete care set for LUT f. This provides a general picture of the don't-care analysis approach and the reader is referred to [11] for full details. Don't-cares have recently been used for area reduction in FPGA circuits [12].

III. Glitch Power and Don't-Care Analysis

A. Glitch Power

To motivate the need for glitch reduction, we examine the amount of glitch power dissipated by 20 MCNC benchmark designs. These designs were fully compiled using Altera Quartus 10.1, targeting 65nm Stratix III devices [1]. ModelSim 6.3e was then used to perform a functional (zero-delay) and timing simulation of each circuit using 5000 random input vectors, producing two switching activity (VCD) files. The dynamic power was then computed using Quartus PowerPlay – Altera's power analysis tool. The glitch power was computed as the difference in dynamic power between the functional and timing simulations.

Circuit	% glitch	Circuit	% glitch
alu4	25.7	ex5p	41.6
apex2	29.2	frisc	10.7
apex4	30.3	misex3	25.4
bigkey	29.6	pdc	36.7
clma	24.2	s298	24.2
des	45.4	s38417	26.8
diffeq	5.8	s38584_1	11.4
dsip	29.9	seq	26.2
elliptic	12.2	spla	33.2
ex1010	35.0	tseng	17.5
		Average	26.0

TABLE II
Percentage of dynamic power from glitches.

Circuit	% inputs DC	Circuit	% inputs DC
alu4	18.4	ex5p	36.2
apex2	7.4	frisc	5.2
apex4	17.8	misex3	17.0
bigkey	3.7	pdc	37.2
clma	32.4	s298	15.3
des	0.8	s38417	10.3
diffeq	3.9	s38584_1	3.1
dsip	4.6	seq	7.6
elliptic	0.7	spla	33.8
ex1010	34.6	tseng	12.4
		Average	15.1

TABLE III
Percentage of simulated local LUT input states corresponding to don't-cares.

The results are shown in Table II. The percentage of dynamic power due to glitches ranges from 5.8% to 45.4%, with an average of 26.0%, which is similar to that reported in the academic FPGA context [8]. This makes glitches an attractive target for power reduction in commercial FPGAs. We do not believe any prior published work has analyzed glitch power in a commercial FPGA.

B. Analysis of Don't-Cares

In order to evaluate the potential for a don't-care-based glitch reduction algorithm, we analyzed every local input vector seen by each LUT in each circuit across its simulation. The percentage of such LUT input states which were don't-cares is shown in Table III. The percentages vary from 0.8% to 37.2%, with an average of 15.1%. This tells us that not only do circuits contain an abundance of don't-cares, but also that, surprising, these don't-cares are often traversed in circuit operation. In other words, a LUT's don't-care minterms are frequently "visited" under vector stimulus. The visits to such don't-care minterms may potentially lead to additional unnecessary toggles on LUT outputs. We can thus potentially reduce glitches through don't-care settings, which is the core idea of our approach.

Algorithm 1 Glitch reduction algorithm.

Input: a netlist G(V, E) with simulation vectors S
Output: a netlist with modified LUT functions
1: **for** each LUT $n \in V$ in order of priority **do**
2: {1. Compute the don't-cares of the LUT}
3: $DC = compute\_dont\_cares(n)$
4: {2. Scan the input vectors}
5: **for** $i = 0$ to $size(S_n)$ **do**
6: **if** $S_n[i] \in DC$ **then**
7: $prev \leftarrow$ previous care output
8: $next \leftarrow$ next care output
9: **if** $prev = 0$ **and** $next = 0$ **then**
10: $Votes0(S_n[i]) \leftarrow Votes0(S_n[i]) + 1$
11: **else if** $prev = 1$ **and** $next = 1$ **then**
12: $Votes1(S_n[i]) \leftarrow Votes1(S_n[i]) + 1$
13: **end if**
14: **end if**
15: **end for**
16: {3. Set the values of the don't-cares and update netlist}
17: **for** each don't-care $d \in DC$ **do**
18: **if** $Votes0(d) > Votes1(d)$ **then**
19: assign 0 as the output of d
20: **else if** $Votes1(d) > Votes0(d)$ **then**
21: assign 1 as the output of d
22: **end if**
23: **end for**
24: **end for**

IV. Algorithm

The glitch reduction algorithm (Algorithm 1) takes a placed and routed netlist as its input. We represent the netlist as a graph $G(V, E)$, where V is the set of vertices (LUTs) and E is the set of edges (routing wires). The algorithm also takes a value change dump (VCD) file containing the results of a timing simulation of the circuit. The simulation vectors are denoted as S, where the i^{th} local input vector to LUT n is denoted as $S_n[i]$. A timing simulation is needed rather than a functional one because glitches arise from delay mismatches, which will only appear under timing simulation.

The algorithm iterates through each LUT in the netlist, progressing from shallower levels to deeper ones. This order is used because glitches prevented on shallower LUTs will be prevented from propagating to deeper LUTs, thus saving more power. Within each level, the LUTs are examined in descending order of power consumption. This prioritizes the LUTs with the greatest potential savings. For each LUT, the following steps are performed:

1) Compute the don't-cares of the LUT.
2) Scan the input vectors.
3) Set the values of the don't-cares.

A. Computing the Don't-Cares for a LUT

As described above in Section II-D, we use ABC's SAT-based don't-care analysis to compute the inputs states (minterms) for the particular LUT which are don't cares (Algorithm 1, line 3). DC is the set of don't-care input states.

B. Scanning the Input Vectors

The sequence of local input vectors to the LUT (denoted S_n) is extracted from the timing simulation VCD file. These input vectors are examined in order (line 5). When an input vector $S_n[i]$ corresponding to a don't-care is reached (line 6), we look at the closest states in the past and future that correspond to *care* input vectors (lines 7-8). We use this information to decide whether this don't-care should be set to a logic-0, logic-1, or whether there is no preference. If the closest past and future cares are identical (both logic-0 or both logic-1) then the don't-care should be set to the same value. Otherwise, there is no preference. For each don't-care minterm, a count of "votes" is kept, indicating how many times in the simulation it would be beneficial to set it to a logic-0 or logic-1 (lines 9-12). This process is repeated for each input vector $S_n[i]$ in the full simulation time (lines 5-15).

Consider again the example shown in Fig. 2 and Table I. Suppose that for input $S_n[i] = 100$, the LUT output is a don't-care. This means that even though it is assigned to logic-1 in the truth table, we can assign it to logic-0 or logic-1 without affecting the functionality of the circuit. In this case, we see a glitch on f making a $0 \rightarrow 1 \rightarrow 0 \rightarrow 0$ transition as the inputs transition $000 \rightarrow 100 \rightarrow 110 \rightarrow 111$. Looking at the closest care states before and after input 100, we see that they both output a logic-0. Therefore, the algorithm votes for the output of 100 to be logic-0.

It is possible that the simulation data may include a long contiguous cluster of don't-cares. In these cases, the more desirable state could be the opposite of the one that would be chosen by this algorithm. For example, it may be beneficial to set a particular don't-care to logic-0 within a cluster of logic-0's (don't-cares) in between two logic-1's (cares) rather than attempting to set the entire cluster to logic-1. However, experimental data shows that such long clusters are uncommon. The average length of don't-care clusters in the benchmark set is 3.5. This justifies our use of the closest care input vectors.

C. Setting the Don't-Cares

When the end of the input vectors is reached, each don't-care is set to the value with more votes (unless the votes are tied, in which case nothing is done). The loop at lines 17-23 walks through each don't-care $d \in DC$ and checks whether logic-0 or 1 has a majority of votes. The netlist is updated accordingly before proceeding to the next LUT. This is critical because changing the logic function of one LUT can affect the don't-cares of other LUTs, due to incompatibility between don't-cares [12]. By ensuring that the don't-cares are computed using the most recent information, the circuit is guaranteed to remain functionally-equivalent to the original.

D. Iterative Flow

Following the modification of the circuit, the simulation results become outdated, due to the changes to the LUT functions. Therefore, we repeat the simulation using the modified circuit after performing glitch reduction on the full circuit. The algorithm is then repeated. In practice, the majority of the glitch reduction occurs within the first three iterations.

It is important to note that the loop of the iterative flow does not involve re-running placement and routing. This is vital for two main reasons. First, the results of the existing compilation will be preserved, so there is no interference with timing closure. Second, the delays within the circuit will be kept the same, thus minimizing the amount of change to the simulation vectors. This allows the algorithm to converge quickly.

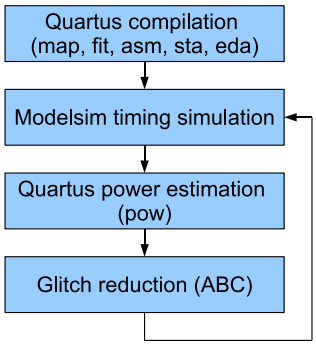


Fig. 4. Experimental flow.

The algorithm runtime is on the order of minutes for the benchmarks used. Although the iterative process employs a timing simulation, the fact that this algorithm is performed after place-and-route mitigates the issue of runtime. We envision a usage scenario in which the designer runs this algorithm as part of a final pass after timing closure has been achieved. Since no modifications are made to the circuit's timing characteristics, timing closure is preserved.

V. Experimental Study

We perform our glitch reduction algorithm on 20 MCNC benchmark circuits. The experimental methodology was chosen to include commercial CAD tools wherever possible, to evaluate the efficacy of the algorithm on real-world FPGAs. The flow is shown in Fig. 4. We perform a full compilation using Quartus II 10.1 (synthesis, placement and routing) targeting the Altera Stratix III 65nm FPGA family [1]. This is followed by a timing simulation using ModelSim SE 6.3e. For each circuit, 5000 random input vectors are applied. We use a set of custom scripts to transform the simulation netlist generated by Quartus into BLIF format, which can then be read into ABC, where the glitch reduction is performed. Combinational equivalence checking (command *cec* in ABC [13]) is used after the glitch reduction step to ensure that the functionality of the circuit remains the same. The output from ABC is used to modify the configuration bits in the simulation netlist, thus ensuring that the placement and routing remain identical. Three passes of the optimization loop are performed. Experiments show that very few changes, if any, are made after this point. The power measurements are performed using Quartus PowerPlay.

A. Results

The leftmost bars in Fig. 5(a) (vs. baseline) represent the percentage reduction in total core dynamic power after performing the glitch reduction algorithm. The average reduction is 4.0%, with a peak of 12.5%. Fig. 5(b) shows the corresponding reduction in *glitch power*. The average reduction is 13.7%, with a peak of 49.0%. Naturally, the amount of power reduction possible is based on the amount of glitching present and the number of don't-cares available. While the overall average power reductions are relatively modest, we believe they will interest FPGA vendors and power-sensitive FPGA customers, as they come at no cost to performance or area. For some circuits, over 10% power reduction can be achieved essentially for "free".

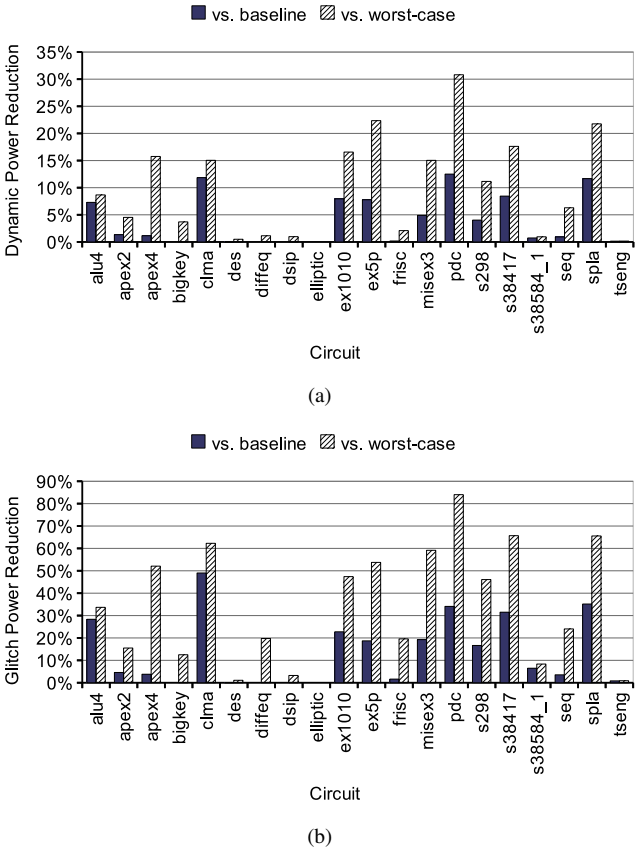
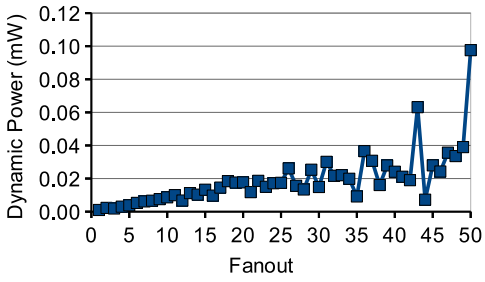


Fig. 5. (a) Dynamic power reduction vs. baseline (default) don't-care settings and worst-case settings. (b) Glitch power reduction vs. baseline (default) don't-care settings and worst-case settings.

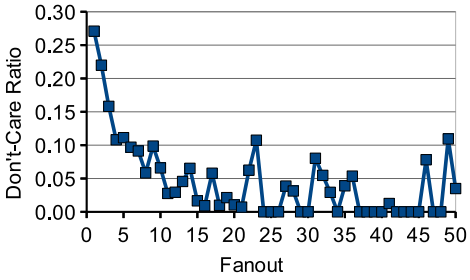
It is also interesting to look at the optimized power vs. the worst case don't-care settings possible, as illustrated by the rightmost bars in Fig. 5 (vs. worst-case). In this experiment, we set the don't-cares to the opposite of how they would normally be set by our optimization algorithm, to examine the potential worst-case glitch power arising from don't-cares. Here, we see an average total dynamic power savings of 9.8% and a peak savings of 30.8% (Fig. 5(a)). These results show that don't-care settings can potentially have a large impact on power if set to sub-optimal values.

The varied results in Fig. 5 can be correlated with the glitch power and don't-care data in Tables II and III. For instance, *des* had a high glitch power in Table II, yet we did not observe a significant power reduction for this circuit. However, in Table III, we see that it had only 0.8% of LUT inputs as don't-cares, thus reducing the number of opportunities for optimization. On the other hand, *pdc* had a high amount of glitching as well as ample don't-cares, thus allowing it to be greatly improved by the algorithm – 12.5% dynamic power reduction.

The relationship between don't-cares, power and fanout presents a challenge to the glitch reduction algorithm. Fanout is closely related to interconnect capacitance, and interconnect can represent 60% of total FPGA dynamic power, on average [14]. Fig. 6(a) shows logic signal power consumption versus fanout, averaged across all signals in all circuits. Observe that, as



(a)



(b)

Fig. 6. (a) Power per signal vs. fanout. (b) Normalized don't-cares per node vs. fanout.

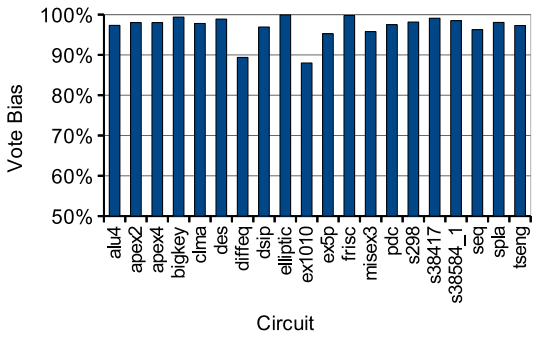


Fig. 7. Average vote bias.

expected, average signal power increases with fanout, due to the increase in capacitance. We also examined, for each signal, the fraction of minterms in its driving LUT that were don't-cares, and averaged this across all signals of a given fanout in all circuits. The results are shown in Fig. 6(b). While the results are "noisy" for high fanout (due to a small sample size for such fanouts), we see that, in general, high fanout signals have fewer don't-cares in their driving LUTs than low fanout signals. The rationale for this is that high fanout signals are more likely to be used by at least one of their fanouts, decreasing ODCs for such signals. Essentially, we have two competing trends in that it is desirable to reduce the power of high fanout signals (as they consume significant power), yet such signals exhibit fewer don't-care opportunities.

We also examined the bias of votes cast on each don't-care minterm in each LUT in each circuit. The average results are shown in Fig. 7. The bias is defined as the percentage of votes that were cast for the more popular setting, whether logic-0 or logic-1. As shown in the figure, the bias value tends to be in the 80-100% range, indicating that there usually exists a highly preferable setting for a particular don't-care minterm in a LUT. This is an important observation because it indicates that our don't-care settings are providing a benefit most of the time (as opposed to the case of a bias around 50%, which would imply that selecting either logic-0 or logic-1 for the don't-care minterm is equally good). These observations suggest that one can pick don't-care logic values with a high degree of confidence.

VI. Conclusion and Future Work

In this paper, we presented an analysis of glitch power in FPGAs and a method for glitch reduction using don't-cares in logic synthesis. We showed that glitch power is a significant portion of total power, and that there exist ample opportunities for don't-care-based optimizations. A novel glitch reduction technique was presented that sets don't-cares in FPGA configuration bits in order to avoid glitch transitions. This method is performed after placement and routing, and has no effect on circuit area or performance. The algorithm was evaluated with a commercial $65n$m FPGA architecture using a commercial tool flow. The algorithm achieved an average total dynamic power reduction of 4.0%, with a peak reduction of 12.5%; glitch power was reduced by up to 49.0%, and 13.7% on average. Future work will involve integrating the algorithm into a fully power-aware FPGA CAD flow, and investigating whether other stages of the CAD flow could improve its effectiveness. For instance, the synthesis stage could be modified to create more opportunities for this optimization. Also, runtime could be reduced by integrating the algorithm with an incremental timing simulation.

References

[1] Altera. Stratix III Device Handbook. http://www.altera.com/literature/lit-stx3.jsp.

[2] Altera. Stratix V Device Handbook. http://www.altera.com/literature/lit-stratix-v.jsp.

[3] L. Cheng, D. Chen, and M. Wong. GlitchMap: An FPGA technology mapper for low power considering glitches. In *ACM/IEEE DAC*, pages 318 –323, 2007.

[4] Tomasz S. Czajkowski and Stephen D. Brown. Using negative edge triggered FFs to reduce glitching power in FPGA circuits. In *ACM/IEEE DAC*, pages 324–329, 2007.

[5] Q. Dinh, D. Chen, and M. Wong. A routing approach to reduce glitches in low power FPGAs. In *ACM ISPD*, pages 99–106, 2009.

[6] R. Fischer, K. Buchenrieder, and U. Nageldinger. Reducing the power consumption of FPGAs through retiming. In *IEEE Engineering of Computer-Based Systems*, pages 89 – 94, 2005.

[7] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE TCAD*, 26(2):203 –215, 2007.

[8] J. Lamoureux, G. Lemieux, and S. Wilton. GlitchLess: Dynamic power minimization in FPGAs through edge alignment and glitch filtering. *IEEE TVLSI*, 16(11):1521 –1534, Nov. 2008.

[9] H. Lim, K. Lee, Y. Cho, and N. Chang. Flip-flop insertion with shifted-phase clocks for FPGA power reduction. In *IEEE/ACM ICCAD*, pages 335–342, 2005.

[10] B. Lin and S. Devadas. Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams. *IEEE TCAD*, 14(8):974 –985, Aug 1995.

[11] A. Mishchenko and R. Brayton. SAT-based complete don't-care computation for network optimization. In *ACM/IEEE DATE*, pages 412–417, 2005.

[12] A. Mishchenko, R. Brayton, J. Jiang, and S. Jang. Scalable don't-care-based logic optimization and resynthesis. In *ACM FPGA*, pages 151–160, 2009.

[13] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to combinational equivalence checking. In *IEEE/ACM ICCAD*, pages 836–843, 2006.

[14] L. Shang, A. Kaviani, and K. Bathala. Dynamic power consumption in Virtex-II FPGA family. In *ACM FPGA*, pages 157–164, 2002.

[15] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, Release 00406. http://www.eecs.berkeley.edu/~alanmi/abc/.

[16] S. Wilton, S. Ang, and W. Luk. The impact of pipelining on energy per operation in field-programmable gate arrays. In *Proc. Intl. Conf. on FPL*, pages 719–728, 2004.

[17] Xilinx. 7 Series FPGAs Overview. http://www.xilinx.com/support/documentation/7_series.htm.