Control Logic Restructuring for Area Optimization

Alan Mishchenko Robert Brayton Walter Lau Neto Pierre-Emmanuel Gaillardon Luca Amarù

University of California, Berkeley University of Utah Synopsys Inc

{alanmi, brayton}@eecs.berkeley.edu {walter.launeto, pierre-emmanuel.gaillardon}@utah.edu lamaru@synopsys.com

Abstract

When synthesizing hardware designs, the frontend of a synthesis tool often elaborates control logic as multiplexers and selectors, which are next bit-blasted into simple gates and processed by a logic synthesis flow. This approach may lead to suboptimal results or take a prohibitive runtime to synthesize optimally. The present paper focuses on a specialized logic synthesis method applied to a word-level view of the control logic after elaboration before bit-blasting, or to the control logic extracted from a netlist after bit-blasting. The method detects clock-enables and other types of shared logic, resulting in substantially reduced area and delay, at a negligible runtime cost.

1. Introduction

In a typical synthesis scenario, a hardware design in Verilog or VHDL is entered into the tool via a frontend, which elaborates it into word-level operations and bit-blasts them into elementary logic gates. The gate-level netlist goes into a logic synthesis engine, which applies standard optimization techniques. In the general case, synthesis works well but often fails to find good solutions for specialized logic. Analyzing the problematic test cases helps logic synthesis researchers and CAD tool developers isolate design/elaboration patterns and address them by dedicated transforms.

One class of such patterns concerns next-state logic cones of word-level registers present in typical hardware designs. In particular, when an N-bit register stores data from multiple places in the design, a typical next-state logic cone selects among the data sources. For example, in designing datapaths, the data saved in a register may come from a multiplier, an adder, a bit-shifter, etc. The data source is determined by the output of the instruction decoders, data comparators, etc., serving as inputs to the next-state logic.

We observed that the default elaboration often leads to suboptimal synthesis results or long synthesis runtimes.

This paper proposes a new way of restructuring word-level next-state logic cones, isolating the shared control logic and using the smallest possible multiplexer needed to select among the data inputs using the shared controls.

Another approach to multiplexer transforms [3] focuses on resource sharing rather than control logic minimization.

The rest of this paper is organized as follows. Section 2 describes the background. Section 3 shows a motivating example. Section 4 describes the algorithm. Section 5 reports experimental results. Section 6 concludes the paper and outlines future work.

2. Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. In this paper, we consider only combinational Boolean networks.

A node n has zero or more *fanins,* i.e., nodes that are driving n, and zero or more *fanouts*, i.e., nodes driven by n. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of network nodes, which deliver the functionality of the network to its environment. A *fanin* (*fanout*) *cone* of node n is a subset of all the network nodes, reachable through the fanin (fanout) edges of the node.

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these two-input ANDs are added to the AIG manager in topological order. The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations of an AIG is to reduce both area and delay.

Structural hashing of AIGs ensures that all constants are propagated and, for each pair of nodes, there is at most one two-input AND with them as fanins (up to a permutation). Structural hashing is performed by hash-table lookups when AND nodes are created and added to an AIG manager. Structural hashing can be applied on-the-fly during AIG construction, which reduces the AIG size.

The concepts of *area* and the number of AIG nodes are used interchangeably in this paper. The concepts of delay, depth, and logic level are also used interchangeably.

A *local* function of an AIG node n, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in n and expressed in terms of the leaves, x, of a cut of n. The *global function* of an AIG node is its function expressed in terms of the PIs of the AIG.

AIGs can represent both local and global functions. Because of low memory usage, speed of manipulation, and scalability, AIGs have recently emerged as a widely-used data-structure in logic synthesis and formal verification.

Additional information can be found in the following publications on AIGs [4] and AIG-based synthesis [5][6].

3. Motivation

This section shows a motivating example, followed by the outline of the proposed algorithm in the next section.

Hardware designs in RTL Verilog [10] represent control logic using *always*-statements. Figure 1 below shows a small design containing one word-level (32-bit) register *out* whose next-state function is given by an *always*-statement. Three control variables, *c0*, *c1*, and *c2*, determine what data is written into the register on the rising edge of the clock signal *clk*. For example, if *c0* is 0 and *c1* is 1, 32-bit data *a* is written into *out*. If for some inputs, the register *rvalue* is not defined, it is assumed to be the same as in the previous cycle. For example, if *c0* is 0 and *c1* is 0, *out* is not defined.

```
module example ( clk, c0, c1, c2, a, b, out );

input clk;
input c0, c1;
input [1:0] c2;
input [31:0] a, b;
output [31:0] out;

reg [31:0] out;
always @ (posedge clk)
  if (c0 == 1'b0)
     begin
       if (c1 == 1'b1)
          out <= a;
     end
  else // c0 == 1'b1
     begin
       case (c2)
         2'b00: ;
         2'b01: out <= a;
         2'b10: out <= b;
         2'b11: ;
       endcase
    end

endmodule
```

Figure 1: A small design block used for illustration.

When the Verilog in Figure 1 is elaborated by a synthesis tool, a circuit in Figure 2 is typically generated. This circuit is suboptimal, as shown below. In the case of small *always*-statements, such as the one in Figure 1, suboptimality is overcome by performing logic synthesis. However, in the case of larger *always*-statements, generic logic synthesis does not lead to an optimal solution, or requires prohibitive runtime. Thus, specialized methods are of great interest.

The method proposed in this paper takes a control logic cone after elaboration and performs restructuring, resulting in a near-optimal circuit structure. The restructuring isolates control logic signals shared across all bits of the register. One of such signals is clock-enable, which is commonly extracted by synthesis tools. Other types of shared logic are not commonly extracted by the tools, but can be extracted using the proposed method.

3.1 Clock-enable extraction

If the hardware primitive used to implement the 32-bit register has a clock-enable (CE) pin, the next-state logic of the register *out* in Figure 1 can be simplified by extracting the condition when the register value changes. This condition can be used to drive the CE pin, as shown in Figure 3.

3.2 Optimizing next-state logic using CE

The next-state logic after the CE extraction shown in Figure 3 can be further simplified by noticing that, when the value of the CE is zero, the value of the next-state function does not matter. In this case, the logic shown in Figure 3 can be transformed into the structure shown in Figure 4.

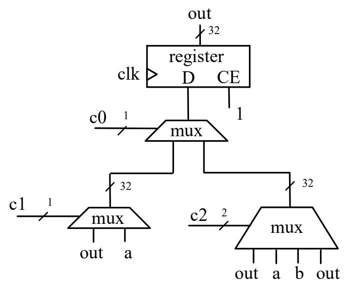

Figure 2: The circuit derived by elaborating the design.

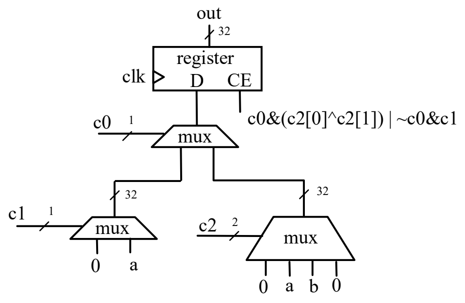

Figure 3: The circuit after the clock-enable extraction.

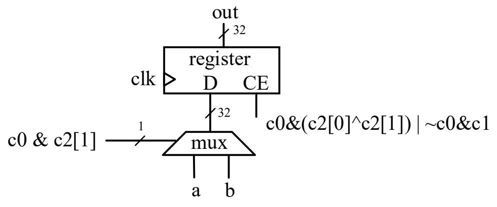

Figure 4: The circuit after the proposed optimization.

If 4-input LUTs are used to realize logic functions, the logic cone in Figure 2 requires 128 LUTs. This is because each 2:1 MUX takes one LUT, while each 4:1 MUX takes two LUTs in each of the 32 bit-level cones. Meanwhile, the simplified logic cone in Figure 4 requires only 33 LUTs (32 LUTs for the next-state functions plus an extra LUT to generate the shared CE signal). This is an almost 4x area reduction compared to the naïve solution. If the bit-level flip-flops do not have the CE pin, the logic cone in Figure 4 would require 64 LUTs, which is still a 2x area reduction.

4. Algorithm

The input to the algorithm is a multi-output bit-level combinational logic cone of the next-state functions of a word-level data register. The logic cone is represented as an AIG whose inputs are primary inputs and flop outputs of the design to which the data register belongs.

4.1 Detecting data and control variables

The algorithm begins by analyzing the multi-output logic cone and detecting *control* and *data variables*. Both of these variables can be primary inputs or internal nodes of the AIG. The main difference between control and data is that the data variables are written into the flops when control variables have specific values.

A practical way to distinguish the control and data variables is to observe that control variables are shared among the logic cones of the bit-level next-state functions, while the data variables are unique for each function.

For example, consider logic cones shown for in Figure 2 derived by elaborating the design in Figure 1. All of the 32 bit-level cones depend on the same control variables, c_0, c_1, and c_2. However, each bit-level data variable appears only in one bit-level logic cone, in particular, a_i and b_i appear only in the logic cone of i-th flop.

In the proposed approach, the detection of *control* and *data variables* in the bit-level logic cones is performed as follows:

- For each bit-level input of the N-bit word-level register
  - Find the maximum fanout-free cone (MFFC) of the input, that is, those AIG nodes that would be removed if the input were removed.
  - Mark the AIG nodes connected to the MFFC nodes (these can be primary inputs or internal AIG nodes).
- Consider AIG nodes marked by the above procedure.
  - If a node has been marked once, it is a data variable.
  - If a node has been marked N times, it is a control variable, shared across N logic cones.
  - If a node has been marked more than once and less than N times, abort the computation because control logic cannot be extracted from all logic cones.

Please note that the computation may be aborted for several reasons: (a) some bit-level functions of the word-level register indeed have different functions and thus logic sharing cannot be extracted by the proposed method, (b) some form of logic synthesis has been applied to the next-state logic cones after elaboration, resulting in the loss of structural uniformity. In both cases, it may be possible to address the situation: for example, split the bit-level function into several groups with uniform bit-level logic within each group. In the interest of focused presentation, we do not consider such complications.

4.2 Transforming one bit-level logic cone

Next, we discuss application of the algorithm to a single-output next-state logic function, $F(C, X)$, represented as an AIG logic cone in terms of two groups of variables: control variables C and data variables $X = (x_0, x_1, ..., x_{n-1})$, which can be either primary inputs or internal nodes of the AIG, detected as discussed in Section 4.1.

The main idea is to decompose this function as follows:

$$F(C, X) = x_0 \,\&\, f_0(C) + x_1 \,\&\, f_1(C) + \ldots + x_{n-1} \,\&\, f_{n-1}(C) \quad (1)$$

where functions $f_0(C)$, $f_1(C)$, … $f_{n-1}(C)$ do not depend on data variables, X, and are mutually disjoint, that is, $f_i(C)$ & $f_j(C) = 0$ for all i and j such that $i \neq j$.

Assuming that this condition holds, the algorithm tries to find functions $f_i(C)$ by setting corresponding values to the X variables. For example, by substituting into $F(C, X)$ value 0 for all X variables, except for one variable x_i set to value 1, we can determine function $f_i(C)$. When all f-functions are found, equivalence checking is performed to confirm that the composition of these functions with the corresponding data variables is equivalent to the given function $F(C, X)$. If the check fails, the algorithm quits, indicating failure to perform logic restructuring. It was observed that failures happen rarely in practice, most often due to functions $f_0(C)$, $f_1(C)$, … $f_{n-1}(C)$ not being mutually disjoint. This situation can also be addressed, but the detailed discussion is beyond the scope of this paper.

4.3 Extracting clock enable

This subsection shows that detecting of the clock-enable condition of a bit-level flop can be performed as a by-product of applying the proposed algorithm.

Indeed, if the function $F(C, X)$ is transformed into the expansion (1), as shown in Section 4.2, the data variables, X, may contain the flop-output variable x_{fo}. In this case, the corresponding control function $F_{fo}(C)$ represents the clock-enable condition that can be used to drive the clock-enable pin of the bit-level flops. Other functions of the expansion (1) can now be minimized using $F_{fo}(C)$ as the care set. In practice, this may leads to a substantial reduction in the size of AIG for $F(C, X)$. We refer the interested reader to the following publications discussing minimization of logic functions using don't-cares [7][8].

4.4 Transforming multiple logic cones

In previous sections, we discussed transformation by the above algorithm of a single-output logic function, $F(C, X)$.

Now, we recall that this function is one bit-level next-state function of an N-bit register. We can apply the same transformation to each next-state function and, if all of them are successfully decomposed using the same control logic functions $f_0(C)$, $f_1(C)$, … $f_{n-1}(C)$ and different data-bits, the shared logic in the N-bit next-state cone has been extracted.

However, in practice, we can avoid transforming each bit-level function and instead assume that the same transformed structure holds for all of them. Thus, we duplicate the transformed cone for each other bit, while keeping the same control functions and replacing data variables accordingly. As a result, the shared control functions are automatically extracted, while the resulting logic needs to be checked for equivalence against the original multi-output logic cones before the transformation. If the equivalence holds, the transformation is correct; otherwise, it is rejected, and the original multi-output logic cone is used.

It can be noted that the transformed multi-output logic cone includes the shared control logic (one per all bit-level flops) and individual MUXes (one MUX per each bit-level flop). The shared control logic selects the input for each flop using the smallest MUX needed to choose among the corresponding data variables.

The resulting area reduction achieved in practice by the proposed algorithm ranges from 10% to 10x. Several factors determine the efficiency, such as:

- The number of bits in the word-level register (the more bits, the more beneficial is sharing of the control logic).
- The complexity of the shared control conditions (the more complex the shared logic, the more savings).
- The number of data sources of the word-level register (the fewer are data sources, the smaller are the MUXes selecting data into each bit-level flop, relative to the size of the shared control logic, and the more savings).

5. Experiments

The proposed algorithm is implemented in ABC [1][2] as command *&reshape* and tested on the example in Figure 1 and logic cones extracted from the hardware design blocks publicly available in the OpenCores design repository [9].

Two experiments have been performed. The first experiment in Section 5.1 lists detailed optimization results for individual word-level registers. The second experiment in Section 5.2 shows the cumulative effect of the optimization applied to the corresponding design blocks.

5.1 Optimizing logic of word-level registers

For this experiment, we selected several word-level registers from different designs and applied the proposed optimization. The results are reported in Table 1.

The table reports the following statistics for each word-level register: the block name ("Design block") and the register name ("Register"), the bit-width ("Bits"), the number of control variables ("CVars"), and data variables ("DVars") detected when considering bit-level functions (as discussed in Section 4.1), the original AIG size before and after AIG rewriting using command *&dc2* ("Base" and "BaseOpt"), the transformed AIG size before and after AIG rewriting using *&dc2* ("Trans" and "TransOpt").

Table 1 shows that in some cases the AIG size is not reduced, and in some cases it is reduced substantially.

The runtime of the proposed method is not reported because it was negligible (less than 5%) compared to that of command &dc2, which takes a few seconds for the largest logic cone reported in Table 1.

5.2 Optimizing the design blocks

For this experiment, we selected several design blocks containing word-level registers whose next-state logic cones could benefit from the proposed optimization. Table 2 reports the following information: the design block name ("Design block"), the number of primary inputs ("PI"), primary outputs ("PO"), word-level registers ("FF"), bit-level flip-flops ("FF"), and internal AIG nodes after elaboration ("Base"). The optimization baseline is given by the number of AND nodes after running command *&dc2* once ("BaseOpt") and 10 times ("BaseOpt10").

The method proposed in this paper is applied to the AIG after elaboration, resulting in a transformed AIG ("Trans") followed by the same logic synthesis script applied once ("TransOpt") and 10 times ("TransOpt10").

As mentioned in Section 5.1, the runtime of the proposed method is not reported because in all cases it was a small fraction (less than 5%) of the runtime of logic synthesis.

Comparing AIG sizes reported in columns "BaseOpt" and "BaseOpt10" with those in columns "TransOpt" and "TransOpt10", shows that, for large design blocks, the proposed optimization substantially reduces the AIG size. On the other hand, for smaller block, iterating logic synthesis reduces the difference.

It should be noted that Table 1 and Table 2 list the results obtained without using don't-cares derived from extracting CEs. The results may be better when this feature is enabled.

6. Conclusions

This paper describes a fast way of detecting shared logic in the next-state logic cones of multi-bit data registers frequently found in hardware designs. The logic cones may be hard to optimize using traditional synthesis because the front-end of the synthesis tools automatically transforms next-state logic cones into bit-level gates, making it impossible or time-consuming to extract the shared logic.

The proposed algorithm performs quick structural analysis to detect control and data variables, followed by restructuring of the multi-bit next-state cones to expose the shared control logic functions. In most cases, the shared clock enable signal is detected as a by-product of the proposed transform. The clock enable, if detected, can be used as the source don't-cares to further reduce the size of shared control logic cones.

Future work may include: (a) addressing the case when a word-level register has to be divided into several parts to expose logic sharing specific to each part, (b) improving the quality of don't-care-based optimization applied to shared control logic functions, (c) extending the algorithm to work for other types of shared logic.

Acknowledgements

This work has been supported in part by the NSA grant "Novel methods for synthesis and verification in cryptanalytic applications" and industrial sponsors: AMD, Siemens, and Synopsys.

REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[3] C. Yu, M. J. Ciesielski, M. Choudhury, and A. Sullivan, "DAG-aware logic synthesis of datapaths", *Proc. DAC'16*.

[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification", *IEEE TCAD*, Vol. 21(12), Dec. 2002, pp. 1377-1394.

[5] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[6] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure", *Proc. IWLS '06*, pp. 15-22.

[7] Y. Miyasaka, A. Mishchenko, J. Wawrzynek, and N. J. Fraser, "Synthesizing practical Boolean functions using truth tables", Submitted to *IWLS'22*.

[8] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "Simulation-based resubstitution", *Proc. IWLS'20*.

[9] Free and Open Source Gateware IP Cores. https://opencores.org/

[10] IEEE Standard for Verilog Register Transfer Level Synthesis (1364-2002).

Table 1. Transforming individual word-level registers by restructuring control logic.

Design block	Register	Statistics			Baseline		Transformed	
		Bits	CVars	DVars	Base	BaseOpt	Trans	TransOpt
example	out	32	4	3	480	480	172	167
i2c	wb_dat_o	8	3	7	146	137	109	109
syntax_dec	mvd_reg	8	7	2	204	36	55	30
iter_pred	inter_pred_reg_ctrl	8	17	5	37604	1035	481	189

Table 2. Optimizing design blocks by restructuring control logic.

Design block	Design				Baseline			Transformed		
	PI	PO	Reg	FF	Base	BaseOpt	BaseOpt10	Trans	TransOpt	TransOpt10
example	69	32	1	32	480	480	480	172	167	167
i2c	19	14	11	129	1601	890	815	1328	862	798
syntax_dec	74	114	21	114	7874	925	912	2080	902	879
iter_pred	55	104	169	1352	391536	123053	94441	29772	22267	20984