Automatic Addition of Reset in Asynchronous Sequential Control Circuits

Vikas S. Vij, Kenneth S. Stevens
University of Utah

***Abstract*—Asynchronous finite state machines (AFSMs) usually require initialization to place them in a desired starting state. This normally occurs by toggling a reset signal upon power-up. This paper presents an algorithm to automatically generate power-up reset circuitry thus adding reset to an AFSM after technology mapping. This approach is independent of design methodology since it is applied to a gate netlist. The algorithm ensures all combinational cycles and primary outputs in the circuit are initialized. Options exist in reset generation to minimize the power or performance impact on the AFSM. Results are reported for applying this algorithm to designs of varying size and complexity.**

I. Introduction

The behavior of a sequential circuit cannot be determined solely by its primary inputs (PIs). Sequential logic can behave differently for identical input sequences based on the starting state. Thus it is essential to initialize sequential logic to a specific state to ensure correct behavior.

The state based behavior of sequential circuits is implemented with *state variables*. State variables are created with feedback cycles in the boolean logic descriptions of sequential asynchronous finite state machines (AFSM). These feedback cycles are explicitly maintained in the circuit realization when the design is technology mapped to static logic gates. Other logic families, such as dynamic logic, can be used to implement AFSMs which change how state variables are implemented. This work applies to designs mapped to static libraries, since they are the most commonly used logic family.

Initialization is implemented with a reset signal that is asserted upon power up. This is usually a one-time event, but can also dynamically occur during operation to reset a sequential circuit back to its starting state. This paper addresses the former case of power-up reset.

Reset can have a significant impact on asynchronous logic design in several ways. The implementation of the reset logic has a direct influence on the power, performance, and area of a sequential circuit. Hence optimizing reset for power or performance can improve the design. Second, it is possible to change the hazard properties of an AFSM through the addition of reset. Additionally, if not fully automated, reset poses a significant manual effort in the synthesis and characterization of asynchronous circuits.

The addition of reset to an AFSM can be performed at different stages of a design flow. Firstly, it can be added in the specification of the design and implemented during synthesis. Secondly, it can be added at the technology mapping phase of synthesis. Lastly, the addition of the reset signal can be performed post technology mapping phase. We have chosen to perform reset at the latest stage in the design methodology because it allows the reset logic generation to be independent of the design or synthesis method used. Thus the method and algorithms presented here can be employed for circuits designed by hand, or from synthesis tools such as 3D, Petrify or Minimalist [1], [2], [3], [4].

The major contributions of this paper are as follows. It generates an AFSM with reset logic resulting in an improvement in power, area, and performance over the reset logic generated by other algorithms. This is primarily achieved with a three step heuristic based on logical effort [5] to optimize the circuit either for performance or power. A relationship between reset and topological cycles in a circuit is shown when a design is exclusively implemented with static logic gates. The algorithm is agnostic to how the circuit was implemented and technology mapped, so it can be used with any of the synthesis engines as well as with hand designed circuits. For the first time reset can now become part of any AFSM design automation flow.

II. Background

Two significant holes currently exist in the CAD tools used for synthesis of sequential asynchronous circuit designs: technology mapping and reset generation. Both of these are interesting and related problems, as technology mapping can introduce hazards [6], and reset is dependent on the technology mapped circuit. Without automating these tasks, synthesis and characterization of asynchronous circuits necessitates manual intervention.

Many tools and algorithms exist for the synthesis of AFSMs. The only one that includes integrated reset support is Petrify [3]. In Petrify the reset logic is performed post synthesis and is not technology mapped, requiring a final manual step to create a circuit. This manual step is being addressed by a tool named Petreset as an academic project. It analyzes the synthesis results and the design specification. Through simulation Petreset determines which gate modifications can be performed to reset the design. The restriction of this tool is that it only applies to designs which are synthesized with Petrify; hence its not independent of design methodology. This work has not yet been published.

Asynchronous finite state machine synthesis algorithms can theoretically be modified to automatically generate reset behavior jointly with the synthesis. However, we are not aware of any such work reported in the literature. We are also not aware of any published work that presents an independent algorithm to add reset for AFSMs to post technology mapped designs.

III. Algorithm

An algorithm is described that automatically synthesizes reset logic for sequential AFSMs regardless of the specification style or method used to generate the circuit. The inputs to the algorithm include (a) the sequential circuit technology mapped to single output static gates, (b) the boolean behavior of static gates available in the cell library, (c) boolean logic levels for all signals in the reset state. Two additional inputs may optionally

978-1-4799-0524-9/13/$31.00 ©2013 IEEE

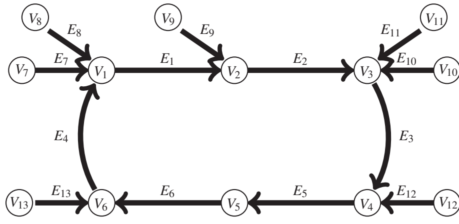


Fig. 1: A Cyclic Directed Graph Example

be included: a set of performance critical paths, and a list of primary inputs that remain undefined upon application of system reset. No design specification information is required.

The algorithm presented here is based on the observations that: (a) State variables will be implemented in a circuit that use static logic with feedback. (b) Feedback creates cycles in the circuit. (c) Cycles and undefined inputs are the only sources of undefined signals in a sequential circuit. (d) Any gate in a cycle can be used to reset the entire cycle.

This algorithm focuses on identifying combinational cycles in a circuit which need to be explicitly reset, and then selecting an optimal location in the cycle to add reset. The cost of each solution is based on heuristics that employ logical effort to estimate performance and energy costs. The determination of which cycles need to be reset is performed by simulating the design and determining which nodes remain undefined. Finding a reset configuration is not necessarily simple. Circuit cycles may interact. By resetting one cycle, the other cycles that it interacts with may automatically be reset.

The algorithm consists of two main sections. The first section identifies and resets the *cycles* and the second one does the same for *paths*. Multiple solutions are generated by adding reset signal to each gate of a cycle (path). Each solution is compared against the others to obtain the least cost solution using optimization heuristics based on logical effort.

A. Generate cycles to reset when PI's are defined

A circuit is represented as a directed graph G where G is the pair (V,E). V is a finite set of vertices v, representing single output combinational gates, primary inputs (*PI*), and primary outputs (*PO*) of a circuit. E is a set of edges e mapping $V \times V$ where e is an ordered pair (v_i, v_j) where v_i is the vertex output and v_j is an input to a vertex. P is a set of paths p where $p \in P$ is defined as an ordered sequence of vertices $< v_i, ..., v_j >$ were $\forall v_k \in p$ no vertex $v_k \in p$ is repeated, $v_k \in V$, and where there is an edge $e_k \in E$ between each adjacent vertex in path p. We also represent path p as $V_i \xrightarrow{p} V_j$. C is a set of cycles c where $c = p_i \in P$ and there exists an edge e_j that maps between the first and last element of path p_i.

Each path and cycle has two associated edge sets, internal edges E_{Int} and external edges E_{Ext}. E_{Int} is the set of edges between each vertex in a path or cycle, and E_{Ext} is the set of edges $e_i : (v_i, v_j)$ where $v_j \in p \wedge v_j \notin E_{Int}$. Note that external edges include fan-in but not fan-out connectivity.

Fig. 1 shows an example cycle consisting of the path $< v_1, v_2, v_3, v_4, v_5, v_6 >$. The internal edge set E_{Int} equals $\{e_1, e_2, e_3, e_4, e_5, e_6\}$ and the external edge set E_{Ext} in the example is $\{e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}\}$.

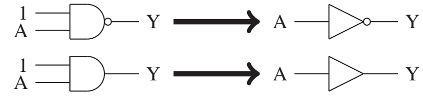


Fig. 2: Gate conversion example for Lemma 1

Each vertex $v_i \in V$ is assigned a value in the set $\{0, 1, x\}$. The value of each edge e_i is derived from the value of vertex v_i where $e_i : (v_i, v_j)$. We use the convention that a vertex (and its associated fanout edges) is *defined* when it has a boolean value of 0 or 1, otherwise it said to be is *undefined*, and is assigned the value x. Since gates (vertices) are single output, the state of all nets in the system are defined once all vertices are defined. One required input to the algorithm is the boolean logic level for all vertices in the reset state.

Definition 1: An input of a static single output gate is said to have a controlling value *if it uniquely determines the output of the gate independent of other gate inputs.*

If an input to a gate does not uniquely determine the output of the gate, it is a *non-controlling value*. If all inputs are non-controlling, then a subset of the inputs must be defined to define the output.

Axiom 1: The output of a static combinational gate is defined if all the gate inputs are defined

Lemma 1: For input set I, the output of a single output static combinational gate will be uniquely controlled by input $i_i \in I$ when all other gate inputs $i_j \in I$ are assigned to non-controlling values and the output remains undefined.

Proof: This holds due to Axiom 1. Since only one gate input is undefined, once that signal becomes defined all gate inputs are defined and the output must switch to a known value of 0 or 1 based on the combinational function. ■

Lemma 1 allows any complex static gate to be represented as a simple inverter or buffer based on the value of input i_i if, when all other gate inputs are defined, the output is still undefined. Fig. 2 shows examples of this representation. The NAND gate acts as a simple inverter when i_i is signal A since all other signals are at a high voltage. Similarly the AND gate can be modeled as a buffer.

Lemma 2: If all edges in E_{Ext} of a path (cycle) are set to logic 0 or 1, then the path (cycle) can be represented as a path (ring) consisting of inverters or buffers.

Proof: Follows Lemma 1. ■

Theorem 3: If all the edges in E_{Ext} for a cycle are defined then all the signals in E_{Int} are either defined or undefined.

Proof: Assume that the set of external inputs E_{Ext} for cycle c are set such that none of the vertices in c are defined. In this case, all internal edges E_{Int} of the cycle will be undefined. Assume the case above where a single edge $e_i \in E_{Ext}$ is modified such that $e_i = (v_i, v_j)$ is controlling or the value of vertex v_j becomes defined. This results in edge $e_j \in E_{Int}$ becoming defined. According to Axiom 1, this results in the vertex (gate) v_k becoming defined where $e_j = (v_j, v_k)$. This continues around the ring until all vertices (gates) become defined to a boolean value. ■

Theorem 4: If the set vertices in a cycle c_0 is a proper subset of the vertices in another cycle c_1, then the cycle c_0

contained in the bigger cycle c_1 must be reset to reset both the cycles.

Proof: Assume all edges in both the rings are undefined. Let V_0 and V_1 be the set of vertices in cycles c_0 and c_1, E_{Int_0}, E_{Ext_0} and E_{Int_1}, E_{Ext_1} the internal and external edges. The smaller ring is the ring where $V_i = V_0 \cap V_1$. This results in a condition where Theorem 3 does not hold since some vertex v_k will have $e_i = (v_i, v_k) \in E_{Int_0}, E_{Ext_1}$ and $e_j = (v_j, v_k) \in E_{Int_1}, E_{Ext_0}$ which are both undefined. Assume c_0 is the smaller cycle, and e_j is the undefined external input to vertex v_k. If we assume that e_j is defined, cycle c_0 can be reset, which will result in edge e_i becoming defined. This results in all external edges in E_{Ext_1} becoming defined, so that Theorem 3 can hold on the larger cycle. Since c_0 is a proper subset of $c_1, \exists e_l = (v_i, v_l)$ where $v_l \in V_1 \wedge v_l \notin V_0$, so the larger cycle c_1 will automatically become reset from the smaller cycle. ■

Theorem 4 allows the number of vertices (gates) that require reset to be smaller than the number of cycles in a sequential circuits that are undefined without reset. Also note that sequential circuits may have many cycles that overlap each other in various ways, not just as proper subsets. Theorem 4 may also be extended to reset interacting cycles that are not non-proper subsets. However, the code developed here only applies optimization of multiple cycles according to this theorem, and thus may not generate the solution with the fewest number of reset vertices (gates). Such an extension is left for related work. Further, due to Theorem 4, this algorithm generates reset logic for cycles based on the smallest vector set cardinality first. This ensures that larger concentric cycles will automatically be reset by their smaller cycles.

B. Generate paths to reset when PIs are undefined

This section removes the initial condition which requires all the PIs to be defined during cycle reset generation. This was necessary to ensure that all signals in E_{Ext} are defined. The netlist generated in the previous section is used, since it guarantees all the cycles in the circuit are defined iff all the PIs are defined. The reset problem now becomes a path based rather than a cycle based problem.

Lemma 5: For output o and input set I of a single output static combinational gate, if o is undefined then at least one of the inputs in $i \in I$ is undefined.

Proof: Applying transposition to Axiom 1. ■

Definition 2: An undefined path is a path were $\forall e_i \in E_{Int}$, the value of e_i is undefined.

Lemma 6: Consider there are no undefined edges in a circuit when all the PIs are defined. If a PI is marked undefined, and this results in a set of POs of the circuit being undefined, then there exists at least one undefined path from the PI to each undefined POs.

Proof: The input netlist of this section considers that if all the PIs are defined then all the wires in a circuit including the POs are defined. Hence if a PI is undefined which results in a subset of the POs being undefined, then there must be an undefined path from the PI to each undefined PO. ■

The path may be represented as a set of inverters, and resetting any vertex will result in all downstream vertices becoming defined. This can be shown using a similar approach as was done for cycles. Therefore, to reset a path, any vertex in the path may be reset.

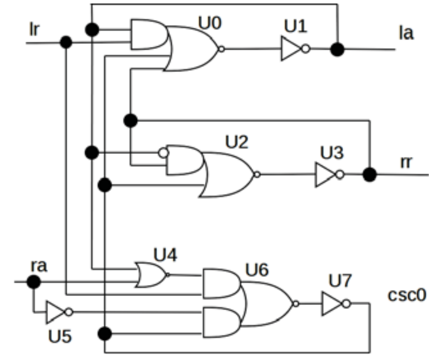


Fig. 3: Example 1 circuit implementation before reset

C. Gate Modifications for Reset Insertion

Each gate in a path (cycle) is a candidate for reset insertion. Therefore every gate is evaluated for the cost and potential of adding reset to that gate. The reset signal must be inserted as a controlling value to the gate, and the resultant gate must be a member of the static gate library employed in the design.

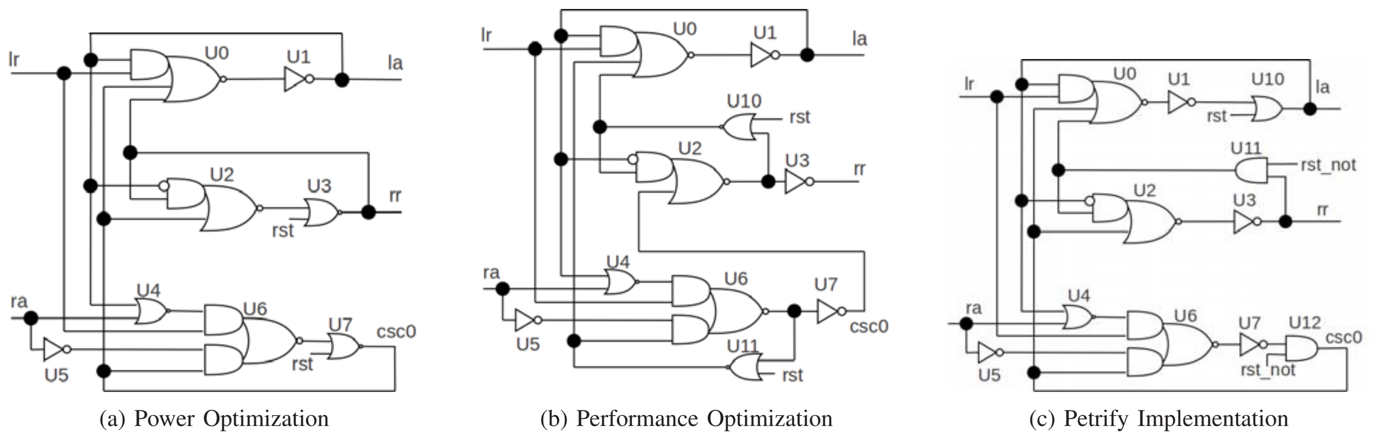
Three separate transformation cases are employed to insert a reset signal into a cycle, but only the first two transformation cases can be applied to paths. These are selected based on the type of optimization being performed and the gate type.

Case1: If the gate is an inverter (buffer), it will be converted into a NAND or NOR (AND or OR) gate depending on the required value of the output of the gate after reset. The asserted reset signal will become the controlling value for the gate.

Case2: This is a generalized condition for Case1 that will add reset to any static single output gate. The input $e_i \in E_{Int}$ in the path (cycle) is identified. The behavior of the gate will be represented in a sum-of-products format. If the output of the gate is inverting, and the desired output is 1, then an active low reset will be ANDed with edge (signal) e_i. If the desired output is 0, then an active high reset signal will be ORed with the full gate function. A similar transformation is performed for non-inverting gates. The new gate is used as a possible solution if it is present in the cell library.

Case3: If the vertex (gate) v_i is an inverter, and the inverter drives an edge (gate) that is not an element of the path (cycle), this transformation can be employed. This transformation creates a duplicate inverter v_j, disconnects v_i from the cycle, and applies the Case1 transformation to the new inverter v_j. This case is only applied to performance optimization of cycles requiring reset as defined in Sec. III-A.

Fig. 4a and 4b illustrate these transformations on the circuit in Fig. 3. The Case1 example can be seen where the inverters U3 and U7 have been converted into NOR gates in Fig. 4a using an active high reset because the desired output values for these gates are 0. Case2 is not directly illustrated because it results in an inferior solution according to logical effort. However, assume U2 is being evaluated. This is an inverting gate, and the desired output value of the gate is 1. Active low reset will be ANDed with $rr \in E_{Int}$, changing U2 from an



(a) Power Optimization (b) Performance Optimization (c) Petrify Implementation

Fig. 4: Example 1 circuit implementation with reset

AOI21 gate into an AOI31 gate. Since this gate was present in our library, it is a valid transformation. However, because this solution is of higher cost than a Case1 transformation on gate U3 in this cycle, it will not be used in the final solution. The Case3 transformation is illustrated with the new gate U11 added to the design in Fig. 4b when the performance path $lr \xrightarrow{p} rr$ is provided. The new gate becoming a branching load to the performance path, but adds more area to the design. Since the structure of the circuit is modified, it is possible this transformation adds a hazard to the circuit. Hence in our design flow a formal verification step is performed to ensure hazard fidelity of the design.

A special condition applies to all of these design cases when the input edge in a cycle passes through an inverter that is inside a gate. The Case2 transformation is applied as usual. Additionally, the gate is split into two gates with the inverter becoming an explicit external gate that is added to the cycle. Case1 is then applied to the inverter, and Case2 is applied to the second gate. This is illustrated with the design of Fig. 5a. The inverter bubble has been split into a separate inverter in Fig. 5b with the Case1 transformation applied.

D. Optimization Heuristics for selecting the Best Solution

Power and performance optimizations are based on heuristics that use logical effort [5]. Logical effort theory provides a first-order approximation of the sizes (power) of the gates and the delay for a circuit path (performance). The optimization uses a priority based approach with Delta Logical Effort having the highest priority and Performance/Power Optimization having the lowest priority. If a heuristic solution is better than the previous best solution then no other solution costs are compared.

1) Delta Logical Effort: Logical effort often favors simpler gates over more complex gates due to their high cost. Hence the first step of optimization looks at the relative increase in logical effort of modifying any gate which we name as Delta Logical Effort (ΔLE).

$$\text{Cost} = \Delta\text{LE} = \text{New LE - Old LE} \tag{1}$$

2) Relative load on a gate: Logical effort can be used to estimate the necessary drive strength (also referred to as size) of a gate by calculating the gate's output load. The load estimate is calculated by computing the sum of the logical effort of a gate and the logical effort of the inputs of all the successor gate to which the wire goes. This heuristic penalizes the modification of a gate which drives a big load and thus prefers simpler gates with small output load.

$$\text{Cost} = \text{LE of gate + LE load on gate output} \tag{2}$$

3) Performance or Power Optimization: The solution for this step is selected based on the optimization selected by the user. The heuristics to calculate the cost of the solution for each optimization is described as follows.

Performance Optimization - All three reset transformation cases are applied for performance optimization. However, Case3 is only applied on performance critical paths such as $lr \xrightarrow{p} rr$ that are optionally supplied by the user.

The quality of the solution for each cycle (path) requiring reset is the delay for each input to output path in the design. The total cost of solution is the sum of the delay of all the performance critical paths in the design. Hence the final solution is selected based on the least overhead cost which is calculated by the following heuristic.

$$\text{Performance cost} = \sum_{\text{all paths}} N * F^{1/N} + P \tag{3}$$

where Delay of N-stage path $= N * F^{1/N} + P$ and $F = G * B * H$ [5]. This heuristic assumes electrical effort H for each path to be 1. This can result in sub optimal results if the fanout load of the circuit output is big.

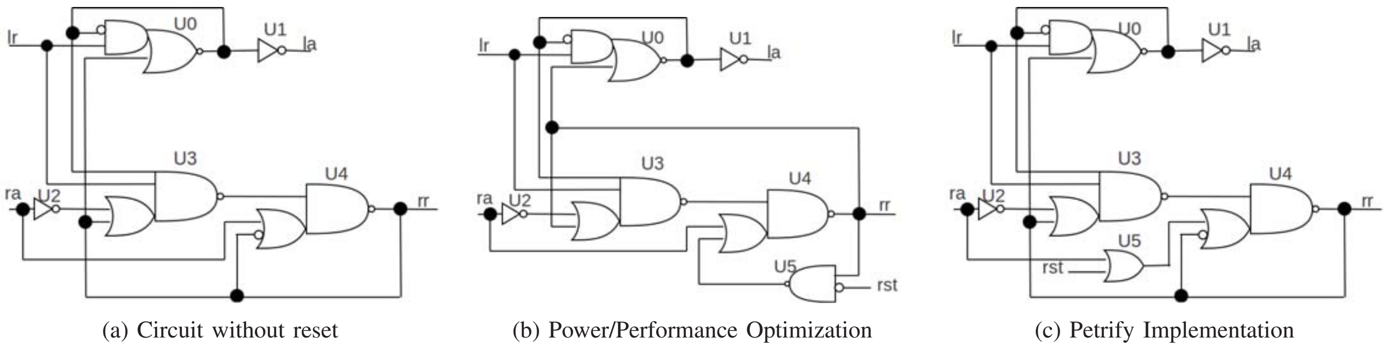
Power Optimization - Power consumption of a design depends on the total capacitance of the circuit that needs to be switched. We approximate the capacitance with the logical effort G of each gate, where a higher logical effort implies a larger input capacitance. Thus the total solution for power optimization is calculated as follows.

$$\text{Power cost} = \sum_{\text{all paths}} \sum_{0}^{i-1} \text{Avg. input LE for gate } V_i \text{ on a path} \tag{4}$$

IV. Examples

A. Example 1

The initial circuit for this example is shown in Fig. 3. For this example, lr, ra, U1, U3, U7 have a logic level 0 while U0, U2, U4, U5, U6 have a logic level 1 at reset state. It consists of six cycles: $< U0, U1 >$, $< U2, U3 >$,



(a) Circuit without reset (b) Power/Performance Optimization (c) Petrify Implementation

Fig. 5: Example 2 Circuit Implementations with and without reset

$< U6,U7 >$, $< U0,U1,U2,U3 >$, $< U0,U1,U6,U7 >$, and $< U0,U1,U6,U7,U2,U3 >$.

Cycle $< U0,U1 >$ does not need to be reset because it is defined by signals in the external signal set $E_{Ext} = \{\text{lr}, \text{rr}, \text{csc0}\}$. Of the other five cycles, only two need to be reset due to shared paths in the cycles. By reseting cycle $< U2,U3 >$ and $< U6,U7 >$, the (U2,U3) and (U6,U7) edges become defined, resetting the remainder of the cycles.

Fig. 4a shows the result applying the power optimization heuristic. The case1 optimization results in the best solution for both $< U2,U3 >$ and $< U6,U7 >$. This optimization modifies U3 and U7 from inverters to a NOR gates.

Results of performance optimization for the same circuit is shown in Fig. 4b. The path from $lr \xrightarrow{p} rr$ ($< U6,U7,U2,U3 >$ and $< U0,U1,U2,U3 >$) and $lr \xrightarrow{p} la$ ($< U0,U1 >$) are defined as performance paths. Thus Both Cycle2 and Cycle3 are candidates for Case3 optimizations, that can push the added complexity of the reset gates off the critical path. Gates U3 and U7 are first duplicated to add U10 and U11 in the feedback of both these cycles. These duplicate gates are then converted to NOR gates that reset the cycles.

This example so far has assumed that the PIs are all defined. Consider the power optimization case when input lr is initially undefined. The algorithm then starts with the circuit of Fig. 4a, marking lr as undefined. This results in the output of U0 and U1 being undefined resulting in la output being undefined. Applying the optimizations results in the gate U1 being changed into a NOR gate with reset. If the performance optimization solution is considered then the path $< U6,U7,U2,U3 >$ is also undefined resulting in gate U7 being converted into a NOR gate with reset. Note that this results in an inferior solution since there are 2 NOR gates performing the same task. Hence the application of undefined input solution is the best for power optimization, but can result in an inferior solution for performance optimization in certain cases.

Petrify is used to apply reset to this sample circuit. Reset is achieved by using generic AND and OR gates as shown in Fig. 4c. U10, U11 and U12 are added to initialize Cycle1, Cycle2, and Cycle3 respectively. Notice that gate U10 is not required, resulting in an inferior solution in terms of power and performance.

B. Example 2

The second example circuit is shown in Fig. 5a. For this example, lr, ra, U1, U4 have a logic level 0 while U0, U2, U3 have a logic level 1 at reset state. It consists of 4 cycles $< U0 >$, $< U4 >$, $< U3,U4 >$ and $< U0,U3,U4 >$. Assuming the PIs are defined, only the $< U4 >$ cycle needs to be reset, because reset values for lr and rr define Gate U3. Fig. 5b and 5c show the solution for this algorithm and Petrify respectively. The optimized circuit generated by this reset algorithm is the same for both power and performance optimizations since the reset is not on a critical path. Petrify adds the OR gate U5. This increases the latency on the $ra \xrightarrow{p} rr$ resulting in a 10% increase in the backward latency and thus a 5% increase in the cycle time.

V. Results

The results of adding reset initialization with this algorithm is compared against Petrify. Benchmark circuits for GCD, PostOffice and PSCSI were employed as well as a set of 128 untimed four-cycle handshake controllers generated by concurrency reduction [7]. Each design in the controller set was tested as a four deep FIFO. All of these designs are synthesized and technology mapped with Petrify with and without reset addition. Our algorithm is applied to these circuits without reset. Petrify adds generic gates for reset addition, hence for comparison these gates are technology mapped using a script. The technology mapping is applied to the academic Artisan library for the IBM 65nm process.

This algorithm resulted in functionally correct circuits for all designs to which power optimization was applied, while application of performance optimization resulted in two circuits that failed due to hazards that were introduced. Petrify failed to generate a working circuit for one of the FIFO controllers since it assumed all the inputs to be defined at logic level 0 upon reset.

Performance, power, and area comparisons are performed by using timing driven optimization in commercial EDA tools. The flow is structured and automated in a way that will produce results that are as fair as possible. The flow uses Design Compiler for sizing, SoC Encounter for place and route, and Modelsim and Primetime for performance and power evaluation using VCD and SPEF files.

The example set ranges in complexity from 4 to 71 gates, and up to 77 cycles. The maximum runtime for the algorithm was less than three seconds for the gcd example, which contains 11 inputs, 9 outputs, 71 gates, and 22 cycles. Critical paths from $lr \xrightarrow{p} rr$ and $lr \xrightarrow{p} la$ were provided for the 128

TABLE I: RESULTS COMPARISON FOR BENCHMARK CIRCUITS

Benchmark Circuit	Petrify Area (um^2)	Petrify Energy/ token (pJ)	Petrify SimTime (ns)	Power Optimization Area (um^2)	Power Optimization Energy/ token (pJ)	Power Optimization SimTime (ns)	Performance Optimization Area (um^2)	Performance Optimization Energy/ token (pJ)	Performance Optimization SimTime (ns)	Power Benefits Area (um^2)	Power Benefits Energy/ token (pJ)	Power Benefits SimTime (ns)	Performance Benefits Area (um^2)	Performance Benefits Energy/ token (pJ)	Performance Benefits SimTime (ns)
gcd	298.3	0.50	303.76	287.2	0.50	297.56	285.4	0.46	298.33	1.04	1.00	1.02	1.05	1.08	1.02
postoffice-rcv-setup	36.0	0.02	87.14	27.4	0.02	85.70	31.7	0.03	84.98	1.31	1.20	1.02	1.14	0.89	1.03
postoffice-sbuf-send-ctl	132.0	0.38	317.85	100.3	0.30	315.62	109.7	0.32	319.69	1.32	1.28	1.01	1.20	1.19	0.99
pscsi-isend	185.2	0.36	244.45	172.3	0.35	276.23	198.0	0.43	256.25	1.07	1.03	0.88	0.94	0.84	0.95
pscsi-trcv-bm	114.0	0.19	143.07	99.5	0.15	136.08	98.6	0.15	135.37	1.15	1.24	1.05	1.16	1.29	1.06
pscsi-tsend-bm	140.6	0.28	202.83	134.6	0.25	213.86	139.7	0.26	208.73	1.04	1.11	0.95	1.01	1.10	0.97
pscsi-tsend	147.5	0.24	203.12	145.7	0.25	191.28	145.7	0.25	191.28	1.01	0.99	1.06	1.01	0.99	1.06
Average Benefit										1.14	1.12	1.00	1.07	1.05	1.01

TABLE II: CONTROLLER CIRCUIT COMPARISON

Optimization	Average Case Power	Average Case Performance	Best Case Power	Best Case Performance	Worst Case Power	Worst Case Performance
Forward Latency	1.00×	1.08×	2.33×	2.33×	0.63×	0.69×
Backward Latency	1.05×	1.12×	1.47×	1.72×	0.61×	0.71×
Cycle Time	1.03×	1.06×	1.39×	1.69×	0.54×	0.54×
Area	1.21×	1.12×	1.91×	1.66×	0.70×	0.69×
Energy/token	1.24×	1.12×	2.19×	1.84×	0.64×	0.64×

FIFO controllers.

Tables I and II show the average benefits for both optimizations with respect to Petrify. Performance optimization results in an improvement of 8%, 12% and 6% in forward latency, backward latency and cycle time for the 128 FIFO circuits. The benchmark circuits show only 1% improvement in performance (reported as simulation time – SimTime). A 12% reduction in area and energy/token for the FIFO controllers is observed, as compared to a 7% and 5% reduction in area and energy/token respectively for the benchmark circuits.

Power optimization results in no improvement in forward latency, and minor improvements in backward latency and cycle time for FIFO controllers as well no performance benefit (SimTime) for the benchmark circuits. However, there is a significant improvement in terms of area and energy. A 21% and 24% reduction in area and energy/token respectively are seen for the FIFO controllers, while a 14% and a 12% reduction was seen for the benchmark circuits.

VI. CONCLUSIONS

Sequential circuits require a reset signal to initialize them to their correct starting state. An algorithm was developed and implemented in C++ to generate reset logic for asynchronous finite state machines. The algorithm defines the relationship between reset and topological cycles in a circuit. The new algorithm also provides heuristics to optimize the reset logic for power or performance. It requires that the design has been technology mapped to the desired implementation library, and that the library consists of single output static logic gates. Inputs to the algorithm include the design netlist, the logic level of all circuit nets, and the behavior of the gates in the technology library. Optional inputs include a set of critical paths for performance optimization, and a set of inputs that may initially be undefined upon reset.

The algorithm is applied to a set of seven large benchmark circuits and a set of 128 pipeline controllers that are configured into linear FIFOs. The designs range in complexity of up to 71 gates and 77 cycles. Maximum runtime for the tool is less than three seconds. Results are compared against Petrify. Performance heuristics show just a 1% performance improvement for the benchmark circuits. The FIFO designs show a 6% performance improvement and a 12% and 8% improvement for backward and forward latency. Power heuristics show an average improvement of 14% and 12% in area and energy per token for the benchmark circuits, and an average area and energy per token improvement of 21% and 24% for the FIFO controllers.

The algorithm is agnostic to how the circuit was implemented and technology mapped, so it can be used with any of the synthesis engines as well as with hand designed circuits. For the first time reset can now become part of any asynchronous finite state machine design automation flow.

VII. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant Number 1218012 and the Semiconductor Research Corporation under Grant Number 2235.001.

REFERENCES

[1] K. Y. Yun and D. L. Dill, "Automatic Synthesis of Extended Burst-Mode Circuits: Part I (Specification and Hazard-Free Implementation)," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 101–117, Feb 1999.

[2] ——, "Automatic Synthesis of Extended Burst-Mode Circuits: Part II (Automatic Synthesis)," *IEEE Transactions on Computer-Aided Design*, vol. 18, no. 2, pp. 118–132, Feb 1999.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, 1997.

[4] R. M. Fuhrer and S. M. Nowick, *Sequential Optimization of Asynchronous and Synchronous Fininte State Machines: Algorithms and Tools*. Kluwer Academic, 2001, minimalist reference.

[5] I. Sutherland, R. Sproull, and D. Harris, *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann, 1999.

[6] S. M. Burns, "General Conditions for the Decomposition of State Holding Elements," in *Advanced Research in Asynchronous Circuits and Systems (ASYNC-96)*, March 1996, pp. 48–57.

[7] S. Nagasai, K. S. Stevens, and G. Birtwistle, "Concurrency Reduction of Untimed Latch Protocols – Theory and Practice," in *International Symposium on Asynchronous Circuits and Systems*. IEEE, May 2010, pp. 26–37.