General Approach to Asynchronous Circuits Simulation Using Synchronous FPGAs

Ruslan Dashkin and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520
{ruslan.dashkin, rajit.manohar}@yale.edu

***Abstract*—Using Field Programmable Gate Arrays (FPGAs) for software and hardware verification and development is a standard step in the digital Application Specific Integrated Circuits(ASIC) design flow. However, asynchronous FPGAs are not available on the market and commercially available FPGAs provide support only for synchronous circuits. Although a lot of research effort has been undertaken in order to use synchronous FPGA to map asynchronous circuits, proposed solutions are typically lacking automation and target specific circuit style and sometimes specific FPGA vendor. In this work we present an automated solution for asynchronous circuits mapping onto the synchronous FPGAs. We build a synchronous model of the original asynchronous circuit based on the event-driven simulation concepts. Proposed approach supports a wide range of circuit styles, including those with various timing assumptions and complex circuitry structures incompatible with the standard synchronous flow. We avoid using vendor specific features, so that the model we generate can be implemented on any commercially available FPGA. We provide an extensive evaluation of our solution and demonstrate that our approach results in a speedup factor of 1.3×10^5 against an asynchronous circuit simulator, 2.8×10^4 against commercial digital simulators, and is 16.5 times slower than the expected performance of the original asynchronous circuit implemented as an ASIC.**

***Index Terms*—FPGA, Asynchronous VLSI, Event-Driven Simulation, Prototyping**

I. Introduction

Using Field Programmable Gate Arrays (FPGAs) for software and hardware verification and development is a standard step in the digital design flow. FPGAs are commonly used for a rapid prototyping of synchronous designs expressed in hardware description languages (HDL) such as Verilog or VHDL. FPGA toolchains provide extensive support for the widely used synchronous circuits and limited support for asynchronous circuits. Asynchronous circuits, also called self-timed, are circuits which do not use external clock signals. Instead, these circuits rely on the handshaking protocols for data sampling and operate only when valid data is available. Asynchronous circuits have many potential advantages over their synchronous counterparts, including energy-efficiency, process variation resilience, avoidance of global clock distribution issues, etc. Asynchronous design methodology includes a wide range of well-defined and accepted circuit families. Each family has distinctive features that lead to better characteristics in terms of area, power and/or performance.

Differences in the data sampling approach, complex timing requirements and a wide range of design styles are the reason why synchronous FPGAs are not well suited for the direct implementation of asynchronous circuits. In particular synchronous FPGAs are not guaranteed to be hazard free, their precise timing characteristics are not readily available to users, and automated placement and routing are not predictable. All these make it challenging to map asynchronous circuits onto a synchronous FPGA in a fully automated fashion. In addition, dedicated resources for circuits like arbiters, configurable delay lines or C-elements, crucial for many asynchronous design approaches, are missing.

In this paper we present a new approach to the problem of mapping asynchronous circuits onto synchronous FPGAs for faster functional simulation. The input to our system is a technology-independent gate-level netlist specified in the ACT asynchronous hardware description language [1], [2]. Our approach supports a wide range of asynchronous circuit families in a unified input format. The input is automatically converted into a synthesizable synchronous Verilog model. The model is functionally equivalent to the original asynchronous design and can be mapped onto commercially available chips using standard FPGA tools. Our approach is flexible and can be used for a wide range of asynchronous circuit families. Unlike many previous approaches, we do not specify predefined building blocks and constraints on the circuit placement and optimization. This allows us to use all the capabilities of an FPGA, and results in good performance and resource utilization.

Our translation preserves both the communication protocols and gate topology of all the components of the original asynchronous design, providing designers with a very precise functional model of the circuit. At the conversion step, circuit structures incompatible with the synchronous methodology are translated into compatible synchronous equivalents. We emphasize that our solution targets gate level simulation rather than cycle accurate synchronous Register Transfer Level (RTL) prototyping. We also note that our model has different timing characteristics from the asynchronous Application Specific Integrated Circuit (ASIC) implementation, although we guarantee that the relative timing constraints [3] are satisfied and the functionality of the original circuit is preserved.

The main contribution of this paper is a unified and automated approach to mapping asynchronous circuits onto synchronous FPGAs. Using our solution, different asynchronous circuit styles can be translated into a synthesizable Verilog model. We provide solutions for the correct implementation of timing assumptions, arbiters and distributed drivers as well as an extensive evaluation of the proposed approach. We demonstrate a runtime comparison with an asynchronous simulator, commercial and open source synchronous simulators, and with an equivalent ASIC.

The rest of the paper is organized as follows: In Section II we describe features from the ACT framework that are crucial to support a wide range of circuit styles. Section III explains in detail the insights that drive our implementation as well as our approach to the translation and the algorithms we use to convert asynchronous circuits into functionally equivalent and synthesizable Verilog. Section IV presents experimental results, including comparison to asynchronous and synchronous software simulators and an equivalent asynchronous ASIC. Section V covers an overview of the related work and compares our solution with existing approaches. Finally Section VI describes our future plans.

II. Specifying Asynchronous Circuits

There is a wide range of asynchronous circuit families. Each circuit family has its own set of typical gates, as well as different forms of timing assumptions required for correct behavior. Instead of adopting a case-by-case approach to describing and specifying each circuit family, for example, by using a large collection of family-specific modules, we use the ACT framework for asynchronous circuit design.

The ACT framework can be used to describe a wide range of asynchronous circuits families in a unified syntax and semantics. Although ACT supports different levels of abstraction (e.g. behavioral versus structural descriptions), we focus on the gate-level circuit description syntax. We also ignore explicit gate sizing specification (transistor length and width) since our goal is functional simulation.

A. Describing arbitrary gates

Logic gates are built with networks of P- and/or N-type transistors to pull output signal up or down, i.e. to $HIGH$ and LOW logic levels respectively. In ACT both networks can be specified with *production rules*. A general structure of a Production Rules Set (PRS), i.e. gate description, is shown below:

1 $G^+ \rightarrow S\uparrow$
2 $G^- \rightarrow S\downarrow$

Here, G^+ and G^- are mutually exclusive Boolean expressions, S is the gate output and arrows specify signal transition direction. When the guard G^+ (G^-) is true, the statement $S\uparrow$ ($S\downarrow$) is executed, and the output of the gate S is pulled-up (pulled-down) to Vdd (GND). An example of the 2-input NAND gate PRS is shown below.

1 $\sim\texttt{in1} \mid \sim\texttt{in2} \rightarrow \texttt{out}\uparrow$
2 $\texttt{in1}\ \&\ \texttt{in2} \rightarrow \texttt{out}\downarrow$

We use logic OR (“|”) to specify parallel connection of the transistors, logic AND (“&”) for serial connection, and NOT (“∼”) for an inverted active level. The example above can be interpreted as follows: when both inputs are set to $HIGH$ the output is set to LOW or when at least one input is set to LOW the output is set to $HIGH$. This syntax is general enough to specify both combinational and state-holding gates, such as C-elements that are commonly used in asynchronous logic.

B. Timing constraints

To provide a wide range of circuit design options, ACT supports the notion of a *timing fork* to define timing assumptions. It has a form as shown below:

$$\texttt{timing A(or}\uparrow/\downarrow\texttt{) : B(or}\uparrow/\downarrow\texttt{)} < \texttt{C(or}\uparrow/\downarrow\texttt{)}$$

Directions of the signals transitions are specified based on design requirements. To define transition direction either no symbol is used, which is interpreted as a stable state after any transition, or a “↑” or “↓” is used to specify a LOW to $HIGH$ or $HIGH$ to LOW transition respectively. The statement above can be interpreted as follows: “after A changes, any change in B should occur before C changes.” More precisely this structure defines an illegal sequence of signals transitions. An execution is illegal if the specified sequence is violated, i.e. an update of C happens before an update of B without intervening of A update. All other executions are valid.

This feature permits ACT to be able to express a large class of timing constraints and support a wide variety of asynchronous circuits families. This is because relative timing constraints [3]—a powerful mechanism for specifying timing constraints in asynchronous control circuits—are readily expressed using timing forks.

C. Relative strengths for a single output

Some digital circuits rely on the relative drive strength of pull-up and pull-down networks for correct operation. This technique is also useful in the circuits with a resistive/passive pull-up/-down. In the realization of such circuits, signal drive strength is characterized by transistor geometry (i.e. width W over length L ratio W/L). ACT supports labeling a production rule as “weak” to express this requirement. The general structure of a gate with weak networks is shown below:

1 $G_1^+ \rightarrow S\uparrow$
2 $G_1^- \rightarrow S\downarrow$
3 $[\texttt{weak} = 1]\ G_2^+ \rightarrow S\uparrow$
4 $[\texttt{weak} = 1]\ G_2^- \rightarrow S\downarrow$

Relative strength is commonly used in various memory cell architectures [4]. An example of such circuit is a standard register file cell shown in Fig. 1. In this example, the pull-up transistors $P1$ and $P2$ should be weaker than the pull-down transistors $N1$ and $N2$ to guarantee correct behavior of read and write operations.

D. Arbiters

Arbiters or mutual exclusion elements are circuits used in asynchronous designs to ensure exclusive state selection for a group of signals. Reasoning about the correct operation of an arbiter is based on the detailed analog properties of the underlying circuit. To support arbiters without having to invoke analog simulation, ACT provides directives that can specify a

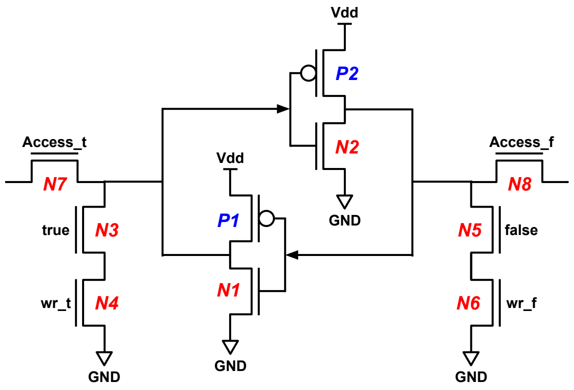


Fig. 1: Register file memory cell transistor level schematic as an example of a relative drive strength application. Transistors P1 and P2 are weaker than N1 and N2 to guarantee correct behavior for read and write operations.

set of mutually exclusive signals. This directive enforces the mutual exclusion constraint regardless of the circuit specified by production rules.

An example below shows the usage of an exclusive high and exclusive low directive in the ACT language. The upper row means that at most one signal in the parentheses can be $HIGH$. The lower row means the opposite, that is at most one signal can be LOW.

```
1        mk_exclhi(a, b, c)
2        mk_excllo(p, q, r)
```

Arbiters are useful in a variety of applications, e.g. on chip routers [5] and clock domain crossing circuits [6].

E. Summary

The syntax above permits ACT to specify a wide range of asynchronous circuits. In particular, the syntax supports quasi delay-insensitive, speed-independent, relaxed quasi delay-insensitive, as well as more aggressive circuit families such as GasP and MOUSETRAP.

III. Circuit Modeling and Implementation

The ACT description can be viewed as specifying a list of production rules (with optional weak directives) along with a collection of mutual exclusion directives and timing directives. The task is to translate this into a synchronous model synthesizable for an FPGA that provides a valid simulation of the asynchronous circuit.

To understand how we approach this translation process, it is instructive to review how circuit simulators operate. A circuit simulator is commonly implemented using a discrete-event simulation engine, where a discrete event is a signal state transition scheduled in a global time order. A change of a signal state can propagate and cause other signals to change at some future time. When a signal transition is initiated, it takes a certain amount of time to switch the signal state. The switching time period is determined by the delay model used in the simulator. Such a simulation can be viewed as constructing an execution trace of the computation.

A correct simulation is the one where the execution trace is consistent with the circuit specification. In our case, this means that the trace has to be consistent with the logic specified by the production rules, the timing constraints and mutual exclusion specifications.

To run a simulation on the synchronous FPGA, we rely on the following approach. First, we build a synchronous model of an original asynchronous circuit. In this model, every original signal is assigned a corresponding synchronous logic signal. Hence, in the end of each clock cycle we can determine values of the asynchronous signals based on the values of the respective synchronous signals. This corresponds to a particular snapshot in time of the simulation. Second, the evolution of the model, as the clock advances, is viewed as constructing an execution trace of the asynchronous circuit.

We exploit the parallelism in the underlying FPGA to execute a large number of parallel signal evaluations, including those which do not change states. Unlike in CPU based solutions, in our approach an idle firing does not incur additional computational cost, i.e. without performance degradation. The main consideration is that we have to ensure that parallel signal changes are still consistent with the original asynchronous circuit description. In particular, we have to ensure that the specified timing constraints are respected by the synchronous model.

This strategy gives us a significant improvement in the simulation run time compared to the software implementation as will be shown in Section IV. The following sections describe different strategies that we have developed to automatically construct a synchronous model of an asynchronous circuit.

A. Option I: Unit delay gates

The simplest and the most direct approach to constructing a synchronous model is to view each clock cycle as advancing time by one unit, and also assume that we can simulate asynchronous circuits using a unit delay model. Such an approach is valid for both quasi delay-insensitive and speed-independent asynchronous circuits that do not require any additional timing constraints. However, it may not be sufficient to guarantee that all timing assumptions are satisfied. We show how timing constraints are handled in Section III-C.

The strategy to implement this unit delay model is the following: (i) we introduce a flip-flop for each signal in the asynchronous circuit. A logic state of this flip-flop corresponds to the state of the original signal; (ii) we translate an ACT specification to the appropriate Verilog syntax using the same Boolean functions as described by the corresponding production rules. We also translate mutual exclusion elements as shown in Section III-E. This translation computes the correct “next state” of the asynchronous circuit, given the “current state” as determined by the values of all the flip-flops in the circuit. It should be noted that each state computation is a fully parallel process, i.e. all gates are evaluated simultaneously on the FPGA.

To understand this approach consider the circuit example in Fig. 2. It is an asynchronous circuit, where each gate in a dashed box is described with a production rule set and

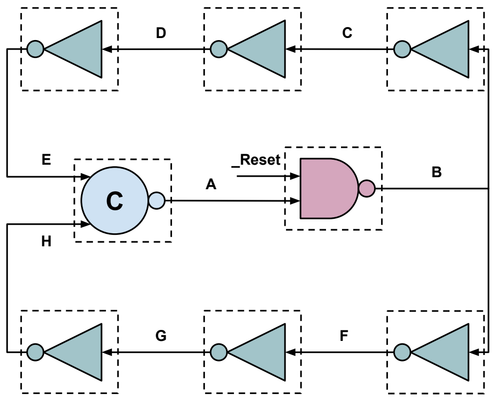


Fig. 2: Gate level diagram of an asynchronous circuit to demonstrate implementation of the synchronous simulation approaches described in Section III-A and III-B. Each gate in a dashed box is described with a set of Production Rules. Signals are tagged with letters for clarity.

each signal is tagged with a capital letter A-H. Applying Option I to this circuit results in the synchronous model shown in Fig. 3a. As stated above for each original signal we introduced a flip-flop, which for clarity is tagged with the corresponding letter. These flip-flops store the state of the model. At each simulation time step, i.e. at every clock cycle, all the gates are evaluated and their next state is determined. At every rising edge of the clock all new signals' states are stored in the assigned flip-flops updating the entire model state. Fig. 4a demonstrates simulation progress of the model from Fig. 3a. Using the same symbols and tags Fig. 4a illustrates how signals values are propagated to the next simulation step and how the next state of the circuit is determined. Such an execution corresponds to the simulation trace where all gates are executed in parallel and assigned a unit delay. This is a reliable and straightforward way to model an asynchronous circuit when the unit delay model results in a valid execution trace. With the support of timing constraints discussed in Section II-B and Section III-C this model can be correctly used for modeling any asynchronous circuit family whose timing requirements can be expressed using timing forks.

On one hand, an advantage of this solution is simplicity. It is straightforward to implement, and this design has extremely small amount of logic between flip-flops, making it easy to operate at close to the peak frequency of the target FPGA chip. Section IV demonstrates performance and resource utilization results of this approach. On the other hand, this solution requires one flip-flop per each gate and for large circuits this number grows significantly which results in a waste of memory and logic resources. The amount of logic between flip-flops is small, which reduces the opportunity for logic optimization and complicates mapping large designs onto the FPGA. Moreover, it would cause a long propagation delay in terms of clock cycles, i.e. even at a very high operating frequency, the number of cycles for signal propagation in the best case will be equal to the number of gates on the path in the original asynchronous circuit. This increases the simulation runtime of this mapping.

B. Option II: Reduced number of flip-flops

Our second approach is intended to improve resource utilization and performance compared to the Option I. This goal is achieved by a significant reduction of flip-flop usage. A lower flip-flop to gate ratio provides FPGA tools with an opportunity for logic optimization, and can reduce signal propagation delay. In this case rather than assigning a flip-flop for each signal, we introduce a set of three rules for flip-flop placement sufficient to construct a synthesizable and functionally equivalent synchronous model. The rules are named CYC, SH and DIR to refer to the objects they are applied to, and are detailed in the following paragraphs.

a) Rule 1 (CYC): The first rule is to cut loops in the asynchronous circuit graph. We name this rule CYC to refer to the CYCles it breaks. This prevents the synchronous model from having any combinational cycles and makes it synthesizable without any additional effort. Although, in some cases cyclic circuits could be mapped onto the FPGA with the assistance of vendor specific constraints/directives, we avoid this approach, since our goal is a vendor-neural Verilog model.

b) Rule 2 (SH): The second rule for flip-flop placement is applied to state-holding gates. Rule name SH stands for State-Holding, to emphasize the fact that it is only applied to this type of gate. A state-holding gate is a production rules set where pull-up and pull-down networks can be turned off simultaneously. This can be determined by a direct analysis of the rules. Hence, when both networks are off, the output of the gate maintains its previous state; we introduce a flip-flop to hold the state of the gate, and use the flip-flop value to generate the output when both pull-up and pull-down networks are off. In terms of our synchronous model, this corresponds to introducing a flip-flop in a feedback topology for the gate.

As an example of the benefits of this optimization, consider the commonly used completion tree in asynchronous circuits. It consists of a tree of state-holding C-elements. Having a flip-flop only at the feedback connection reduces delay of this structure from the tree depth with Option I to at most one unit.

It is worth noting that if a state-holding gate is a part of a combinational loop we combine rules CYC and SH and move the flip-flop from the feedback of the gate to its output to improve resource utilization and avoid redundant flip-flops.

c) Rule 3 (DIR): The third rule for the flip-flop placement is on the paths which directly connect inputs to the outputs within an individual module and that have no flip-flops from the previous two cases. The rule is named DIR after the DIRect input-to-output paths it detects for flip-flop insertion. The introduction of this rule means that we can apply rule CYC *within a module*, without having to analyze the entire circuit. In other words, this rule is used to break loops locally without requiring global analysis of the circuit topology. We experimentally determined that this rule also results in good simulation performance by reducing the length of combinational paths. Note that in some highly modular designs this rule can be less effective and result in an implementation closer to Option I. An example of this would be if every gate were in a separate module.

This synchronous model has a different execution trace compared to Option I. In particular, we can view each group of

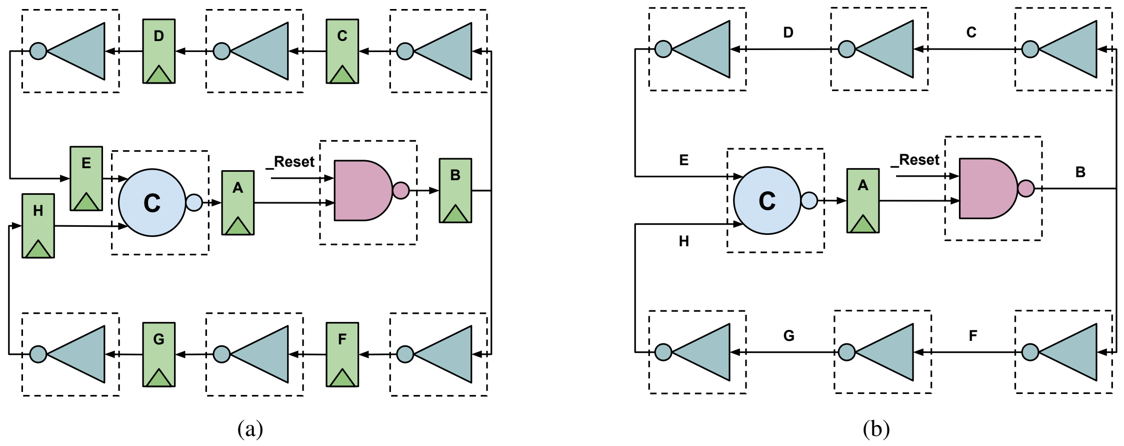


Fig. 3: Gate level diagrams of the synchronous models of the circuit from Fig. 2 implemented with: (a) Option I (Every signal assigned a Flip-Flop); (b) Option II (Reduced number of Flip-Flops). All gates in dashed boxes are described with Verilog using the same Boolean functions as in the Production Rules specification. Flip-flops are tagged using the same letters as the corresponding signals in the original asynchronous circuit.

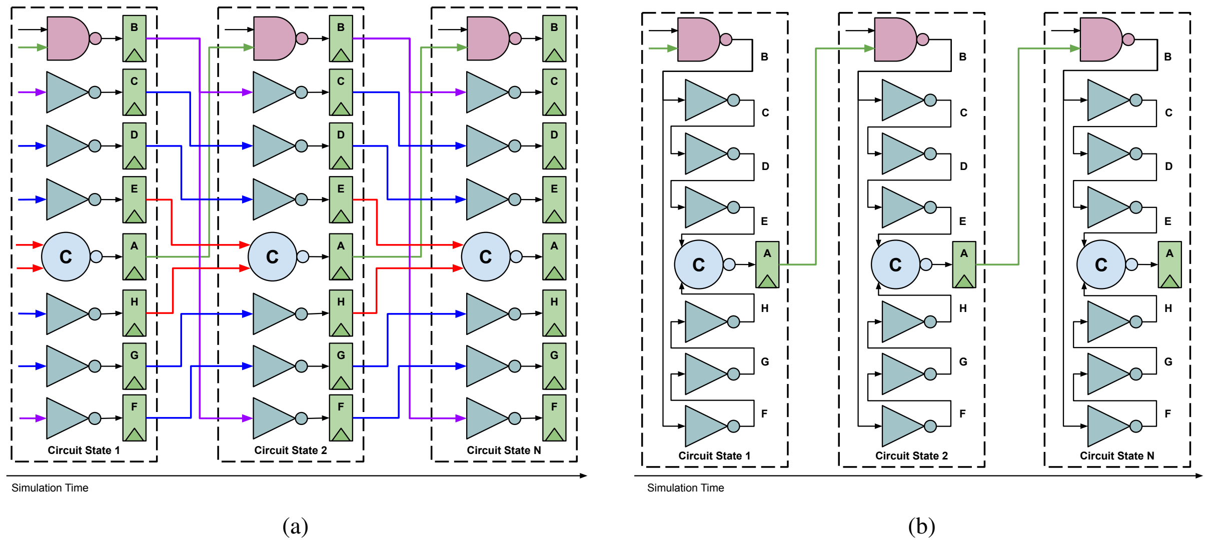


Fig. 4: Simulation progress diagrams of the synchronous models from Fig. 3 implemented with (a) Option I; (b) Option II. Each dashed box represents the state of the model at one simulation time step, i.e. at every clock cycle. All gates in each box are evaluated in parallel. Colored arrows demonstrate signal propagation from each model state to its successor. Black arrows are internal signals.

gates between the flip-flops introduced by the proposed rules as a directed acyclic graph. Vertices of the graph are gates and edges are wires from the output of one gate to the input of another. The subcircuit is completed by the edges that have flip-flops or connected to the primary I/Os. The execution of the asynchronous circuit corresponds to the firing of all the gates in each acyclic portion of the graph in a topologically sorted order and only once at each clock cycle. Such an approach is a correct execution trace for both quasi delay-insensitive and speed-independent circuits, but other circuit families require this approach to be augmented as described in Section III-C.

To clarify the goal of this approach, consider the same circuit example in Fig. 2. Applying Option I to this circuit results in the model shown in Fig. 3a, while Option II produces model in Fig.3b. By following the rules stated above the number of flip-flops in the model is reduced from 8 to 1. In particular, we apply rules CYC and SH combined, i.e. although a combinational cycle is detected, the path is checked for state-holding gates and because of the C-element the cycle is broken by moving feedback flip-flop to the gate output. The circuit state is now stored in one memory cell, and in the outputs of other internal signals (as labeled in Fig.3b). A single clock cycle is sufficient to propagate the new signal state to the rest of the circuit as opposed to Option I where multiple cycles are needed—one per gate. We note that although the number of flip-flops is reduced the circuit is still synthesizable and there are additional opportunities for logic optimization compared to the previous approach. Fig. 4b demonstrates simulation progress of the Option II model. This figure clarifies how the signal state is propagated to the next simulation step and why it takes only one clock cycle. It is worth noting that with this approach, logically we still have access to the values of the signals which do not have an assigned flip-flop. Hence, this model provides the same simulation precision as Option I.

There are various methods that can be used to identify locations in the circuit that require flip-flops for rules CYC and DIR. In our solution we use the depth first search algorithm [7] to find cycles and direct input to output paths in the graph. It is an inexpensive and efficient algorithm, with linear time and space complexity.

Due to the heuristic in this approach for flip-flop placement, sequential and combinational logic may be imbalanced with significantly more combinational logic compared to flip-flops. A small number of registers potentially makes the critical path longer, negatively affecting the clock frequency. However, with this approach signal propagation delay (in terms of the clock cycle count) is much smaller, and logic optimization is more efficient. We quantify this effect in Section IV. For example, a logic optimization tool might optimize the inverter chains in Fig.3b. Note that the logical values of those signals can still be determined from the part of the state in the flip-flops.

C. Timing constraints

When an asynchronous circuit has certain timing constraints that must be satisfied for correct operation, these constraints are explicitly specified in the input description (Section II-B).

In our synchronous simulation models, the delay of the combinational part of the circuit is ignored in terms of the *logical* signal delay, i.e. it does not contribute to the total propagation delay. Hence, to determine the actual simulation delay from one signal to another we count the number of flip-flops on the signal propagation path. To satisfy a timing constraint, we have to make sure that the delay of the “slower” signal is bigger than the delay of the “faster” signal. It means that the number of flip-flops on the faster path must be strictly less than on the slower path. It should be noted that it is possible to have multiple paths connecting the same signals, in which case we consider the worst case for the faster signal and the best case for the slower signal. We compute the maximum and minimum numbers of flip-flops from the source to the destination point and if it is necessary we add extra delay to the path by inserting additional flip-flops. Fig.5a and Fig.5b show an example before and after insertion of extra flip-flops respectively, so that the latter satisfies the condition below:

$$\texttt{timing A(or}\uparrow/\downarrow\texttt{) : B(or}\uparrow/\downarrow\texttt{)} < \texttt{C(or}\uparrow/\downarrow\texttt{)}$$

We note that since we target a functional simulation we consider the case when both up and down transitions of one gate have the same delay.

By ensuring that timing constraints are satisfied, the generated model execution corresponds to a valid simulation interleaving of the original asynchronous circuit.

We complete the description of synchronous model generation by showing how we handle production rules, mutual exclusion constraints, and the hierarchical translation of signals that have multiple drivers in different parts of the design hierarchy.

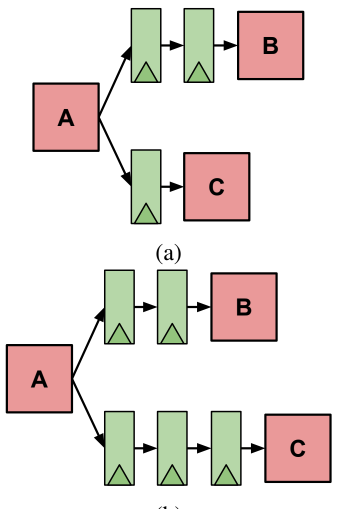


Fig. 5: Example of timing assumptions modeling approach. (a) Violated timing constraint; (b) Satisfied timing constraint.

D. Production rule translation

As it is described in Section II-C PRS consists of up to four transistor networks. As weak networks have smaller drive strength at the circuit level, they are considered to have lower firing priority in the simulation. The example below shows how a generic PRS is converted into Verilog. It is safe to arrange guards in the presented order because their mutual exclusiveness is guaranteed by the design methodology. Also, this order ensures that weak conditions have lower priority.

```
if (G1+) S <= 1'b1;
else if (G1-) S <= 1'b0;
else if (G2+) S <= 1'b1;
else if (G2-) S <= 1'b0;
else S <= S;
```

Combinational gate translation can be simplified by removing feedback connections. Since guard expressions of a combinational gate are compliments of each other it is guaranteed that the output will be updated at every simulation time step. An example of the NAND gate translation is shown below:

```
if (~in1 | ~in2) out <= 1'b1;
else if (in1 & in2) out <= 1'b0;
```

An example of a standard state-holding gate (the Muller C-element) with a feedback connection is shown below:

```
if (~in1 & ~in2) out <= 1'b1;
else if (in1 & in2) out <= 1'b0;
else out <= out;
```

E. Arbiter

To support arbiters or exclusion elements described in Section II-D we implement a behavioral model of the corresponding analog circuit. The main idea is that if multiple inputs are high or low, only one output should be high or low respectively, until the driving input changes its value and

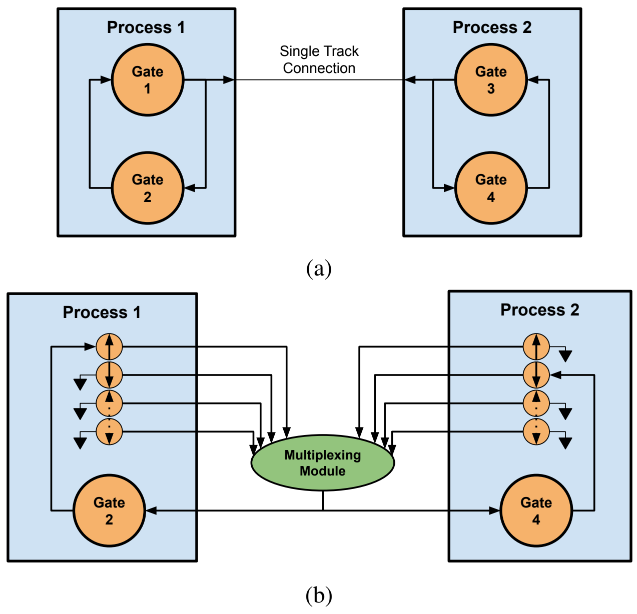


Fig. 6: Abstract example of the distributed drivers circuit. (a) Circuit with a single track communication line; (b) FPGA compatible solution for the single track circuit.

triggers arbiter to pick the next input request. Although it is possible to implement such circuit on synchronous FPGAs as in [8], we avoid using vendor specific constraints and use a fully synchronous model. We also keep track of the previous choice made by the arbiter using local state, to ensure that the arbiter behaves in a fair fashion.

F. Hierarchical distributed drivers

A common practice in some asynchronous circuit methodologies are gates with distributed drivers. One case is when transistor networks associated with a single gate are placed at different levels of hierarchy. Another example are gates with multiple drivers, i.e. not just the gate itself but multiple drivers are distributed in the design—for example, to implement a wired-OR operation.

Not all FPGA architecture supports direct implementation of these structures. Therefore, our solution automatically detects distributed drivers in the circuit hierarchy and converts them into a synthesizable FPGA vendor-/chip-neutral synchronous model. It does not change circuit functionality, does not require manual constraints and does not contribute to the total signal propagation delay.

The circuit in Fig.6a shows an abstract example of a single track communication line. Such a communication approach is used in the GasP and Single Track Full Buffer circuit families. Two processes drive the same line and, for the sake of illustration, we assume that Process 1 pulls the signal up and Process 2 pulls the same signal down, i.e. the drivers are distributed.

A challenge in a hierarchical translation approach arises due to the problem of local variables, not accessible from the scope other than where they are declared. For example, the pull-up in Process 1 might use local variables only accessible within Process 1, and the same situation is likely in Process 2. This means that we cannot simply relocate the gate into one of the two modules to eliminate the distributed driver. The most straightforward way would be to flatten the circuit, i.e. to recursively move both Processes contents to the higher level of the design hierarchy until they appear at the same common level. In the worst case this approach leads to a fully flattened project to propagate local variables from the bottom level to the top.

Instead, in order to preserve the hierarchy where it is possible, we propose another solution shown in Fig.6b. As mentioned earlier (Section II-C), each gate can have up to four transistor networks. The two distributed circuit fragments Gate 1 and Gate 3 (Fig. 6a) are split into four separate gates, each of which represents either pull-up (arrow up), pull-down (arrow down), weak pull-up (dashed arrow up) or weak pull-down (dashed arrow down) network as shown in Fig.6b and in the listing below:

$\texttt{S\_up} \Leftarrow G_1^+;$
$\texttt{S\_dn} \Leftarrow G_1^-;$
$\texttt{S\_wk\_up} \Leftarrow G_2^+;$
$\texttt{S\_wk\_dn} \Leftarrow G_2^-;$

(If the network is missing then it is assigned to a constant zero and later removed from the circuit at the synthesis step.) The port list for the modules is augmented with the newly generated signals S_up, . . ., S_wk_dn. A new module (shown in green in Fig.6b) is introduced that combines all networks with OR gates and that generates only one output as shown in the code below:

$\texttt{if (S}_1\texttt{\_up}|\ldots|\texttt{S}_N\texttt{\_up) S} \Leftarrow 1'\texttt{b1};$
$\texttt{else if (S}_1\texttt{\_dn}|\ldots|\texttt{S}_N\texttt{\_dn) S} \Leftarrow 1'\texttt{b0};$
$\texttt{else if (S}_1\texttt{\_wk\_up}|\ldots|\texttt{S}_N\texttt{\_wk\_up) S} \Leftarrow 1'\texttt{b1};$
$\texttt{else if (S}_1\texttt{\_wk\_dn}|\ldots|\texttt{S}_N\texttt{\_wk\_dn) S} \Leftarrow 1'\texttt{b0};$

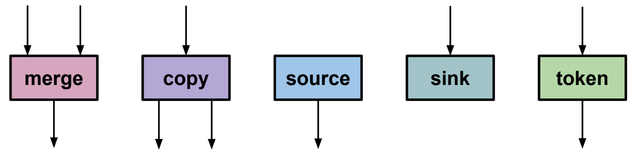
Finally, in Fig.6a, Gate 1 and Gate 3 outputs are used internally as inputs to Gate 2 and Gate 4 respectively. Fig.6b demonstrates how this problem is solved in our implementation. If such port is detected we create an extra input which drives the new net back to the source process.

An analogous solution is applied to the signals with more than one driver. All driving gates will be split in four and directed to the multiplexing module which has single output to drive a common signal.

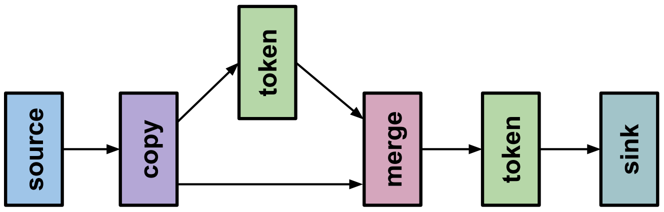
However, we note that our solution cannot detect and resolve distributed drivers related to the primary I/O interface of the top level module. It is not clear how the environment should behave in this case, as this depends on the context of the overall system and the discipline adopted by the designer to handle such signals. Hence, our translation assumes that the primary I/O signals to the entire design are unidirectional and are not a part of a multi-driver structure.

IV. Evaluation

In this section we present our results of implementing various asynchronous circuits on a synchronous FPGA, and compare them to those obtained from the commercial and open-source software simulators.



(a)



(b)

Fig. 7: (a) Basic dataflow elements; (b) Synthetic QDI circuit generated with the set of dataflow elements.

A. Evaluation Setup

For the evaluation we used the Xilinx Virtex-7 FPGA (XC7VX690T-3FFG1761C). We used Vivado 2020.1 for FPGA synthesis and implementation steps. For software evaluation we used both a commercial Verilog simulator and the open source simulator `Verilator`. Both of them show good performance results compared to other tools. To directly simulate an asynchronous description, we used an open-source `prsim` simulator which is a part of the ACT Toolkit. Note that we expect `prsim` to be slower than just running the Verilog model of the design, because it continuously monitors the circuit being simulated for switching hazards and reports those as errors in addition to monitoring all timing constraints during a simulation run. All simulators produced the same results in terms of the computation performed by the circuit in question.

B. Circuits Diversity Evaluation

In this part we show the results of simulating a 15-stage pipeline consisting of a sequence of 4-bit adders, where each example is implemented using a different asynchronous style.

We use it to showcase the fact that our approach is capable of supporting a wide range of asynchronous circuit families with a unified translation approach.

This set of benchmarks is diverse enough to include a range of timing assumptions, different handshaking protocols, and examples of distributed drivers. Table I shows the numbers of LUTs and Flip-Flops in columns 2 and 3 respectively. Columns 4-5 show simulation runtime results for the FPGA implementation and asynchronous software simulator. The simulation scenario is simple and used for illustrative purposes. The circuit takes all zeros at the input (`4'b0000`) and propagates this vector to the output while incrementing results at every stage so that the output is all ones (`4'b1111`). A more comprehensive performance evaluation with longer runtimes and bigger circuits is demonstrated in the following sections.

TABLE I: Synchronous models synthesis results for the circuit implemented in different asynchronous circuits styles with various timing assumptions and synchronization protocols

Family	Number of LUT Option I/ Option II	Number of FF Option I/ Option II	FPGA, Option I/ Option II, sec	Async. Simulator Prsim, sec
Sutherland Micropipeline [9]	955 / 1109	1396 / 1247	$7.1 \cdot 10^{-7}$/ $6.8 \cdot 10^{-7}$	$4.1 \cdot 10^{-2}$
Mousetrap [10]	1042 / 1103	1771 / 1336	$8.5 \cdot 10^{-7}$/ $7.6 \cdot 10^{-7}$	$6.4 \cdot 10^{-2}$
Click Element [11]	1462 / 1439	1951 / 1156	$9.7 \cdot 10^{-7}$/ $8.1 \cdot 10^{-7}$	$6.7 \cdot 10^{-2}$
Quasi-Delay Insensitive [12]	1831 / 2145	2185 / 1870	$4.1 \cdot 10^{-7}$/ $2.2 \cdot 10^{-7}$	$4.5 \cdot 10^{-2}$
Relaxed Quasi-Delay Insensitive [13]	1877 / 2228	2402 / 1997	$4.6 \cdot 10^{-7}$/ $2.4 \cdot 10^{-7}$	$5.3 \cdot 10^{-2}$
Scalable Delay Insensitive [14]	1447 / 1709	1754 / 1509	$2.8 \cdot 10^{-7}$/ $1.5 \cdot 10^{-7}$	$5.3 \cdot 10^{-2}$
GasP [15]	1027 / 1099	1650 / 1243	$11.7 \cdot 10^{-7}$/ $11.4 \cdot 10^{-7}$	$7.0 \cdot 10^{-2}$
Single Track Full Buffer [16]	936 / 1024	1500 / 1108	$4.6 \cdot 10^{-7}$/ $4.3 \cdot 10^{-7}$	$3.3 \cdot 10^{-2}$

C. Generated random QDI circuits

Our next set of benchmarks includes randomly generated QDI circuits built of the cells from Fig.7a. An example of the topology of such a circuit is shown in Fig. 7b.

The number of gates in the original asynchronous circuits as well as the results of logic synthesis for the test circuits are in columns 2–4 of Table II. Option II demonstrates a significant improvement in the resource utilization—up to 1.92x for LUTs and up to 2.31x for FFs against Option I as shown in column 5 of Table II. Note that in both cases, the FPGA tools successfully applied optimizations to the circuit so that the number of LUTs is less than the number of gates in the asynchronous implementation. It means that although Option I uses one flip-flop per gate and is expected to waste FPGA resources, there was still room for optimization.

Columns 6-7 of Table II show performance results for both solutions, the number of clock cycles required for the circuits to go through their internal cycle and return to the initial state and the ratio of Option I over Option II. This metric defines how many clock cycles it takes to execute simulation of the entire circuit starting from the request signal at the primary input to the point when the next request can be sent. Both solutions were successfully implemented at an FPGA target frequency of 500MHz. However, the number of cycles required by Option I, i.e. propagation delay, is almost twice that of Option II. Such results meet our expectation because of the

TABLE II: Benchmark results for the set of QDI circuits with a random topology (Section IV-C)

Name	Initial number of gates	Number of LUT Option I/ Option II	Number of FF Option I/ Option II	LUT ratio/ FF ratio	Number of clock cycles per internal cycle Option I/ Option II	Cycle ratio	Async. Simulator Prsim, sec	Commercial Simulator Option I/ Option II, sec	Free Simulator Verilator Option I/ Option II, sec	FPGA Option I/ Option II, sec	FPGA to ASIC Option I/ Option II
gb1	770	603 / 314	650 / 281	1.92 / 2.31	25 / 11	2.27	28.97	5.01 / 4.61	3.54 / 0.77	0.002 / 0.00088	46.38 / 20.41
gb2	1725	1368 / 765	1480 / 666	1.79 / 2.22	36 / 19	1.89	49.73	8.61 / 8.06	7.79 / 3.37	0.002 / 0.00106	33.54 / 17.69
gb3	3285	2610 / 1467	2818 / 1272	1.78 / 2.21	76 / 39	1.96	44.34	9.10 / 8.29	17.59 / 7.47	0.002 / 0.00102	28.49 / 7.31
gb4	8010	6455 / 3990	7106 / 3399	1.62 / 2.09	180 / 91	1.96	45.79	11.42 / 10.57	47.79 / 23.29	0.002 / 0.00102	32.54 / 16.45
gb5	13473	10703 / 6042	11597 / 5258	1.77 / 2.21	92 / 47	1.96	180.67	35.02 / 32.17	114.22 / 53.35	0.002 / 0.00102	33.67 / 17.20
gb6	28289	22564 / 13107	24586 / 11340	1.72 / 2.17	100 / 51	1.96	452.72	75.53 / 68.81	250.62 / 119.48	0.002 / 0.00102	29.73 / 15.16
gb7	56698	45220 / 26219	49239 / 22687	1.72 / 2.17	105 / 57	1.85	682.13	146.47 / 144.03	575.52 / 288.65	0.002 / 0.0108	31.41 / 17.05
gb8	114026	90916 / 52601	98951 / 45535	1.73 / 2.17	224 / 108	2.08	802.20	148.46 / 130.96	1267.65 / 586.44	0.002 / 0.00096	30.67 / 14.79

larger number of flip-flops used by Option I.

Columns 8-11 of Table II demonstrate FPGA runtime in seconds compared with asynchronous simulator `prsim`, a commercial simulator, and open source simulator `Verilator`. The asynchronous simulator runs for one million unit delays which is equivalent to one million clock cycles in the synchronous domain for Option I. For Option II all simulation runtimes are reduced by the cycle ratio from column 7. In this case both FPGA models are several orders of magnitude faster than their software counterparts. It should be noted that the asynchronous simulator shows results comparable to the synchronous simulators which means that our model has reasonable advantages only in case of being implemented on FPGA.

Finally, column 12 of Table II shows the operating frequency comparison of the FPGA model versus a direct ASIC implementation of the same circuit in a 28nm process technology. The ASIC performance is obtained from an asynchronous static timing analysis tool Cyclone [17]. Our target FPGA chip is also built in a 28nm process. As can be seen, our approach has a 7x to 46x slowdown compared to the ASIC implementation. This is about the same as mapping synchronous circuits to synchronous FPGAs. However, the actual FPGA implementation is fast enough to be used for the purpose of functional verification of an ASIC and software development prior to fabrication.

D. Synchronous circuits converted to asynchronous

In this part of evaluation we used circuits converted from existing synchronous benchmarks into asynchronous logic using the same library of element as in the previous section (Fig.7a). During the conversion a fan-out was translated as a tree of copy elements and any gate as a merge tree. The topology of this group of benchmarks significantly differs from those in the previous section and extends the results inclusiveness.

Columns 2-5 Table III shows the results of the logic synthesis. In this case LUT utilization improvement is up to 1.82x and FF utilization improvement is up to 2.52x.

Columns 6-7 of Table III show results of the implementation of benchmarks including the number of clock cycles per full internal circuit cycle and Option I to Option II ratio. In this case both solutions were implemented at clock frequency of 500MHz. However, the cycle time is better for Option II. Thus, the total runtime in seconds (see columns 8-11 of Table III for Option II is better for this set of circuits as well.

Finally, a comparison to the ASIC performance with the same feature size as in the previous section matches with our expectations (Column 12 of Table III). In this case FPGA model is 10x to 67x slower than ASIC.

Table IV presents a summary of the performance results for this set of benchmarks. It shows geometric mean values of the ratios of the FPGA over ASIC performance as well as the simulators runtime over FPGA simulation for Option I and Option II. Our FPGA simulation demonstrates a significant speedup of multiple orders of magnitude against all software simulators and relatively small slowdown against an ASIC implementation.

E. Further Optimization

It should be noted that all our benchmarks could easily fit on the FPGA chip we targeted, however we understand that if circuit gets closer to the full FPGA capacity it may cause frequency reduction especially in case of Option II with high

TABLE III: Benchmark results for the set of QDI circuits with the topology obtained from the original synchronous circuits (Section IV-D)

Name	Initial number of gates	Number of LUT Option I/ Option II	Number of FF Option I/ Option II	LUT ratio/ FF ratio	Number of clock cycles per internal cycle Option I/ Option II	Cycle ratio	Async. Simulator Prsim, sec	Commercial Simulator Option I/ Option II, sec	Free Simulator Verilator Option I/ Option II, sec	FPGA Option I/ Option II, sec	FPGA to ASIC Option I/ Option II
s27	148	130 / 85	151 / 64	1.53 / 2.36	24 / 12	2	4.83	1.29 / 1.29	0.41 / 0.24	0.002 / 0.001	35.19 / 17.59
c432	1776	1394 / 875	1684 / 668	1.59 / 2.52	82 / 41	2	4.83	5.44 / 4.27	11.20 / 3.13	0.002 / 0.001	30.75 / 15.37
s838x	3970	3380 / 1916	3861 / 1525	1.76 / 2.53	92 / 41	2.22	99.14	24.49 / 16.93	22.63 / 11.26	0.002 / 0.0009	67.45 / 30.06
c7552	12979	10328 / 6316	12357 / 4941	1.64 / 2.50	42 / 21	2	396.51	98.03 / 75.48	114.08 / 66.49	0.002 / 0.001	20.43 / 10.21
s9234x	28560	24595 / 13529	27387 / 10884	1.82 / 2.52	84 / 42	2	526.67	118.39 / 90.77	275.48 / 156.50	0.002 / 0.001	30.59 / 15.29
s15850	49125	43431 / 23813	48450 / 19222	1.82 / 2.52	138 / 69	2	671.09	135.10 / 104.95	544.65 / 304.71	0.002 / 0.001	31.58 / 15.79
tv80	66485	56518 / 30972	62631 / 24908	1.82 / 2.52	106 / 53	2	1402.82	243.58 / 178.89	735.69 / 409.08	0.002 / 0.001	33.13 / 16.56

TABLE IV: Simulation runtime comparison summary for the set of benchmarks with a predefined topology.

	Option I	Option II
FPGA to ASIC	33.54	16.49
Asynchronous Simulator prsim to FPGA	6.21×10^4	12.60×10^4
Commercial Simulator to FPGA	1.76×10^4	2.79×10^4
Free Simulator Verilator to FPGA	2.63×10^4	2.76×10^4

combinational to sequential logic ratio. To improve throughput of the synchronous model a standard technique such as a C-slow retiming [18] can be applied to simultaneously run multiple simulation scenarios. It is a safe method which supports feedback loops and based on adding extra flip-flops to the circuit. An option of adding extra flip-flops can be easily adopted in both our translation approaches.

V. Related Work

Existing solutions can be split into three main categories: (i) Direct mapping; (ii) Asynchronous FPGA architectures; (iii) Synchronous emulation. Each approach is discussed in detail in the following subsections. In the end we summarize the difference between our solution and those presented in this section.

A. Direct mapping

There were many successful attempts to directly map asynchronous circuits onto synchronous FPGAs i.e. to preserve the clockless nature of the original design. This approach is not just based on the clock signal removal from the chip. Usually it is achieved by a set of timing and/or placement constraints to prevent uncertainty in the relative placement of modules and to satisfy timing assumptions.

As noted in Section I, a lack of key circuit blocks in commercially available FPGAs is one of the reasons why they are not well suited to asynchronous designs. Thus, implementing delay lines, C-elements and arbiters is one of the goals of the direct mapping approach. Multiple works proposed solutions that use available FPGA primitives to emulate the required behavior. Hazard free implementation of the C-element on LUT based FPGAs were introduced in [8], [19], [20]. Controllable delay lines can be implemented with LUTs or Carry Chains connected in series as suggested in [21], [22]. However, as the FPGA interconnection delay is not predictable, the delay margins have to be large and be accompanied by a set of placement constraints to ensure correctness of the timing assumptions. An arbiter implementation with analog nature based on the pinned placement is shown in [8].

Another goal of the direct mapping is to provide a methodology for the correct, predictable and reproducible implementation of an entire circuit. One way to achieve this goal is to handcraft a library of crucial building blocks whose behavior has been shown to be correct and leave the configuration of the delay-insensitive part of the circuit to the designer. Predefined elements are usually covered by placement and optimization constraints. [19], [23]–[25] provided such cell libraries. An early work [24] targets 2-phase bundled data circuits and while using fixed blocks, it relies on automated placement and routing to meet timing constraints. [23] provides a library of dataflow blocks for 2-phase bundled data circuit without automating delay line design. [19], [25] both target QDI circuits. The former presents a C-element library and the latter a library of QDI Boolean functions. In these cases an isochronic fork assumption is respected by an appropriate physical relative placement constraint. Another group of methods relies on the iterative analysis of the implementation results and adjustment of timing and/or placement constraints as well as updating delay lines if necessary to ensure correct operation. A complete methodologies for the bundled data circuits designs are presented in [8], [26], [27]. In addition to bundled data circuits [28] showed a method for speed independent circuits mapping.

The main limitations of the solutions mentioned above compared to our proposed technique include: (i) necessity for manual and/or scripting assistance to ensure correct circuit operation; (ii) specific circuit family implementation; (iii) vendor/chip specific features involvement. All these limits a user's ability to reuse these solutions if the design style is changed or the FPGA platform is switched to another vendor.

B. Asynchronous FPGA

An asynchronous FPGA architecture is a good way to avoid the complications of preserving the asynchronous nature of the design while mapping it to reconfigurable logic. Various solutions were proposed in [29]–[33]. FPGA architectures [29], [32] explicitly support self-timed circuits alongside with synchronous circuits. [29], [31], [33] demonstrate architectures designed specifically for self-timed designs. However, such FPGAs while being perfect candidates for asynchronous circuits implementation, are not easily available to the researchers in the same way as synchronous FPGAs are. Hence, our approach is more accessible to researchers looking for high-performance simulation of asynchronous circuits.

C. Synchronous Emulation

Another approach to asynchronous circuits mapping onto commercial FPGAs is based on synchronous emulation described in [34]. This solution uses predefined synchronous behavioral analogs of the basic asynchronous components used for Tangram [35] syntax-directed translation into asynchronous circuits. Instead of the actual signal transitions they use pulse based logic, where each transition is represented with a single pulse. Thus, 2-phase and 4-phase two wire handshake protocols with arbitrary design functionality can be implemented. The drawback of the pulse based approach is that it is less precise and would require extra resources to keep track of the current state of the control signals. In contrast, our approach can handle a much wider class of asynchronous circuits with various control circuit topologies and preserves an original gate structure and communication protocols.

D. Translation differences: an example

A simple asynchronous pipeline component is one with a single input and output channel, where the channels use request and acknowledge signals to implement a handshake protocol. A linear array of such components can be used to implement an asynchronous pipeline. To highlight the difference between our proposed translation and existing approaches, we show the translation of this basic asynchronous circuit using different approaches.

Fig.8 shows a linear pipeline controller, where each stage has 4 ports: REQ_LEFT an input request signal from the previous stage, ACK_RIGHT is an input acknowledgment signal from the next stage, REQ_RIGHT is an output request signal to the next stage and ACK_LEFT is an output acknowledgment signal to the previous stage. Fig.9a shows the original asynchronous circuit implementation, where 'C' stands for the C-element and one of its inputs is inverted. The delay box represents an implementation of a delay line. Fig.9(b,c,d,e) show various possible mappings of this circuit to FPGAs.

a) Direct Mapping: One possible direct mapping approach incorporating multiple proposed solutions is shown in Fig.9b. It uses a hazard free LUT based C-element([19]) and a delay line constructed of a 1-input LUT chain ([21]). In this approach all modules are placed at fixed locations to guarantee hazard free control signal implementation and the delay line is marked so that the tools do not optimize it away. For more complex gates, resolving the same issues can be more difficult and resource inefficient.

b) Synchronous Emulation: The synchronous emulation approach described in [34] is shown in Fig. 9c. The C-element behavior is emulated with a predefined *Join* module. The *Join* element itself is implemented using an adder with the sum output connected back to one of the adder inputs through the flip-flop, and the carry signal connected to the module output. Delay lines require the manual addition of chains of flip-flops. It should be noted that control gates that do not match the pre-existing set of blocks cannot be translated without manually extending the set. Also, since pulses are used to represent signal transitions, the current state of the synchronous circuit cannot be directly used to determine the state of the asynchronous circuit being emulated.

c) Asynchronous FPGA: A potential solution for asynchronous FPGA mapping is shown in Fig. 9d. In this example we consider an FPGA Slice architecture proposed in [32]. We do not state that the presented mapping is precise, however, we emphasize that the underlying architecture is sufficient to place the necessary modules in one Slice and to guarantee a hazard-free implementation. It is shown that the LUT can be configured with a state-holding function and have a legal feedback connection. Also, the configurable delay line is built-in and does not waste extra FPGA resources.

d) Proposed Solution: Finally, Fig.9(e) is an implementation based on our approach. The C-element is mapped onto a 3-input LUT with a stable feedback connection through the flip-flop. The delay line is a chain of flip-flops. No assumptions about the underlying FPGA are necessary beyond what is already required to map standard synchronous logic. Also, the FPGA tools are free to optimize any of the combinational logic in the usual way. We note that the presented circuit is automatically generated from the asynchronous description and involves no adjustment either manual or using scripts to ensure a correct implementation on the FPGA.

E. Summary

Table V summarizes the synchronous FPGA solutions where column 2 shows the supported asynchronous circuit families, column 3 shows the total number of the supported control circuit topologies in all mentioned families (e.g. Mousetrap,

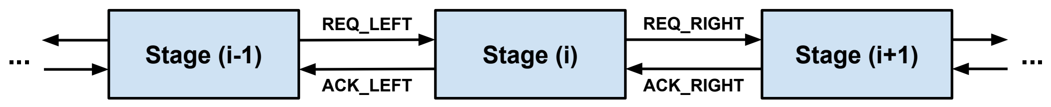


Fig. 8: C-element linear pipeline controller structure.

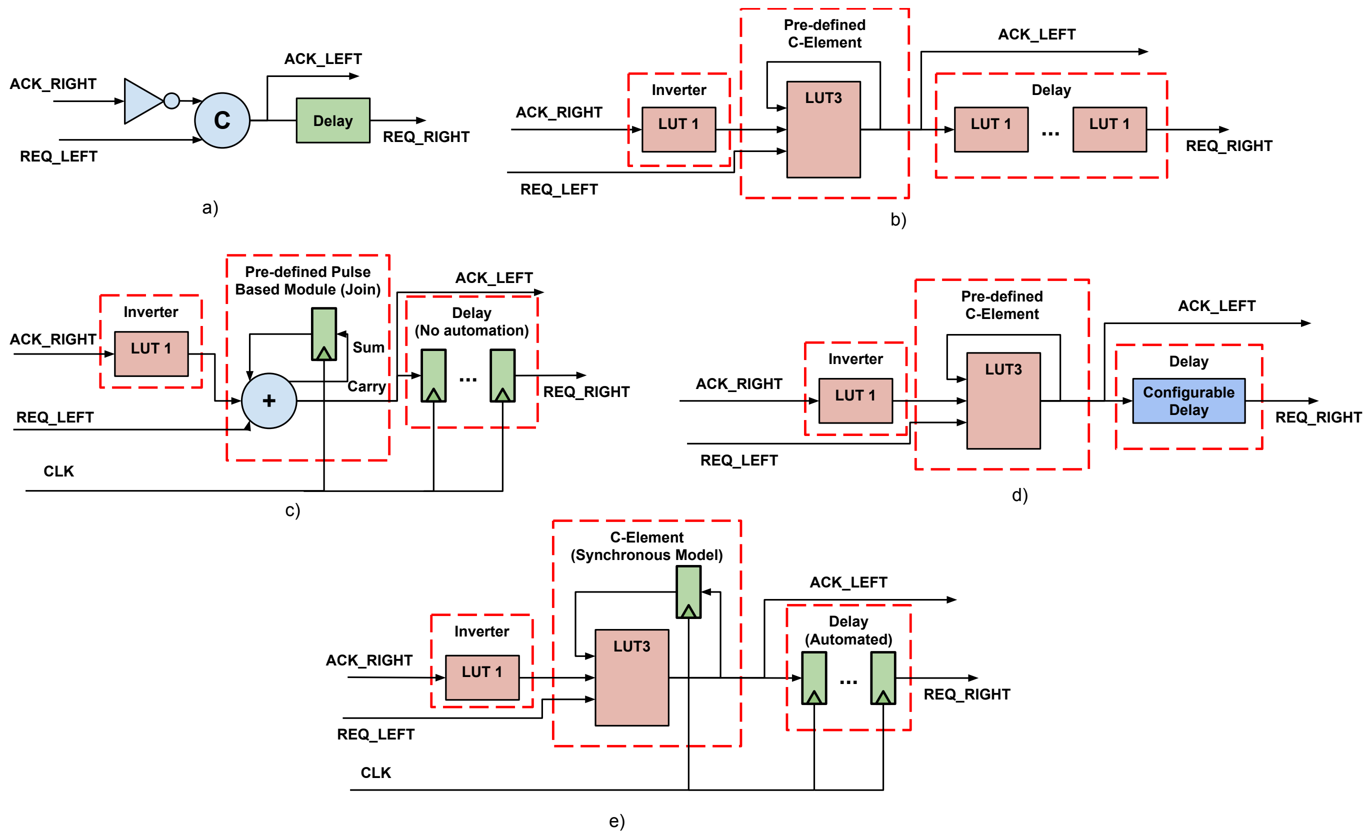


Fig. 9: A set of solutions for the asynchronous circuits implementation and FPGA mapping. (a) Muller Pipeline Stage Asynchronous Circuit Implementation; (b) Synchronous Implementation of a Muller Pipeline Stage with Pulses; (c) Synchronous Emulation of a Muller Pipeline Stage; (d) Asynchronous Implementation of a Muller Pipeline Stage; (e) Synchronous Simulation of a Muller Pipeline Stage.

GasP, Micropipelines etc.), column 4 shows whether predefined blocks are used, and column 5 shows whether this solution targets a specific FPGA vendor. Notably, our approach is the only one that can support a wide range of circuits in a vendor-neutral manner.

VI. Conclusion and Future Work

In this paper we proposed two approaches to the synchronous simulation of asynchronous circuits on commercially available FPGAs. We showed how to convert an asynchronous circuit description in ACT into a synthesizable and functionally equivalent Verilog model, as well as techniques to optimize resource utilization. Our solution supports several crucial implementation features, including timing constraints, arbiters, and distributed drivers. Both approaches demonstrate good performance results, however Option II is superior in terms of resource consumption. Although the performance results are relatively close there are cases when imbalance in combinational and sequential logic for Option II may lead to a significant drop in clock frequency. Thus, having two different solutions provides users with the flexibility to make area/performance trade-offs.

It should be noted that our benchmarks fit on one FPGA chip which in the real world scenarios of FPGA prototyping may not be the case. The most significant slowdown will occur when the circuit reaches maximum FPGA capacity, and requires interchip communication lines and multi FPGA mapping.

Our future work will be focused on the following problems. First, we will try to find a better way to reduce the number of flip-flops, so that it will not affect maximum frequency but still achieve good results in resource optimization. Second, we are exploring multi FPGA partitioning, which is a very important goal because of rapidly growing sizes of integrated circuits. At the same time it should be done while minimizing performance losses.

TABLE V: Comparison of the solutions flexibility for the asynchronous circuits mapping onto the synchronous FPGAs.

Project	Circuit Family	Control Topologies	Predefined Blocks	Vendor Specific
[19]	QDI, 2-phase bundled-data	2	Yes	No
[26]	4-phase bundled-data	1	No	No
[23]	2-phase bundled-data	1	Yes	Yes
[28]	4-phase bundled-data, Speed independent	2	Yes	Yes
[24]	2-phase bundled-data	2	Yes	Yes
[25]	QDI	1	Yes	Yes
[8]	2-phase bundled-data	1	Yes	Yes
[27]	4-phase bundled-data	1	No	Yes
[34]	2-phase/4-phase bundled-data	2	Yes	No
Our Solution	2-phase/4-phase bundled-data, QDI, rQDI, SDI, GasP, STFB	8*	No	No

*We note that although we demonstrate only 8 examples, our solution will work for all circuit which timing assumption can be specified using a timing fork.

References

[1] Rajit Manohar, “An open-source design flow for asynchronous circuits,” Government Microcircuit Applications and Critical Technology Conference, March 2019.

[2] ACT Framework. [Online]. Available: https://github.com/asyncvlsi/act/

[3] K. S. Stevens, R. Ginosar, and S. Rotem, “Relative timing,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 1, p. 129–140, Feb. 2003.

[4] D. M. H. Neil H. E. Weste, *CMOS VLSI Design A Circuits and Systems Perspective*, 4th ed. Pearson, 2011.

[5] A. Ghiribaldi, D. Bertozzi, and S. M. Nowick, “A transition-signaling bundled data noc switch architecture for cost-effective gals multicore systems,” in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 332–337.

[6] A. Lines, “Asynchronous interconnect for synchronous soc design,” *IEEE Micro*, vol. 24, no. 1, pp. 32–41, 2004.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[8] K. Bhardwaj, P. Mantovani, L. P. Carloni, and S. M. Nowick, “Towards a complete methodology for synthesizing bundled-data asynchronous circuits on FPGAs,” in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2019, pp. 1–6.

[9] I. E. Sutherland, “Micropipelines,” *Commun. ACM*, vol. 32, no. 6, p. 720–738, Jun. 1989.

[10] M. Singh and S. M. Nowick, “Mousetrap: High-speed transition-signaling asynchronous pipelines,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[11] A. Peeters, F. t. Beest, M. d. Wit, and W. Mallon, “Click elements: An implementation style for data-driven compilation,” in *2010 IEEE Symposium on Asynchronous Circuits and Systems*, 2010, pp. 3–14.

[12] A. J. Martin, *The Limitations to Delay-Insensitivity in Asynchronous Circuits*. New York, NY: Springer New York, 1990, pp. 302–311.

[13] C. LaFrieda and R. Manohar, “Reducing power consumption with relaxed quasi delay-insensitive circuits,” in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 217–226.

[14] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, “Titac-2: an asynchronous 32-bit microprocessor based on scalable-delay-insensitive model,” in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, 1997, pp. 288–294.

[15] I. Sutherland and S. Fairbanks, “Gasp: a minimal fifo control,” in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, 2001, pp. 46–53.

[16] M. Ferretti and P. A. Beerel, “Single-track asynchronous pipeline templates using 1-of-n encoding,” in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, 2002, pp. 1008–1015.

[17] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, “Cyclone: a static timing and power analysis engine for asynchronous circuits,” ASYNC, 2020.

[18] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[19] Q. T. Ho, J.-B. Rigaud, L. Fesquet, M. Renaudin, , and R. Rolland, “Implementing asynchronous circuits on LUT based FPGAs,” FPL, 2002.

[20] C. Pham-Quoc and A. Dinh-Duc, “Hazard-free Muller gates for implementing asynchronous circuits on Xilinx FPGA,” in *2010 Fifth IEEE International Symposium on Electronic Design, Test Applications*, 2010, pp. 289–292.

[21] J. N. Lassen, “FPGA prototyping of asynchronous networks on chip,” Master’s thesis, Technical University of Denmark, 2008.

[22] P. D. Ferguson, A. Efthymiou, T. Arslan, and D. Hume, “Optimising self-timed FPGA circuits,” in *Proc. Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 563–570.

[23] A. Mardari, Z. Jelcicová, and J. Sparsø, “Design and FPGA-implementation of asynchronous circuits using two-phase handshaking,” ASYNC, 2019.

[24] E. Brunvand, “Using FPGAs to implement self-timed systems,” *Journal of VLSI signal processing systems for signal, image and video technology*, no. 6, p. 173–190, 1993.

[25] Y.-F. E. Chang, R.-Y. Huang, and J.-H. R. Jiang, “Effective FPGA resource utilization for quasi delay insensitive implementation of asynchronous circuits,” ASYNC, 2019.

[26] H. Saito, N. Hamada, T. Yoneda, and T. Nanya, “A floorplan method for asynchronous circuits with bundled-data implementation on FPGAs,” in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 925–928.

[27] J. Furushima, M. Nakajima, and H. Saito, “Design of an asynchronous processor with bundled-data implementation on a commercial field programmable gate array,” *Informatica (Slovenia)*, vol. 40, 2016.

[28] A. Motaqi, M. Helaoui, S. Aghlimoghaddam, and M. R. Mosavi, “Detailed implementation of asynchronous circuits on commercial FPGAs,” *Analog Integrated Circuits and Signal Processing*, vol. 103, no. 3, p. 375–389, 2020.

[29] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for asynchronous circuits," *IEEE Des. Test. Comput.*, vol. 11, no. 3, pp. 60–69, Fall, 1994.

[30] R. Payne, "Self-timed field programmable gate array architectures," Ph.D. dissertation, University of Edinburgh, 1997.

[31] R. Manohar, "Reconfigurable asynchronous logic," in *Proc. Custom Integrated Circuits Conference*, 2006, pp. 13–20.

[32] J. V. Manoranjan and K. S. Stevens, "An a-FPGA architecture for relative timing based asynchronous designs," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–6.

[33] N. Huot and H. Dubreuil and L. Fesquet and M. Renaudin, "FPGA architecture for multi-style asynchronous logic [full-adder example]," in *Design, Automation and Test in Europe*, 2005, pp. 32–33 Vol. 1.

[34] J. O'Leary and G. Brown, "Synchronous emulation of asynchronous circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 205–209, 1997.

[35] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The vlsi-programming language tangram and its translation into handshake circuits," in *Proceedings of the European Conference on Design Automation.*, 1991, pp. 384–389.