Towards a Complete Methodology for Synthesizing Bundled-Data Asynchronous Circuits on FPGAs

Kshitij Bhardwaj, Paolo Mantovani, Luca P. Carloni, and Steven M. Nowick
Columbia University, NY, 10027
Email: {kbhardwaj, paolo, luca, nowick}@cs.columbia.edu

***Abstract*—Asynchronous circuits are gaining momentum as a promising low-power alternative to the conventional synchronous design approaches. In particular, single-rail bundled-data design style has seen significant interest both for designing GALS systems and in the emerging area of neuromorphic computing. However, there has been only limited research on implementing these asynchronous circuits on commercial FPGAs, which can be challenging due to the use of relative timing constraints in these designs for correct operation. This paper proposes a systematic CAD methodology to synthesize efficiently bundled-data asynchronous circuits on commercial FPGAs, achieving a two-fold goal for the target implementation: robustness and high performance. The methodology is targeted to the existing Xilinx Vivado tool set. As a case study, two asynchronous NoC switches are prototyped on Xilinx Virtex 7 in 28 nm: one supporting unicast, and the other also handling multicast. The former shows significant energy and idle power improvements, with some performance benefits, over a high-performance synchronous FPGA-based switch. The asynchronous multicast router also shows promising energy and performance results. Although a NoC case study is used, the proposed approach is general and can be used for other bundled-data asynchronous circuits.**

***Index Terms*—Asynchronous circuits, FPGAs, Synthesis CAD methodology, networks-on-chip**

I. Introduction

In the last two decades, asynchronous design is emerging as a promising alternative paradigm to address the challenges faced by the conventional synchronous approaches. Given the ever-increasing scale of integration, modern circuits are vulnerable to process variability, aging, chip power and thermal challenges, and scalability issues [1]. Asynchronous designs have several potential advantages: they are naturally energy-proportional (dissipating power only when active), do not require complex clock distribution, are highly modular, and can exhibit robustness to process- and environment-induced variability [1].

Recently, there has been a significant interest in designing asynchronous circuits using a *single-rail bundled-data* style [1] for higher performance, which employ an energy-efficient synchronous-style datapath. There are a number of applications of this design style: (i) high-speed asynchronous pipelines [2]; (ii) asynchronous networks-on-chip (NoCs) to efficiently connect different components in globally-asynchronous locally-synchronous (GALS) systems [3]; (iii) low-power asynchronous processors [4]; and (iv) large-scale commercial systems such as Philips' 80C51 microcontroller [5] and Intel's recent *Loihi* neuromorphic chip [6]. Several of these designs achieve considerable energy benefits over their synchronous counterparts [3], [4]. However, such designs also rely on relative timing constraints, in both datapath and control logic, for correct operation [1]. While the bundled-data design style is highly-efficient, it can also be challenging for synthesis CAD flows due to these timing requirements. In addition, there has been only limited work on synthesizing these designs on commercial FPGAs.

Contributions. The first contribution of this paper is a systematic CAD approach for synthesizing bundled-data asynchronous circuits on FPGAs. Unlike prior approaches, the methodology targets not only correct (i.e. hazard-free) operation, but also simultaneously incorporates systematic performance optimization techniques. Moreover, this methodology only uses the existing *Xilinx Vivado tool set,* leading to ease of implementation. *To the best of our knowledge, this is the first concrete and systematic methodology for efficiently mapping bundled-data asynchronous designs on modern FPGAs using existing synchronous CAD tools.*

The second contribution is a comprehensive guide on how to map some specialized asynchronous components on FPGAs. Such components are common in asynchronous systems, e.g. NoCs. These async-specific components are the C-element (for storage), the mutual exclusion element (i.e. mutex, for arbitration), and a 4-input arbiter. The elements are synthesized largely following the standard Vivado tool flow, with small manual interventions. Existing FPGA implementations of these elements are largely inefficient in terms of several cost metrics, and do not include a systematic mapping strategy. In contrast, we propose a step-by-step procedure, to achieve mappings with high performance and low resource utilization.

Finally, to demonstrate the effectiveness of the proposed methodology, two distinct recent bundled-data asynchronous NoC switches are synthesized on commercial FPGAs. The target FPGA is *Xilinx Virtex 7 in 28 nm.* One of these switches only supports unicast [7], while the other also handles multicast (1-to-many traffic) [8]. Compared to a high-performance unicast-only synchronous switch, the unicast asynchronous switch achieved promising results: 47% lower energy-per-packet and 75% lower idle power, along with some latency benefits. Interestingly, the asynchronous multicast switch showed similar energy as the synchronous switch for a unicast transmission with interesting performance advantages, despite the extra instrumentation for multicast.

II. Related Work

Early related works mostly have a narrow focus and only target synthesizing small asynchronous components and designs on FPGAs. Some of this prior research focuses on specialized components: C-element [9], or a mutual exclusion element [10]. The former only presents one possible implementation of the C-element, which uses 4 gates. In contrast, our work performs thorough exploration of various implementations and selects the most efficient and robust one. The latter mutex implementation is quite complex, consisting of 4 FFs and 2 gates, as opposed to a much more efficient

K. Bhardwaj is now a post-doctoral research fellow at Harvard University, Cambridge, MA, 02138 (kbhardwaj@g.harvard.edu).

This work was supported in part by the National Science Foundation, under grants CCF-1527796 and CCF-1764000

mutex implementation in this paper. Furthermore, the FPGAs and the synthesis tools used in these papers are now outdated; for example, these tools did not use any advanced performance optimization directives, which are now supported and critical for implementing high-performance asynchronous circuits.

There has been only limited research on synthesizing more complex asynchronous/GALS systems on commercial FPGAs. Earlier works target a quasi-delay insensitive (QDI) design style with only one simple timing assumption that all wire forks must be isochronic [1]: an RSA cryptography core [11] and asynchronous NoCs for GALS systems [12], [13]. The more recent GALS systems use bundled-data [14], [15]. The former QDI circuits use delay-insensitive data encoding [1], and therefore do not have any timing constraints but can incur large area/power overheads. These designs also use expensive FF- or latch-based mutex implementations. The latter bundled-data designs rely on non-trivial timing constraints, but an implementation methodology is largely missing.

Recently, two CAD flows have been proposed to map bundled-data circuits on FPGAs [16], [17]. Both flows have several limitations: (i) they do not address how to handle the special asynchronous components, which form a critical part of several circuits and systems such as NoCs; (ii) they focus only on correctness, while performance optimization, which is critical, is not targeted, in contrast, to the proposed approach; (iii) one approach requires a set of six specialized and custom tools, which are used in addition to the standard FPGA synthesis tool [16]. A preferable alternative is to only use the FPGA synthesis tool, as in the proposed work, exploiting the recent advances; and (iv) the second approach focuses only on *click-based asynchronous pipelines,* that employ high-overhead FF-based datapaths and control to simplify automation [17]. However, this flow may not be used for the majority of more efficient bundled-data designs based on lightweight Mousetrap pipelines that use normally-transparent single-latch registers with different timing constraints [3], [8].

III. Asynchronous Background

This section presents relevant background on common handshaking protocols, data encoding schemes, a Mousetrap pipeline which forms the basis of the targeted bundled-data circuits, and timing requirements for these circuits.

Handshaking protocols. Two common protocols are used to synchronize a sender and a receiver [1]: (i) *four-phase (RZ),* and (ii) *two-phase (NRZ).* In four-phase, the two control wires, *req/ack,* are initially low, and to complete a transaction, assertion on *req* causes an assertion on *ack,* followed by both deasserting in turn. In contrast, in two-phase, a single toggle on *req,* followed by a toggle on *ack,* completes one transaction. In this paper, a two-phase protocol is used for higher throughput: it only involves one roundtrip communication per transaction, rather than two roundtrips in four-phase.

Data encoding schemes. Two common data encoding schemes [1] for asynchronous communication are: (i) *delay-insensitive (DI) codes,* and (ii) *single-rail bundled-data.* In DI encoding, the *req* wire is replaced with data bits encoded so as to include their own validity, e.g., dual-rail or m-of-n codes. However, the focus of this paper is on single-rail bundled-data encoding, which typically has higher coding efficiency and lower power. This encoding has a synchronous-style data channel and an accompanying bundling *req* signal. For correct operation, a simple one-sided timing constraint is enforced: the *req* must transition only after the data is stable.

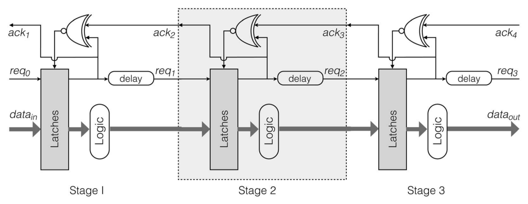


Figure 1: Implementation of a 3-stage Mousetrap pipeline

Mousetrap pipeline. Mousetrap is a widely-used high-performance asynchronous pipeline that uses a two-phase handshaking protocol and single-rail bundled-data encoding [18]. Figure 1 shows a 3-stage Mousetrap pipeline. Each stage consists of a single bank of normally-transparent D-latches with a simple local control. The interface between adjacent stages has single-rail data and a bundling req going forward and an ack going backward. Mousetrap operation is based on a *capture-pass protocol,* where the latches are initially transparent with all req/ack wires at 0. At a stage i, as new data arrives with its bundling req, it is passed through the latch and the corresponding req_i is toggled, causing the XNOR of that stage to close the latch, storing the data. In parallel, ack_i is sent to request new data. Finally, when an ack_{i+1} is received from right, the register becomes transparent, completing the entire cycle.

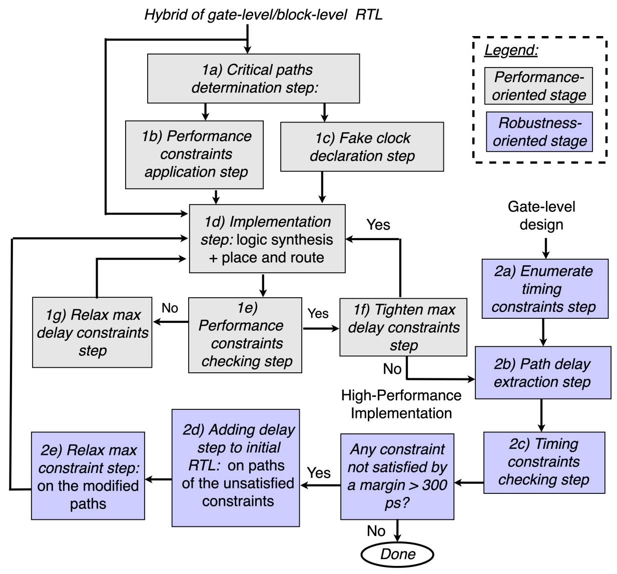
Timing requirements. To ensure correct operation of the bundled-data circuits, two types of timing constraints must be satisfied: (i) *datapath bundling constraints:* the *req* must transition after data is stable, and (ii) *relative timing constraints (RTCs)* in control: the delay across one path should be less or greater than the other. Mousetrap exhibits examples of both timing constraints: (a) *bundling constraint:* a delay element is added on the req to match the worst-case logic delay, (b) *data protection RTC:* once data enters a stage (e.g. stage 2), it must be securely stored in the latch register before new data is produced by the previous stage. Both of these constraints can be satisfied by adding small delays if needed.

IV. Methodology for Bundled-Data Circuits

We propose a systematic CAD methodology to synthesize bundled-data circuits on FPGAs, using the standard tool flow. This methodology not only targets a high-performance mapping of a single-rail bundled-data circuit but also ensures that it is robust. A design is first mapped focusing only on maximizing the performance, ignoring robustness, in a *performance-oriented stage.* Next, the flow enters a *robustness-oriented stage,* which takes a set of bundling constraints and RTCs, supplied by the user for the input design, and checks to make sure these constraints are satisfied. If a subset of these constraints is not satisfied, then to enable meeting these constraints, small incremental delay insertions are performed in the given design, on the offending paths. These two stages are iterated until all timing constraints are satisfied, and the result is an efficient and safe mapping.

Performance-oriented mapping stage. The gray boxes in Figure 2 show the steps of this stage. The synthesis tool tries to find the implementation with the best performance under some performance constraints. These constraints are selected based on the gates involved in the critical paths and an estimated delay of these paths in the target FPGA technology.

This stage takes a hybrid of gate-level/block-level RTL of a hazard-free design as input. Gate-level RTL, in addition with *DONT_TOUCH* directives, is used for parts of the design that explicitly require disabling of any logic manipulations that can



Hybrid of gate-level/block-level RTL
1a) Critical paths determination step:
1b) Performance constraints application step
1c) Fake clock declaration step
Legend:
Performance-oriented stage
Robustness-oriented stage
1d) Implementation step: logic synthesis + place and route
Yes
Gate-level design
2a) Enumerate timing constraints step
1g) Relax max delay constraints step
No
1e) Performance constraints checking step
Yes
1f) Tighten max delay constraints step
No
2b) Path delay extraction step
High-Performance Implementation
2e) Relax max constraint step: on the modified paths
2d) Adding delay step to initial RTL: on paths of the unsatisfied constraints
Yes
Any constraint not satisfied by a margin > 300 ps?
2c) Timing constraints checking step
No
Done

Figure 2: Proposed CAD methodology

introduce hazards. The rest of the input specification, on the other hand, can be block-level.

In Step 1a, *critical path determination* is performed to identify control and datapaths that govern latency (for example, the forward request and data in Mousetrap).

In Step 1b of *performance constraints application,* max delay constraints are imposed on all the critical paths using *set_max_delay.* These max delays are decided based on the gates involved and an estimate of the logic and wire delays for the target FPGA, starting with somewhat relaxed constraints. For example, in Mousetrap with simple computation logic (e.g. involving two 2-input gates on the critical path between the registers, with a path delay of $\sim 600ps$ on Virtex 7 in 28 nm), an initial max-delay constraint of 800 ps can be used.

Step 1c, *fake clock declaration,* is performed to overcome a tool flow limitation: max delay constraints can only be applied in a synchronous setting. For asynchronous designs, this issue is eliminated by using hypothetical clock boundaries, i.e, declaring the enable signals of the registers on critical paths as 'fake' clocks. For example, in Mousetrap, latch enables of all the registers must be declared as clocks using *create_clock.*

The gate-level RTL, max delay constraints, and the fake clock declarations are then used in the *implementation Step 1d* to perform logic synthesis, placement and routing.

The resulting implementation is checked to see if all the max-delay constraints are met in the *performance constraints checking Step 1e.* If all these constraints are satisfied, the methodology tries to find a mapping with higher performance: if possible, the performance constraints are tightened (*tighten max delay constraints Step 1f*), followed by re-implementation. However, if some constraints are not satisfied, the max delay constraints are relaxed in the *relax max constraints Step 1g,* and then the design is re-implemented. After each re-implementation, the performance constraints are again checked. Therefore, an iterative process achieves a high-performance implementation with all the max delay constraints satisfied. For example, for the previous Mousetrap case with simple computation logic in each stage, Step 1f can be used to tighten the initial max-delay constraint from 800 ps to 700 ps, which is still satisfied after synthesis. A further tightening to 580 ps may not be satisfied, and therefore this constraint needs to be relaxed to around 630 ps (Step 1g), which is then satisfied and also results in a high-performance implementation.

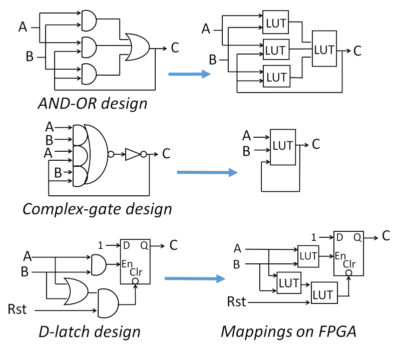


Figure 3: C-elements: standard-cell designs and FPGA mappings

Robustness-oriented mapping stage. The blue boxes in Figure 2 show the steps involved. The implementation from the previous stage may not satisfy all the timing constraints, and therefore a robustness-oriented stage is required.

First, as a pre-processing Step 2a, all the bundling and relative timing constraints are determined in the initial gate-level RTL design in a *timing constraints enumeration step.*

Next, in Step 2b of *path delay extraction,* the delays across all the paths involved in the timing constraints are extracted from the previous implementation, using *get_timing_paths.*

These path delays are then used to check if all the timing constraints are satisfied by a margin of more than 300 ps, in a *timing constraints checking step* (Step 2c). For a Virtex 7, fabricated in 28 nm, 300 ps is equivalent to adding a buffer LUT delay, and can be considered a safe margin. If all the constraints are met then no further action is required.

However, if a subset of constraints is not satisfied then Step 2d of *adding delay* is used. The offending paths are slowed down by adding localized delay in the initial gate-level RTL, keeping the rest of the design unmodified. These delays are composed of buffer LUTs, and picked based on the differences by which the constraints are not satisfied.

Since, an extra delay is added, max delay performance constraints for these paths are slightly relaxed in Step 2e of *relax max delay constraints.*

The updated gate-level RTL is then re-synthesized using the performance-oriented stage, followed by again checking if all the timing constraints are satisfied in the robustness-oriented stage. The two stages are repeated until all the timing constraints are satisfied.

V. Synthesis of async-specific components

We present the synthesis of the asynchronous elements on FPGAs: the C-element, the mutex, and a 4-input arbiter.

A. C-Element

A C-element is an asynchronous storage component [1]. It has two inputs A, B and one output C, which is asserted high if both inputs are high, deasserted low when both inputs are low, otherwise the output is held. In this work, we explore three different standard-cell designs of the C-element for mapping on to FPGAs, and the best design in terms of performance, resource utilization and robustness is selected.

Figure 3 shows the three designs for a C-element: using AND/OR gates, AOI222 complex gate, and a D-latch based design. Since, the C-element is an asynchronous state machine, the two combinational designs require the output of the C-element to be fed back to provide state information. In contrast, the latch-based design does not require any feedback,

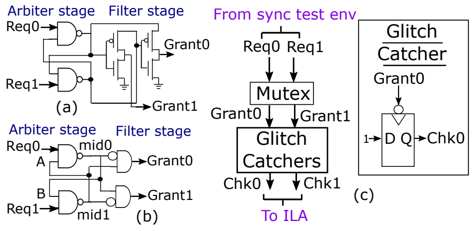


Figure 4: Mutex: (a) analog, (b) digital, (c) validation

but it can be very slow. However, the former combinational designs require an extra timing constraint for glitch-free operation: after any change in the output, the feedback input must arrive before the next change on the primary inputs.

All three designs are synthesized on the FPGA using the standard FPGA synthesis tool flow. For each design, a gate-level RTL model, with *DONT_TOUCH* directives to disable any logic manipulations, is the input to the tool, which is then used to perform logic synthesis, placement and routing.

Figure 3 shows the FPGA mappings obtained for each of the three C-element designs. The FPGA considered is Xilinx Virtex 7 in 28 nm. The AND-OR design leads to a mapping with 4 LUTs and has a post place-and-route latency of 434 ps. The complex-gate design, on the other hand, only uses 1 LUT and has a latency of just 43 ps. The latch-based design is the most expensive, and uses 3 LUTs and 1 FLOP with a latency of 778 ps, which is large due to the FLOP delay of around 330 ps. Therefore, in terms of latency and resource utilization, the complex-gate design that leads to a 1-LUT mapping is the best. This design is also very robust as the feedback wire delay is small (159 ps), and will arrive significantly faster than the new primary inputs after any change in the output.

B. Mutex

A mutex performs arbitration between two asynchronous requests, and therefore faces unique challenges for its correct and efficient implementation on FPGAs. While a synchronous arbiter arbitrates based on input arrival order per discrete clock cycles, an asynchronous mutex must resolve arrival order in continuous time. The correct mutex is designed using analog circuits [1], as shown in Figure 4(a). However, since an analog implementation is not possible on FPGAs, a digital mutex, with some inherent mean time between failure (MTBF), is used for mapping.

Structure and operation of the digital mutex. As shown in Figure 4(b), the digital mutex has two stages: an arbiter and a filter [19]. The former performs arbitration between the two incoming requests, designed using cross-coupled NAND gates. The latter is used to keep both the grant outputs deasserted while the arbitration is being resolved, and only after the arbitration is complete, it allows exactly one of the grant outputs to be cleanly asserted high. When two requests arrive close to each other, the arbiter can become metastable, with the internal mid values exhibiting either oscillations or intermediate voltage levels. The goal of the filter is to safely block this internal activity. This filter is designed using ANDs.

Requirements for correct and high-performance operation of the digital mutex. In the digital mutex, the focus for achieving high performance is on the arbiter stage. A simple timing requirement must be satisfied to enable quick blocking of a competing request, leading to fast resolution time. To this end, the wire delays of the two feedback wires, i.e. *mid0* to *A* and *mid1* to *B,* must be small.

The focus of the correctness issue is on the filter stage. Contrary to the analog mutex, where the filter stage performs a stable masking of any internal glitches while resolving arbitration, the filter stage of the digital mutex is unstable and can get transiently disabled during the internal oscillations. Therefore, to improve its filtering capability, an electrical intervention in the form of large inertial delay is required at the inputs of the filter stage. To this end, large capacitances can be used at the inputs of the filter.

Mapping of digital mutex on FPGAs. The above requirements are used to guide the mapping. These requirements can be met by manually forcing the placement of the mutex's components on certain locations on the FPGA. First, each gate of the mutex must be mapped to a LUT. For fast arbitration resolution, both the arbiter LUTs should be placed symmetrically in the same CLB such that the wire delays from *mid1-A* and *mid0-B* will be small. For correctness, the filter stage must have a high input capacitance. Since, no such high-capacitive elements exist on FPGAs, long wires can be used between the arbiter stage and the filter as such wires can exhibit high capacitance [20]. Hence, the filter LUTs must be placed in the CLB next to the arbiter CLB.

The implementation procedure for the digital mutex largely uses the standard synthesis flow. The Xilinx Vivado tool takes two inputs: (i) a gate-level RTL model with *DONT_TOUCH* directives forcing each gate to be mapped to a LUT, and (ii) a set of constraints to manually force these LUTs to be placed at the above discussed locations on the FPGA. The tool then performs logic synthesis, place and route.

The final mutex implementation operates with equalized paths and high performance. In this implementation, the wire delays between the cross-coupled LUTs of the arbiter (*mid0-B* and *mid1-A* delays) are small and almost the same for fast arbitration resolution: $\sim 114ps$ on Xilinx Virtex 7 in 28 nm. Also, longer wires are used between the arbiter and the filter stage for high-capacitive filtering, with delays of $\sim 350ps$. The resulting mapping is also balanced with a latency of $\sim 445ps$ between each req and the corresponding grant.

Validation of mapped mutex on FPGA. Since a digital mutex is used, it is important to stress test its final implementation to check its reliability. Figure 4(c) shows the setup used to validate the mapped mutex using a synchronous test wrapper during emulation. Two extreme cases are considered: requests arriving simultaneously and near simultaneously (3 ps apart). The outputs of the mutex are checked to see if the arbitration is resolved cleanly without glitches. A specialized FF-based *glitch catcher* is used on each output, since glitches may be short transients (oscillation period $\sim 300ps$) and cannot be directly detected by the logic analyzer as its sampling frequency is small. If a glitch occurs while resolving arbitration, the glitch catcher output, which is normally reset, will become 1, which can then be detected by the ILA during its sampling window. An exhaustive testing was performed by running 10000 samples of each of the above two cases: no glitches were observed and the mutex was shown to perform correctly. Since such extreme cases are rare in a real system, this test gives strong confidence that the MTBF of the mapped design will be very high for more realistic traffic.

C. 4-Input Arbiter

The arbiter takes four input requests and has a grant output corresponding to each request. This design uses three mutexes in parallel for high performance [7].

A hierarchical approach is used to synthesize the arbiter. Each of the three mutexes is first synthesized in isolation, followed by synthesizing the remaining design. The implemented mutexes are placed and locked side-by-side on the FPGA. The remaining design is synthesized using largely the standard flow but with small manual interventions to optimize the latency. The Xilinx Vivado tool is used to synthesize the remaining arbiter. The tool receives two inputs: (i) a gate-level RTL of the remaining arbiter, with *DONT_TOUCH* primitives, forcing each gate to be mapped to a LUT, and (ii) a set of constraints to force the LUTs on the critical paths, such as the ANDs and the C-elements, to be placed at specific locations to achieve small wire delays between these LUTs. Finally, logic synthesis, place, and route are performed. The resulting implementation is high-performance and robust. It also achieves very balanced latencies between the requests and their corresponding grants: 1.8 ns, 1.8 ns, 1.7 ns, and 1.9 ns, respectively on Xilinx Virtex 7 in 28 nm. This implementation inherently satisfies all the timing constraints.

VI. Case Study: Asynchronous NoC Routers

To demonstrate the effectiveness of the proposed methodology, we synthesize two recent asynchronous 5-port routers on FPGAs. Both routers are highly-efficient: one only supports unicast [7] while the other also handles multicast [8].

Unicast-only and multicast asynchronous routers. Each router has two main components: input port modules (IPMs) and output port modules (OPMs) [7], [8]. Each IPM buffers the packets in a circular buffer, performs route computation and selects the correct output port, while each OPM arbitrates between packets from the four IPMs (using a 4-input arbiter) and selects the winner for the output channel. The multicast router's IPM differs from the unicast IPM: (i) the former performs route computation on the header in parallel with its buffering as opposed to serial operations in the latter; (ii) it supports more complex address decoding stage than the unicast IPM; and (iii) also handles parallel replication capability. However, both the routers use the same OPM for arbitration. These IPMs/OPMs are based on Mousetrap.

Synthesizing asynchronous routers on FPGAs. A hierarchical approach is used to synthesize the asynchronous routers on FPGAs. The arbiters of a router's OPMs are first synthesized in isolation, as in Section V-C, and then placed and locked on the FPGA, followed by synthesizing the remaining router design using the methodology from Section IV.

The performance-oriented stage for the router follows the the gray boxes in Figure 2. A gate-level RTL of a router is input, with all the arbiters in the RTL kept as black boxes. Step 1a extracts all critical data/control paths for the header and body latencies from each IPM to each OPM. In Step 1b, *set_max_delay* constraints are applied to all these paths. In parallel, Step 1c declares the enable signals of all the registers in the input buffers and the OPMs as 'fake' clocks. Next, the *implementation Step 1d* is performed. The resulting implementation then undergoes an iterative process (Steps 1d-1g) to find a high-performance mapping.

The robustness-oriented stage follows the blue boxes in Figure 2. First, all the bundling and RTCs are enumerated in Step 2a. Next, the high-performance implementation from the previous stage is input to the Step 2b of *path delay extraction.* Step 2c then checks if all timing constraints are satisfied by more than 300 ps. In the router implementation, all the RTCs were met, but not all the bundled-data constraints. So, Step 2d is used to add appropriate delay lines to the request paths of these unsatisfied constraints. Finally, in Step 2e, the max delay constraints for these paths were slightly relaxed, followed by re-implementation using the performance-oriented stage. Both stages are repeated until all the timing constraints are met.

The final implementations were stress-tested using different traffic patterns, e.g. uniform and hotspot, and various packet sizes. Both routers were shown to operate correctly.

VII. Experimental Results

The two synthesized asynchronous switches are compared with a state-of-the-art synchronous switch.

Setup. Three final prototypes are compared: (i) asynchronous unicast switch (*Uni-Async*), (ii) asynchronous multicast switch (*Multi-Async*), and (iii) a high-performance single-cycle synchronous unicast switch (*Uni-Sync*). The latter also includes aggressive optimizations such as lookahead routing, which were not used in the asynchronous switches. The synchronous switch has been used in efficient accelerator-based systems [21]. These switches use the same 34-bit datapath with no virtual channels. Each input port is buffered with a FIFO queue of depth 5. The target FPGA is Xilinx Virtex 7 in 28 nm. All evaluations are at post place-and-route level.

CAD Methodology convergence. For both the *Uni-Async* and *Multi-Async* routers, the methodology converged rapidly with only 3 runs of each of the performance- and robustness-oriented stages. In particular, in the performance stage, the tool flow is able to handle 4615 and 5040 max-delay constraints for the two routers very efficiently: implementing these designs (Step 1d in Figure 2) in just 6 and 8 mins, respectively, during each iteration. In the first run of performance stage, the number of internal optimization iterations (Steps 1d-1g) performed are 12 and 15, respectively for the two routers. These iterations are required as a large number of initial max-delay constraints are tightened/relaxed for best performance. The first run of the robustness stage identifies only 10 unsatisfied bundled-data timing constraints for each of the two routers, and tries to satisfy them using Step 2d. The next run of the performance stage only involves 4 internal iterations, for both the routers, as max-delay constraints corresponding only to these timing constraints are perturbed. The second run of the robustness stage yields 5 unsatisfied bundled-data constraints, which results in only 3 internal iterations for the performance stage run, followed by satisfying all the timing constraints by the robustness stage for both the routers.

Resource utilization. Figure 5(a) shows the number of LUTs/FLOPs used on the FPGA. *Uni-Async* takes 28% more LUTs and 15.7% more FLOPs than *Uni-Sync*. However, the majority of the FLOPs in *Uni-Async* are used as single latches (1120/1210), while the *Uni-Sync* only uses FFs; the former can lead to better overall performance and reduced switching energy. Further, *Multi-Async* uses considerably more number of LUTs/FLOPs than the others, which is expected due to the extra instrumentation required for parallel multicast.

Latency. Figure 5(b) shows the switch latency results for routing the header and the body flits. While the synchronous switch shows a fixed latency for each flit, the asynchronous switches can have different latencies.

Uni-Async shows 75% longer header latency, but achieves 30.7% lower body latency than *Uni-Sync*. For the header, *Uni-Async* performs all critical operations in series: buffering, route computation and arbitration, while *Uni-Sync* performs all these operations in parallel, and hence achieves better latency.

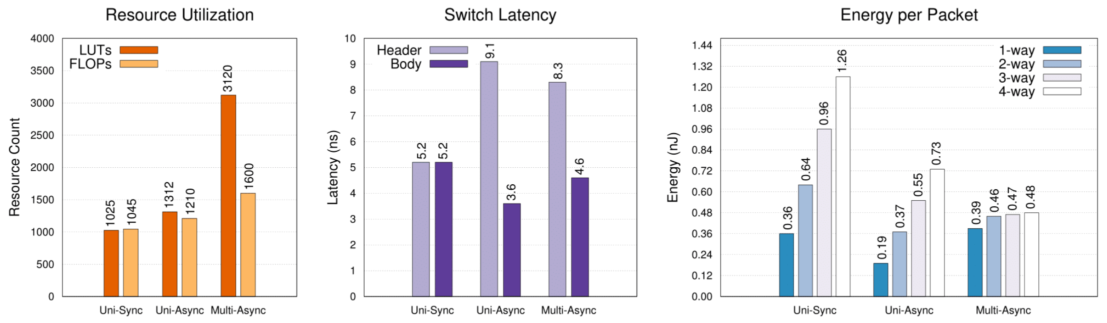


Figure 5: Asynchronous vs. synchronous switch prototypes: (a) resource utilization, (b) switch latency and (c) energy per packet

However, improved header latency results are expected for the asynchronous switch after incorporating optimizations such as lookahead routing. On the other hand, for body flits, the routing of the header through the asynchronous switch pre-allocates the path. The remaining body and tail flits are then fast forwarded using this much simpler pre-allocated path. This body-latency benefit can lead to major system performance improvement, particularly for loosely-coupled accelerators that use long packets (1000s of body flits) [22].

Multi-Async showed 59.6% higher header latency than *Uni-Sync* but still achieved 11.5% lower body latency. For the header, *Multi-Async* performs the buffering and route computation operation in parallel, which leads to 8.8% lower latency than *Uni-Async*. Moreover, for body flits, fast forwarding through a simple pre-allocated path in *Multi-Async* leads to 11.5% latency savings over the synchronous switch.

Energy per packet. Figure 5(c) shows the energy-per-packet results for the three switches. In this study, four different transmission scenarios are considered: one unicast (1-way) and three multicast (2/3/4-way). The packets consist of 5 flits. In the absence of multicast support, *Uni-Async* and *Uni-Sync* switches serially inject and route multiple unicast copies for each multicast packet, while *Multi-Async* routes a single multicast packet through multiple output ports in parallel.

Uni-Async achieves significantly lower energy-per-packet, in the range of 42% (4-way) to 47.2% (1-way), than *Uni-Sync* due to the absence of clock. Even though *Multi-Async* is more complex than *Uni-Sync*, it still has almost the same energy under a unicast transmission, and achieves considerably lower energy (28.1-61.9%) for multicast transmissions. For the latter, the absence of clock energy and the parallel routing of a multicast packet in *Multi-Async* leads to lower energy.

Idle power. In the absence of any routing activity also, as expected, the asynchronous switches achieve 75% and 25% lower idle power than the synchronous switch, respectively, due to the absence of any clocking activity in the former (*Uni-Async:* 2 mW, *Multi-Async:* 6 mW, *Uni-Sync:* 8 mW).

Interestingly, some of the above FPGA results show similar trends to those in our recent industrial ASIC comparison with an AMD synchronous NoC switch [3]: the latter demonstrated 58% active power savings and 28% latency improvement (on header flits). However, a more detailed ASIC-to-FPGA comparison is not yet possible, given the different switch architectures (the latter included VC's and multi-plane switches).

VIII. Conclusions and Future Work

This paper proposes a systematic CAD methodology for synthesizing bundled-data asynchronous circuits on commercial FPGAs, using only the existing tools. As a case study, two asynchronous NoC switches are synthesized on Xilinx Virtex 7 in 28 nm, where one only supports unicast and the other also handles multicast. The former achieved significant energy and idle power improvements, with some performance benefits, over a high-performance synchronous switch. As future work, these asynchronous switches will be used in a complete instantiated GALS multicore system on FPGAs. We will also explore the tradeoffs in the tool flow of selectively disabling performance- and robustness-driven optimization, to see the contribution and impact of each optimization phase.

References

[1] S. M. Nowick and M. Singh, "Asynchronous design - part 1: overview and recent advances," *IEEE Design & Test*, vol. 32, no. 3, pp. 5–18, 2015.

[2] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *ASYNC*, 2001, pp. 46–53.

[3] W. Jiang *et al.*, "An asynchronous NoC router in a 14nm FinFET library: comparison to an industrial synchronous counterpart," in *DATE*, 2017, pp. 732–733.

[4] D. Bhadra and K. S. Stevens, "Design of a low power, relative timing based asynchronous MSP430 microprocessor," in *DATE*, 2017, pp. 794–799.

[5] H. van Gageldonk *et al.*, "An asynchronous low-power 80C51 microcontroller," in *ASYNC*, 1998, pp. 96–107.

[6] M. Davies *et al.*, "Loihi: a neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, pp. 82–99, 2018.

[7] A. Ghiribaldi, D. Bertozzi, and S. M. Nowick, "A transition-signaling bundled data NoC switch architecture for cost-effective GALS multicore systems," in *DATE*, 2013, pp. 332–337.

[8] K. Bhardwaj and S. M. Nowick, "A continuous-time replication strategy for efficient multicast in asynchronous NoCs," *TVLSI*, vol. 27, no. 2, pp. 350–363, 2019.

[9] Q. Thai Ho *et al.*, "Implementing asynchronous circuits on LUT based FPGAs," in *FPL*, 2002, pp. 36–46.

[10] S. W. Moore and P. Robinson, "Rapid prototyping of self-timed circuits," in *ICCD*, 1998, pp. 360–365.

[11] J. J. H. Pontes *et al.*, "SCAFFI: an intrachip FPGA asynchronous interface based on hard macros," in *ICCD*, 2007, pp. 541–546.

[12] X. Wang, T. Ahonen, and J. Nurmi, "Prototyping a globally asynchronous locally synchronous network-on-chip on a conventional FPGA device using synchronous design tools," in *FPL*, 2006, pp. 1–6.

[13] J. Quartana *et al.*, "GALS systems prototyping using multiclock FPGAs and asynchronous network-on-chips," in *FPL*, 2005, pp. 299–304.

[14] J. Lassen, "FPGA prototyping of asynchronous networks-on-chip," *M.Sc. thesis, DTU, Kongens Lyngby*, 2008.

[15] H. Katabami, H. Saito, and T. Yoneda, "Design of a GALS-NoC using soft-cores on FPGAs," in *MCSOC*, 2013, pp. 31–36.

[16] K. Takizawa, S. Hosaka, and H. Saito, "A design support tool set for asynchronous circuits with bundled-data implementation on FPGAs," in *FPL*, 2014, pp. 1–4.

[17] J. Zhang *et al.*, "From click based asynchronous design to Xilinx FPGA," in *ASYNC*, 2018.

[18] M. Singh and S. M. Nowick, "MOUSETRAP: high-speed transition-signaling asynchronous pipelines," *TVLSI*, vol. 15, pp. 684–698, 2007.

[19] R. Ginosar, "Handshake circuit implementations: slide 13," *http://slideplayer.com/slide/4906671/*, 2009.

[20] G. Lemieux *et al.*, "Directional and single-driver wires in FPGA interconnect," in *FPT*, 2004, pp. 41–48.

[21] L. P. Carloni, "Invited - The case for embedded scalable platforms," in *DAC*, 2016, pp. 17:1–17:6.

[22] P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "High-level synthesis of accelerators in embedded scalable platforms," in *ASP-DAC*, 2016, pp. 204–211.