Interleaved Architectures for High-Throughput Synthesizable Synchronization FIFOs

Ameer M. S. Abdelhadi and Mark R. Greenstreet
University of British Columbia
{ameer@ece.ubc.ca, mrg@cs.ubc.ca}

***Abstract*—This paper presents a family of FIFOs for clock-domain crossings. These designs are distinguished by an interleaved architecture for the control and data-paths. This approach eliminates most of the throughput bottlenecks in the FIFO design, allowing operation at well over 1GHz in a 65nm process using a standard ASIC design flow. Furthermore, these designs are low-latency: the fall-through time for an empty FIFO is only a few gate delays greater than the synchronizer latency. Our designs are fully synthesizable using widely available design libraries. Furthermore, we identify a glitch vulnerability that is lurking in many published designs, and describe our solutions to these hazards.**

I. Introduction

Current chip designs are partitioned into multiple clock domains and can involve large numbers of clock domain crossings. Clock domain crossings (CDCs) can involve interfaces between IP blocks in a SoC or NoC design, interfaces between CPU cores and caches, and interfaces to external, high-speed links. These interfaces often require high bandwidth and low latency. CDC designs should be robust so they do not cause metastability failures on the chip, and they should be synthesizable to work with industry standard design flows. FIFO based designs are commonly used because they can achieve bandwidths that are set by the slower of the sender and receiver, and they allow a decoupling of timing constraints between the two domains, greatly simplifying chip-level timing closure [1], [2].

The two most common approaches to CDC FIFOs are based on Gray code counters [3] and one-hot designs [4]. The typical Gray code design uses a dual-port SRAM, where the sender and receiver each have Gray code counters to address their respective ports. The main limitation of the Gray code design is that efficient comparison of the read and write pointers requires conversion back to ordinary binary, and the combinational logic for the conversion and comparison limits the operating frequencies. The access times for the SRAM are another throughput limiter. In contrast, the one-hot designs have fast control logic. Typically, data is stored in latches or flip-flops to provide a correspondingly fast data path. A serious drawback of the one-hot design is that there is a separate stage of control logic for each word of storage along with synchronizers between the sender's and receiver's clock domains for each stage. The control circuitry for one-hot FIFOs can readily dominate the total area.

We present an *interleaved* variant of the one-hot design. The key observation is that a N-stage FIFO can be organized as a two-dimensional, $N_v \times N_h$ array, where $N = N_v N_h$. The read and write interfaces each have a pair of one-hot counters: one that counts modulo N_v; and the other modulo N_h. This retains the speed of the one-hot design while reducing the area for the counters to the point that data storage dominates FIFO area, as in a Gray code design. We also apply interleaving to the data paths. In particular, using separate data buses for reading odd- and even-indexed data latches provides an extra-cycle for the read-path enabling high-throughput operation. Section III presents the FIFO architecture in more detail.

In the course of designing the FIFO, we discovered a glitch hazard similar to the ones described in [5]. This hazard appears in many previously published designs. Section III.3 describes the hazard and our solutions.

Our FIFO is fully synthesizable using only logic functions that are present in typical cell libraries. The FIFOs are highly configurable, and the design includes support for simulation, static timing analysis, and a testbench for regression tests of the design. The complete design is public and open source with the BSD license [6]. Section IV summarizes the capabilities of this open-source package. Section V presents throughput, latency, and other metrics for FIFOs generated with our open-source Verilog and standard design tools using a cell library for a 65nm process. Our FIFOs achieve throughputs that exceed typical clock frequencies for ASIC design flows. The fall-through latency for an empty FIFO is the synchronizer latencies plus one cycle for glitch safety.

II. Related Work

Increasing chip densities have led to a rapid increase in the number of clock domain crossings that occur in a single design. FIFOs are attractive for such interfaces because they offer high throughput and simple flow control. Synchronizing FIFOs are distinguished largely by the design of the put and get control logic, the implementation of the data store, and the synchronization mechanisms between the put and get interfaces.

The most common synchronizing FIFOs are based on Gray code counters [1], [3]. The advantage of a Gray code is that on a clock transition, exactly one bit of the counter makes a transition. If the put-controller uses a Gray code for its write-pointer, then the bits of the pointer can be synchronized to the get-controller using a separate synchronizer for each bit. Because at most one bit will be changing at a `clk_get` edge, at most one synchronizer will enter metastability. When that bit resolves, the synchronizer outputs either the "before" or "after" value of the write-pointer. Either is valid. The disadvantage of Gray codes is the difficulty of comparing two Gray code values to determine which is greater. Typical designs convert the Gray code value to standard binary, and then perform the comparison. The conversion requires a chain of XOR gates whose length is the number of bits in the pointer (minus one). This tends to be a slow operation that limits FIFO performance.

An alternative to Gray code pointers is to use some kind of unary encoding [4], [7], [8]. Such FIFOs offer very high throughputs because the counters are fast, and comparing unary values is easy. However, FIFOs with unary control suffer from a large flip-flop count. In particular, an N-stage FIFO requires $2N$ synchronizers, N for the put-to-get synchronization, and N for get-to-put. Each synchronizer consists of multiple flip-flops, and more flip-flops are needed to implement the unary counters and other state such as per-stage full-empty status. The area and power for such designs dominated by the control logic. While the performance is attractive, we seek to mitigate the large flip-flop counts.

In addition to Gray code and unary counters, many other designs have been proposed. Keller [9] presents a novel implementation for GALS applications based on "pointer-increment" signals. Keller's design uses mutex elements to arbitrate between communication and clock generation; because metastability is rare and usually resolves quickly, Keller's design, like most pausible clock designs, achieves very low latency for cross-domain communication. While we note a growing interest in pausible clock GALS (e.g. [1], [9]), the most common clock-domain-crossing designs remain synchronous-to-synchronous, and we focus on that scenario here.

Another approach to synchronization is to use a ripple FIFO instead of a pointer based design. Seizovic [10] showed synchronization can be incorporated into the control path of a ripple FIFO. More recently, Jackson and Manohar [11] showed a generalization of Seizovic's scheme where some pipeline processing can be done along the datapath of the FIFO while the control path accomplishes synchronization. These are clever designs, but they use special handshaking cells that are not amenable for standard synthesis flows.

Finally, there has been extensive work on mesochronous designs where the sender and receiver operate at identical clock frequency with an unknown phase relationship. Examples include [12]–[16]. When the communicating clock domains have a common source, these methods offer excellent performance and efficiency. We are addressing the more general, and common case, where either there is no common source to the clocks, or where it is impractical to bound the variation in the skew under operation. We note that [16] provides an excellent survey of prior work in clock-domain crossing that transcends the space limitations of this paper.

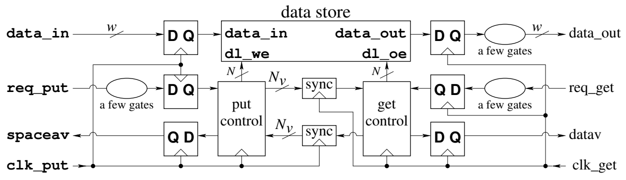

Fig. 1. The Synchronizing FIFO

III. The FIFO Architecture

Fig. 1 shows the structure of our synchronizing FIFO. The "put interface" operates synchronously with the sender's clock, `clk_put`, and the "get interface" operates synchronously with the receiver's clock, `clk_get`. The signal `spaceav` indicates that the FIFO has space available and that the sender may insert a data word of w bits into the FIFO by asserting `req_put`. Likewise, `datav` indicates that a valid data word is available for the receiver at `data_out` and can be removed by asserting `req_get`. The flip-flops in Fig. 1 indicate the clock domains for each signal.

Data is stored in an array of latches. The put-control uses a pair of ring counters to generate the write-enable signals for these latches. Likewise, the get-control uses a pair of ring counters to generate data latch output-enable signals. By using two counters, the FIFO has a capacity set by the product of the counter lengths. This leads to a logical organization of the FIFO as an "array" of rows and columns. We show later in this section, that it is sufficient to synchronize empty and full information on a per-row basis. This dramatically reduces the number of synchronizers needed in the FIFO compared with other designs based on unary-counters. In addition to simplifying synchronization, organizing the FIFO as an array allows many internal operations to be performed in an interleaved fashion. This interleaving is crucial for the performance of our design.

Another critical feature of our design is that we explicitly prevent the "flow-through" path that is common in other synchronizing FIFOs. In many designs, the value of `data_out` can change *asynchronously* with respect to `clk_get` when the FIFO is empty (and therefore, `data_valid` is false). While a designer might expect such ill-defined values to be ignored, aggressive optimizations performed by logic synthesis can introduce glitch failures. Section III.3 describes our solution to this glitch problem.

III.1. The Control Path

The put- and get-control blocks are very similar. This section describes the put-control in detail noting a few details that are specific to the get-control. The put-control performs three functions: it manages the enables for the data latches to store values from the sender; it notifies the get-interface (through the synchronizers) of available data; and it notifies the put-client when space is available to insert a value into the FIFO.

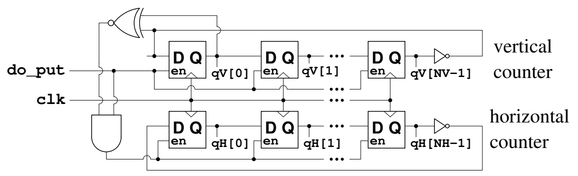

Fig. 2. Ring counters in the put-control

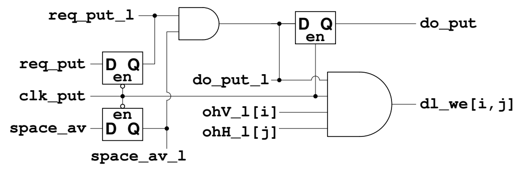

Fig. 3. Latch and Counter Control

To insert a word into the FIFO, the sender must present the data at data_in and assert req_put satisfying the set-up and hold requirements of the input flip-flops (see Fig. 1). If spaceav was asserted on the same cycle as req_put, then the word will be recorded on the rising edge of put_clk and stored in the FIFO. If req_put is asserted on a cycle when spaceav is false, the put request is simply ignored. This allows the sender to eagerly attempt inserts and use spaceav as a confirmation that the insert succeeded. The sender can persist with attempts to insert a value until the insert operation succeeds. The maximum throughput of one data word per cycle can be achieved by holding req_put high as long as there is space available.

Each control block is based on a pair ring counters, the "vertical" and "horizontal" counters as shown in Fig. 2. Logically, we can think of the vertical counter as specifying which of N_v smaller FIFOs of N_h stages is selected for the next put operation, and these smaller FIFOs are written in a round-robin fashion. We refer to the logic and data-store for each value of the vertical counter as a "row" even though the physical layout is unconstrained (and thus place-and-route finds a roughly linear arrangement). A reset initializes the counters $\mathtt{qV} = \mathtt{qH} = 0$. We require N_h and N_v to be even. The counters implement a "thermometer code" that we interpret as an integer count as shown below:

$$\mathsf{count}(\mathtt{q}) = \begin{cases} 0, & \text{if } \mathtt{q}[n-1] = \mathtt{q}[0] \\ i, & \text{if } \mathtt{q}[i] \neq \mathtt{q}[i-1] \end{cases} \tag{1}$$

where n is the number of stages in the ring counter. Note that negating all of the bits of q does not change $\mathsf{count}(\mathtt{q})$. The numeric value for the pair of counters is

$$\mathsf{count}(\mathtt{qH},\mathtt{qV}) \;=\; \mathsf{count}(\mathtt{qV}) + N_v\mathsf{count}(\mathtt{qH}) \tag{2}$$

We will treat qV, qH, and $(\mathtt{qV},\mathtt{qH})$ as bit-vectors or integers (i.e. a short hand for $\mathsf{count}(\mathtt{qV})$, etc.) when the interpretation is clear from context.

Using two counters reduces the total number of flip-flops for the counters from N to $N_v + N_h$. More significantly, it reduces the number of synchronizers from the put-control to the get-control from N to N_v, and likewise for the number of synchronizers from get to put. This allows the FIFO to have the performance of one-hot control with only modest area overhead.

Figure 3 sketches the latch and counter control circuit. The counters are post-incremented on put operations to avoid creating a long combinational path on req_put. In a bit more detail, req_put and data_in are latched on clk_put↑. The data is immediately written into the data storage latch; this minimizes the FIFO latency. The ring-counter pointers are then incremented on the *next* clk_put↑.

The ohV_l and ohH_l are one-hot recodings of the thermometer counters that are latched when put_clk is low. The one-hot representation is obtained by XORing adjacent bits of the thermometer counter (XNOR for the wrap-around case). Note that the latch control signals are updated on the falling edge of clk_put. In particular, data is presented to an enabled data latch on the rising edge of clk_put and the latch goes opaque a half-period later on the falling edge.

When put operations are performed on two consecutive cycles, the counter increment for the first put occurs *after* latching the counter values in ohV_l and ohH_l. We retimed these signals so that ohV_l and ohH_l take on the values that correspond to the state the ring counter will have after then next clk_put↑. The rest of the write control is simple: when do_put is asserted, the latch selected by ohV_l and ohH_l is written during the high phase of clk_put.

The empty or full status of each data latch is determined by converting the two counters for each into their combined thermometer values as illustrated in Fig. 4 and specified by the equations below:

$$\mathsf{therm}(i,j) = \begin{cases} \neg\mathtt{qH}[N_h-1], & \text{if } j=0,\ \mathtt{qV}[\mathrm{i}] = (j \bmod 2) \\ \mathtt{qH}[j-1], & \text{if } j>0,\ \mathtt{qV}[\mathrm{i}] = (j \bmod 2) \\ \mathtt{qH}[j], & \text{if } \mathtt{qV}[i] \neq (j \bmod 2) \end{cases} \tag{3}$$

We write $\mathsf{therm}(i+N_v j)$ as a synonym for $\mathsf{therm}(i,j)$ and note that values of $\mathsf{therm}(0)$ through $\mathsf{therm}(N-1)$ mimic the outputs of a $N = N_v N_h$ stage thermometer counter. Let $\mathsf{therm}_{\mathrm{put}}$ denote the thermometer values derived from the counters in the put-control, and $\mathsf{therm}_{\mathrm{get}}$ to those from the get-control. The empty and full functions indicate the status of the data latches:

$$\begin{aligned} \mathtt{empty}(i,j) &= \mathsf{therm}_{\mathrm{put}}(i,j) = \mathsf{therm}_{\mathrm{get}}(i,j) \\ \mathtt{full}(i,j) &= \neg\mathtt{empty}(i,j) \end{aligned} \tag{4}$$

This very simple logic for comparing the write and read pointers enables high throughput operation. Eq. 4 is the "bridge" from Fig. 4 to Fig. 5. Due to space limitations, we have not included a figure for this circuitry; it's just an array of XNOR gates (for empty) with inverters (for full), with one such pair for each (i,j).

Figure 5 shows the synchronization for the space available signal. The FIFO has space available if any data latch is

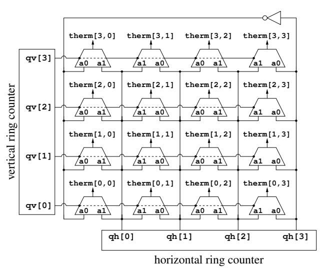

Fig. 4. Combining Thermometer Codes

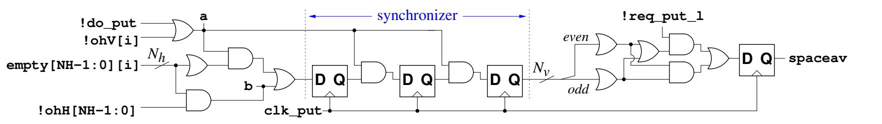

Fig. 5. Synchronization

empty; thus, the space-available logic is based on an OR-tree with N inputs. We again exploit an interleaved approach to simplify the design. The synchronizers are in the middle of the OR-tree with one synchronizer for each row of the design. Figure 5 shows an implementation with three flip-flop synchronizers. The logic to the left of the synchronizer flip-flops determines, for each row, if there is an empty latch in that row. The logic to the right of the synchronizer determines if there is any row with an empty latch. The next few paragraphs describe this approach in more detail.

Rising transitions of `empty` are synchronous to `clk_get`, and must be synchronized to `clk_put`. If any latch in row i is empty and that latch is not being written in the current cycle, then there is space available in the row. Detecting that there is an empty latch in row i is simple, this is the OR over `empty`$[N_h - 1:0]$ $[i]$. We need to check that such an empty latch is not being written in the current cycle. A safe approach would be assume that any write to the row might fill any latch in that row. As we will show in Section V, minimizing the number of cycles that must elapse between consecutive puts (or consecutive gets) to the same row has a large impact on throughput. This motivates giving more attention to how we can distinguish writes to different latches in the same row.

The simplest case occurs if there is no put in the current cycle (i.e. `do_put` is false) or if the put controller's row counter points to a different row (i.e. `ohV`$[i]$ is false), then no latch in this row will be written in this cycle. Thus, if `empty`$[j]$ $[i]$ holds for some j, then there is an empty slot available in row i. This is the condition checked by indicator "`a`" in Fig. 5. The other possibility is that there is a write to this row, but to a different column; this is checked by indicator "`b`". With this logic, the FIFO can perform a put to row i, and still determine that some other latch in the row is empty.

When a put is performed to row i, an empty slot is consumed for the row. Thus, we clear all but the first flip-flop of the synchronizer. If there is another empty slot in the row, then the D-input of the first flip-flop will remain high. Because rising edges of `empty` are asynchronous to `clk_put`, the indicator that there are other empty slots in the current row must be properly synchronized.

After synchronization, the per-row indicators for empty slots are combined with a second OR-tree. In this case, we divide the tree into two parts: one for even-indexed rows and the other for odd-indexed. If there is no put request in the current cycle, then the presence of an empty slot in either half is sufficient to assert `spaceav`. On the other hand, if there is a put request, then we can only assert `spaceav` if there are at least two empty slots: one for the current request, and one to offer for a subsequent put. Empty slots occupy consecutive locations in the FIFO. If there are two empty slots, there must be one in an even-indexed row and another in an odd indexed row. Thus, if both subtrees indicate that they have empty slots, then `spaceav` can be asserted regardless of the value of `req_put`.

We observe that the put-interface can attempt a write to row i at most once for every N_v cycles of `put_clk`. Let S denote the number of flip-flops in each synchronizer. A put to row i will clear the last $S-1$ flip-flops of the synchronizer on the cycle that the put is performed. It will take $S-1$ cycles of `put_clk` for this to propagate to the output of the synchronizer for row i. To sustain a maximal throughput of one put for every cycle of `put_clk`, the interleaving factor for the synchronizers must be at least as large as each synchronizer's depth; in other words, $N_v \geq S$.

A final note on synchronization is "the glitch that doesn't matter". Consider the case where $\mathtt{qV}_{\mathrm{get}} = N_v - 1$, $\mathtt{qH}_{\mathrm{get}} = j$, and a get is performed. On the next edge of `get_clk`, $\mathtt{qV}_{\mathrm{get}} \leftarrow 0$, and $\mathtt{qH}_{\mathrm{get}} \leftarrow j+1$. If $\mathtt{qV}_{\mathrm{get}}$ transitions before $\mathtt{qH}_{\mathrm{get}}$, then there may be a high-low-high glitch on `empty`$[0,j]$. For this to occur, it must be the case that

$$\mathsf{therm}(0,j) < \mathsf{therm}(\mathtt{qV}_{\mathrm{put}}, \mathtt{qH}_{\mathrm{put}}) < \mathsf{therm}(0,j+1) \quad (5)$$

Therefore, `empty`$[0,j+1]$ is making a low-to-high transition, and there is an empty slot available in this row. Thus, the glitch on `empty`$[0,j]$ is benign.

III.2. The Data Path

Figure 6 shows the data path through the FIFO. Data is acquired with a low-enabled latch at the input of the FIFO. As described above, the put-controller asserts the write enables of the data latches when `clk_put` is high. Together, the input latch and the selected data latch implement the behaviour of a positive, edge-triggered flip-flop. Thus, the timing is familiar to designers. The data is stored in the latch immediately following the rising edge of `clk_put`.

The output path is a bit more involved. We observed that the output select logic, whether tri-state drivers or multiplexors, was a critical cycle time limiter. Once again, interleaving provided a solution. The FIFO has two output buses driven by

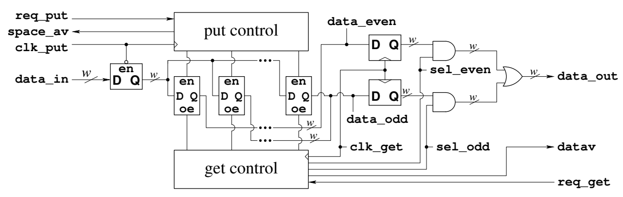

Fig. 6. The Data Path

the data latches: data_even is driven by latches with even row indices, and data_odd is driven by the odd-indexed latches. A key observation is that each data value must dwell in its data latch for at least the synchronization delay. Thus, when the get-controller is ready to read a value from a latch, it has already been there for at least one cycle of clk_get. The get-controller exploits this observation by enabling the output driver for a data latch when its predecessor is being accessed. Because the data latches are accessed at alternating even and odd indices, the current latch and the "next up" latch drive different buses. The get-controller generates sel_even and sel_odd signals to control which bus drives data_out. If both sel_even and sel_odd are false, then data_out will be driven with all zeros.

The signal datav indicates that the value on data_out is valid data. It will remain asserted until a req_get requests a new word. If that word is immediately available, it will be output following clk_get↑, and datav will remain high. Otherwise, datav will drop until valid data is output, and in the interim, both sel_even and sel_odd will be false, and data_out will be driven with driven with all zeros. We use this to suppress the glitch hazard described below.

III.3. Blocking the Glitch Path

While developing the FIFO, we realized that the hazards described in [5] can occur in synchronization FIFOs. Logic synthesis tools can introduce glitches when a logic block includes inputs from other clock domains. Consider the Verilog fragment:

```
always @posedge(get_clk) begin
  if(data_valid) qq <= f(data_out, local1)
  else qq <= g(local2)
end
```

where f and g are combinational logic functions, and local1 and local2 are state variables local to the get_clk clock domain. The designer may expect the if-then-else to be implemented with a multiplexor circuit, but the logic synthesis tools can generate a network of gates that intermixes the operations of f, g, and the multiplexor. The synthesized network is logically equivalent to the RTL source *if* all of the inputs to the network from the FIFO, are stable between clock events. If this stability assumption is violated, then glitches can occur and cause the circuit to malfunction. In particular, if the FIFO's output, data_out, has unstable signals on cycles when data_valid is false, a glitch failure can occur.

Such glitch hazards are prevalent in published designs of clock-domain crossing FIFOs, and we are aware of no prior work that has reported this hazard. Most of the bi-synchronous FIFOs cited in Section II produce outputs that can be unstable on cycles where data validity is not guaranteed. For example, data can flow from the sender, through a dual-ported SRAM, to the data_out port of the FIFO and change asynchronously with respect to clk_get [1], [3], or data_out may be driven by a flip-flop or latch that is updated on clk_put [2], [7]. The ripple FIFO designs in [10], [11] avoid this problem by ensuring that data is properly synchronized by the time it reaches the final, output FIFO stage. This comes at a cost of extra power consumption because of the repeated copying of the data value from stage to stage. The design in [4] does not allow the get pointer to be advanced until the corresponding data register is known to hold valid data. This avoids the glitch hazard because the receiver can only issue a get request when the FIFO has asserted that it has data available. This data is output following on the following clock cycle. Mesochronous designs such a such as [13]–[16] avoid these glitches because the metastability is resolved during initialization and does not occur under subsequent operation.

Our solution forces data_out to be all zeros when data_valid is false. In a bit more detail, the get-controller has an interleaved synchronizer to identify full data latches; this synchronizer is equivalent to the one in the put-controller shown in Fig. 5. As in the put-controller, the get-controller divides the final OR-tree into two subtrees, one for even indexed rows and the other for odd. These provide separate data-validity signals for the even and odd paths: dv_even and dv_odd. A toggle enabled by req_get identifies whether the next data word should come from the even or odd latches. Combining the output of this toggle with dv_even and dv_odd provides sel_even and sel_odd respectively. If neither path has valid data, the FIFO outputs a word of all zeros for that clock cycle. This ensures that the FIFO produces no ill-timed signals that could trigger synthesis-induced glitches in the receiver's logic.

The cost of this design is an extra pipeline stage in the output of the FIFO, and our FIFO incurs an extra clock cycle of latency like the one in [4]. As described below, the fan-out tree for these enables is a critical timing path when designing for high-frequency clocks. Thus, we believe that this extra latency is necessary. An alternative would be to expose the glitch and use a tool like the one described in [5] to identify any resulting glitches and then take corrective measures. We are not aware of any open source implementation of that tool. Furthermore, we expect that most designers prefer a design that always produces a correct FIFO. That is the Verilog code that we provide.

III.4. Timing Details

An "ideal" synchronizing FIFO would look to the sender and receiver as if it were a flip-flop with two clock-inputs: clk_put and clk_get. Because these clocks may run at different frequencies, the FIFO must have flow control signals as well: `spaceav` an output of the FIFO that asserts there is space available for a put operation; `req_put` an input to request a "put" operation – i.e. insert the value of `data_in` into the FIFO; `datav` an output of the FIFO that asserts that the data currently being output is valid; `req_get` an input to request a "get" operation – i.e. update `data_out` with the next value when it is available. For an ideal FIFO, we would like the signals `data_in`, `req_put`, and `spaceav` to be synchronous to `clk_put`. It should be sufficient for `data_in` and `req_put` to settle within the set-up and hold times for a standard flip-flop in the implementation technology. Likewise, `spaceav` should settle within the clock-to-Q delay for such flip flops. Furthermore, we would like `data_out`, `req_get`, and `datav` to be synchronous to `clk_get`.

No real FIFO can meet these constraints. The value of `spaceav` on one cycle of `clk_put` depends on the value of `req_put` on the previous cycle. Thus, there must be a few logic gates in this path. Our design requires `req_put` to settle two gate delays before the flip-flop set-up time. However, our design also allows the sender to "aggressively" assert `req_put`. The put will only be performed on cycles where `spaceav` was true. Thus, `spaceav` can be treated as an acknowledgement that the requested put will take place on the next rising edge of `clk_put`. In the same way, the values of `data_out` and `datav` on one cycle of `clk_put` depend on the value of `req_get` on the previous cycle. In this case, the critical cycle includes the fan-out of the control signals to all of the data-bits of the output word. In our design, `req_get` must settle two gate delays before the flip-flop set-up time. The fan-out delays are incurred after the rising edge of `clk_get`, and `data_out` does not settle until the control signals and selected data value propagate through the output multiplexor. The receiver can "aggressively" assert `req_get` and treat `datav` as an acknowledgement.

IV. An Open Source FIFO

In order to verify and simulate the suggested approach, we wrote a fully parameterized Verilog module. To simulate and synthesize the proposed designs with various parameters in batch using Synopsys Synthesis tools, we also provide a run-in-batch flow manager. The Verilog modules and the flow manager are available online [6]. The design package is composed of a Verilog description of the proposed FIFO, a Verilog testbench, tool configuration scripts, and a run-in-batch manager. The Verilog description of the proposed FIFO is parameterized and can be instantiated in other Verilog projects as a stand-alone IP.

The Verilog testbench emulates five use cases as follows. (1) **Fast:** the sender will put data in the FIFO whenever space is available, and the receiver will get data whenever data valid is asserted. (2) **Random:** when space is available, the sender chooses randomly to put data or to remain idle and likewise for the receiver. (3) **Empty:** the sender and receiver keep the FIFO close to empty. (4) **Half:** the FIFO is kept close to half full. (4) **Full:** the FIFO is kept close to full.

The run-in-batch manager allows synthesizing and simulating a number of FIFOs in batch. A list of design parameters (e.g. vertical stages, horizontal stages, data width, and synchronizers' depth) can be provided to the run-in-batch manager, together with the required flow stages. The design stages are implemented using Synopsys tools and ordered as follows. (1) **Logic synthesis** using *Design Compiler*. (2) **Placement and routing** using *IC Compiler*; this include delay and parasitic extraction. (3) **Static timing analysis** using *PrimeTime* (4) **Gate-level-Simulation** using *VCS* with the placed and routed netlist, delays and parasitics data, and the Verilog testbench. The testbench will generates input vectors checks the outputs, comparing them against a generic FIFO. The simulation also generates the activity data for power analysis. (5) **Power analysis** using *PrimeTime* to estimate both dynamic and leakage power of the FIFO.

TABLE I
"Efficiency" vs. N_v and S

	$S=2$	$S=3$	$S=4$
$N_v=2$	[0.48 0.50 0.50]	[0.38 0.47 0.50]	[0.33 0.38 0.40]
$N_v=4$	[0.87 0.97 1.00]	[0.73 0.93 1.00]	[0.62 0.72 0.75]
$N_v=6$	[1.00 1.00 1.00]	[1.00 1.00 1.00]	[0.95 0.98 1.00]
$N_v=8$	[1.00 1.00 1.00]	[1.00 1.00 1.00]	[0.99 0.99 1.00]

Each entry is [min mean max].

V. Results

To validate the design, we used the batch-in-run scripts to test all combinations of the following parameters: number of vertical stages, $N_v \in \{2,4,6,8\}$; number of horizontal stages, $N_h \in \{2,4,6,8\}$; data word width, $w \in \{8,16,32\}$; and synchronizer depth, $S \in \{2,3,4\}$ flip-flops. We synthesized each FIFO with different speed targets, and used binary search to find the top operating frequency for each design. We used the cell library for a 65nm process. Each FIFO was tested with thousands of cycles for the various usage scenarios described above. Performance numbers are based on extracted layout.

The results with $N_v = 2$ are primarily for checking that the design and scripts work in the corner cases. These FIFOs achieve clock rates of 1.7GHz for $N_v = N_h = 2$ and a data word width of 8 or 16 bits. These high clock frequencies are useless for realistic applications because the FIFOs are nearly always stalled due to inadequate synchronizer interleaving. The same FIFOs achieve throughputs of 520-850M transfers/second. For $4 \leq N_v \leq 8$, the designs achieve clock frequencies from 1.0 to 1.6 GHz. The highest clock frequencies are again for imbalanced designs, but there are many choices of parameters that achieve clock rates of 1.2 or 1.3 GHz and sustain one transfer per clock cycle. We will focus on these "practical" designs, along with some trends that illustrate the key trade-offs.

We first look at the impact of interleaving on FIFO throughput. Let "Efficiency" denote the throughput of the FIFO (transfers per second) divided by the clock frequency – if the FIFO could sustain one transfer per cycle on both interfaces, then the Efficiency would be 1.00. For $N_v \leq S$ we see that the FIFO is often stalled waiting for the synchronizers. When $N_v > S$, efficiency approaches 100%. There are two reasons for the lower numbers: the main cause is that for small values of N_h, the FIFO does not have enough slots to cover the latency of the put and get control and the synchronizers. In particular, when $N_v > S$ and $N_h \geq 4$ the efficiencies are always above 99%. The remaining fraction of a percent appears to be a simulation artifact.

The synthesizable clock frequency decreases for larger values of N_v and N_h, and has little sensitivity to the data word width, w, and no significant dependency on the synchronizer depth, S. The data fits a model of

$$T = [74\log_2(N_v) + 112\log_2(N_h) + 36\log_2(w) + 293]\,\text{ps} \quad (6)$$

where T is the clock period. The RMS error is 52ps. As we only tried clock frequencies that are multiples of 100MHz, this is roughly the same as the frequency resolution from the dataset. Using the logarithms of the parameters reduced the fitting error by about 10% compared with using a linear model. This suggests that the critical structures are trees, for example, the OR-trees in the empty and full synchronizers. For the choices of parameters we used in our sweep, the operating frequencies are in the range of 1GHz to 1.5GHz. For example, a 16 stage FIFO ($N_v = N_h = 4$) with 8-bit data words and two or three flip-flop synchronizers achieves full throughput at an operating frequency of 1.3GHz. The same configuration but with 16 or 32-bit data words achieves full throughput at 1.1GHz. These frequencies and throughputs are well-above typical target frequencies for an ASIC in a 65nm process.

By itself, clock frequency is not a very meaningful measure of performance. Thus, we also consider throughput and latency. To achieve full throughput, the N_v must be large enough to maintain synchronizer interleaving without any stall cycles. Furthermore, the total capacity, $N = N_v N_h$ must be large enough to accommodate a round-trip from the sender to the receiver and back – intuitively, the sender consumes an empty slot when it performs a put, and the FIFO must have sufficient capacity to operate until that slot can be used by the sender again. Both of these conditions depend on the synchronizer latency. From the simulation data, full throughput i.e. one put and get per clock cycle – requires $N_v \geq 4$. Furthermore, when the synchronizer depth, S, is 2 or 3, then full throughput is achieved if $N_v N_h \geq 12$. When the synchronizer depth is increased to 4, then full throughput is achieved when $N_v N_h \geq 16$.

The latency of the FIFOs is given by

$$L = (S+1)T_{\text{get}} + 368\text{ps} \quad (7)$$

where S is the number of synchronizer stages and T_{get} is the period of `clk_get`. The RMS error is 55ps (1.2%). The latency is composed of the synchronizer delay, ST_{get} plus one more cycle for the glitch blocking FFs plus some combinational logic delay. Any further reductions in the latency would require reducing the combinational logic delay which Eq. 7 shows is already fairly small.

Power consumption is dominated by the vertical and horizontal counters and associated control logic. Normalizing for frequency, a linear regression yields a model for the energy per clock cycle of:

$$E = [95N_v + 83N_h + 42w + 20N_vS - 395]\,\text{fJ} \quad (8)$$

with a relative error of $\pm 7.5\%$. Using the linear terms, N_v and N_h produced a better fit than their logarithms – these terms are effectively a measure of the flip-flops and logic gates in the control. In a similar fashion, the N_vS term accounts for the synchronizer flip-flops. This suggests that power consumption is dominated by the control flip-flops and the associated logic. Including the number of FIFO stages, $N = N_v N_h$, in the regression only produced a slight reduction in the error term, but it did it by making fairly large, "cancelling" changes to the other coefficients. This shows that for FIFOs of practical sizes, the power consumption for our architecture is largely independent of the number of stages. Including N in the power model risks overfitting the data; so, we didn't. To get power (in mW), multiply the energy per operation from Eq. 8 (in fJ) by the operating frequency (in GHz). For the choices of parameters we used in our sweep, power consumption ranges from 0.98mW to 3.4mW.

Area is dominated by the data-storage latches and associated circuitry. A linear regression yields:

$$A = [10N_vN_hw + 119N_v + 30N_vS + 112N_h + 208]\,\mu^2 \quad (9)$$

with a relative error of $\pm 2.9\%$. For example, consider a small FIFO with $N_v = 4$, $N_h = 2$, $w = 8$, and $S = 2$. Equation 9 predicts an area of $1791\mu^2$ and the actual, synthesized FIFO has an area of $1745\mu^2$. Our model predicts that $\frac{640}{1791} = 36\%$ of the area is for the data-storage array. The data from the synthesis tools shows that 18.5% of the area goes to the storage latches themselves. Other area goes to the tri-state buffers and perhaps other, per-data-latch logic – we have not as yet, fully reverse-engineered the synthesized design. For a FIFO with $N_v = 8$, $N_h = 8$, $w = 32$, and $S = 2$, our model predicts an area of $23,710\mu^2$ ($23,702\mu^2$ for the actual, synthesized FIFO) of which 87% is for data storage. For moderate sized FIFOs and larger, the area is dominated by the data store and not the control logic.

We would like to compare our design with other published FIFO, such as those surveyed in Section II. Alas, many of these design make use of full-custom logic elements and cannot be synthesized using standard cell library. Only one of the designs, [3] provides open-source Verilog. Unfortunately, their design requires a SRAM generator, and we did not have a license for such a tool at the time of writing this paper. We believe that this underscores the value of providing an open-source design, and hope that by making our design available, other designers will benefit from using it, and other researchers can compare their designs with ours.

VI. Conclusions

We have presented a novel, high-performance synchronization FIFO for crossing clock domains. The key feature of the design is its "interleaved" approach. The current write and read locations are each maintained by a pair of thermometer (unary) counters. This enables our FIFO to achieve fast read/write location comparisons of a unary design using significantly less control logic. Most parts of the control scale roughly as the square-root of the FIFO capacity rather than linearly. The per-row synchronization for communicating full and empty status between the put- and get-interfaces is a novel approach that offers excellent performance and efficiency. The interleaving style simplifies many other challenges in achieving a high performance, synthesizable design. We showed how we use interleaving in the computation of control signals for maximizing the throughput of the FIFOs data path.

We described how the synthesis-induced glitches reported in [5] can arise from the use of synchronization FIFOs. Many published designs have flow-through paths that allow the data output from the FIFO to change asynchronously with respect to the receiver's clock domain. While the data valid indicator is false during such asynchronous data transitions, standard synthesis tools can combine signals and logic from a "non-selected" path with those from the intended path and create glitches. We have not seen this vulnerability previously reported. Our FIFO is safe with respect to such hazards.

We constructed analytical models for performance, power consumption, and area based on the synthesis and simulation results. These models match the actual, synthesized FIFOs to with a few percent. Such models allow architects or module designers to evaluate trade-offs quickly without needing to actually synthesize and simulate each alternative. By their simplicity, these models provide insight into where the bottlenecks and trade-offs are in the design.

Our FIFO is available as an open-source, highly parameterized design [6]. We provide scripts for an industry standard design flow based on Synopsis tools. This includes synthesis, layout generation, extraction, timing analysis, simulation, and power estimation. We include a simulation test bench, and scripts for batching sweeping design parameters to enable exploration of the design space. Our design should be both useful to designers and provide a repeatable reference case for other researchers.

Acknowledgements

We appreciate helpful discussion with Brad Quinton and Tarik Ono in the early phases of this work. Many thanks to the anonymous reviewers for their feedback and suggestions. This research would not have been possible without the access to design tools and libraries that we obtained through the Canadian Microelectronics Corporation. This work was supported by the NSERC Discovery Grant programme.

References

[1] R. Apperson, Z. Yu, J. Meeuwsen, T. Mohsenin, and B. Baas, "A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 19, pp. 1125–1134, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2007.903938

[2] A. Strano, D. Ludovici, and D. Bertozzi, "A library of dual-clock FIFOs for cost-effective and flexible MPSoC design," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2010, pp. 20–27. [Online]. Available: http://dx.doi.org/10.1109/ICSAMOS.2010.5642098

[3] C. Cummings and P. Alfke, "Simulation and synthesis techniques for asynchronous FIFO design with asynchronous pointer comparison," in *SNUG-2002, San Jose, CA*, 2002. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

[4] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Transactions on VLSI Systems*, vol. 12, no. 8, pp. 857–873, Aug. 2004. [Online]. Available: dx.doi.org/10.1109/TVLSI.2004.831476

[5] Y. Peng, I. W. Jones, and M. R. Greenstreet, "Finding glitches using formal methods," in *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE Computer Society, May 2016, pp. 45–46. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2016.12

[6] A. Abdelhadi, "Cell-based mixed fifo," Open source repository, 2017. [Online]. Available: https://github.com/AmeerAbdelhadi/Interleaved-Synthesizable-Synchronization-FIFOs

[7] I. M. Panades and A. Greiner, "Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in GALS architectures," in *First International Symposium on Networks-on-Chip (NOCS'07)*, May 2007, pp. 83–94. [Online]. Available: http://www.dx.doi.org/10.1109/NOCS.2007.14

[8] T. Ono and M. R. Greenstreet, "A modular synchronizing FIFO for NoCs," in 3rd *Network on Chip Symposium*, ser. NOCS'09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 224–233. [Online]. Available: http://dx.doi.org/10.1109/NOCS.2009.5071471

[9] B. Keller, M. Fojtik, and B. Khailany, "A pausible bisynchronous FIFO for GALS systems," in *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, May 2015, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2015.9

[10] J. N. Seizovic, "Pipeline synchronization," in *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, 1994, pp. 87–96. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.1994.656289

[11] S. Jackson and R. Manohar, "Gradual synchronization," in *22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2016, pp. 29–36. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2016.21

[12] D. G. Messerschmitt, "Synchronization in digital system design," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 8, pp. 1404–1419, Oct. 1990. [Online]. Available: http://dx.doi.org/10.1109/49.62819

[13] M. R. Greenstreet, "STARI: A technique for high-bandwidth communication," Ph.D. dissertation, Department of Computer Science, Princeton University, Jan. 1993. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.9349&rep=rep1&type=pdf

[14] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, May 2003, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2003.1199168

[15] W. Dally and S. Tell, "The even/odd synchronizer: A fast, all-digital, periodic synchronizer," in *Proceedings of the* 16th *International Symposium on Asynchronous Circuits and Systems*, 2010, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ASYNC.2010.20

[16] D. Verbitsky, R. Dobkin, R. Ginosar, and S. Beer, "StarSync: An extendable standard-cell mesochronous synchronizer," *Integration, the VLSI Journal*, vol. 47, no. 2, pp. 250–260, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926013000497