Multi Buses: Theory and Practical Considerations of Data Bus Width Scaling in FPGAs

Lukáš Kekely, Jakub Cabal
CESNET, a. l. e.
Zikova 4, 160 00 Prague 6
Czech Republic
{kekely, cabal}@cesnet.cz

Viktor Puš
Netcope Technologies
Sochorova 3232/34, 616 00 Brno
Czech Republic
pus@netcope.com

Jan Kořenek
IT4Innovations Centre of Excellence
FIT, Brno University of Technology
Božetěchova 2, 612 00 Brno, Czech Republic
korenek@fit.vutbr.cz

***Abstract*—As the throughput of computer networks and other peripheral interfaces is rising, developers are forced to use ever-wider data buses in FPGA designs. However, utilization of wide buses poses a serious threat of performance degradation, especially for the shortest data transactions (packets), as aliasing and alignment overheads on the bus can be extremely increased.**

In this paper, we propose a novel design method for the description of very wide data buses that we call Multi Buses. The key idea is to enable the processing of multiple transactions per clock cycle with very high and predictable effective throughput even in the worst-case. The feasibility of the proposed method is shown via analysis of achievable performance by both theoretical means and selected proof of concept implementations. Thanks to the proposed method, we were able to design FPGA cores for key operations in networking (e.g. parser, match table, CRC, deparser) with sufficient throughputs for wire-speed packet processing of 400 Gbps, 1 Tbps and even 2 Tbps Ethernet links.

I. Introduction

The working frequency of current FPGAs can reach several hundreds of MHz, however, it is unusual to see any complex FPGA logic running at more than 500 MHz. The need to increase the achievable frequency is documented by works of major FPGA manufacturers in their latest chip families. Xilinx came up with the Time Borrowing concept for UltraScale+ FPGAs [1]. It promises average maximal frequency (F_{max}) increase of 5.5 % to 8.5 %, depending on the scale of design changes. Intel introduced the HyperRegisters [2] inside its Stratix 10 FPGAs. These registers are spread across the FPGA routing fabric and they are used to balance critical paths during routing (when retiming is performed). The promised F_{max} improvement is up to $2\times$ in some cases. Even with these advances, practical FPGA firmware running at frequencies over 1 GHz is not expected in the foreseeable future.

On the other hand, there is an ever-increasing demand for communication throughput. It is driven mainly by traffic generated by high-quality content delivery services and aggregated at network backbones and in data centers. The demand is also apparent in many related fields, like in high-performance memories driven by wider adoption of stacked DRAM solutions [3], [4] or in propositions of ever-faster standards for serial system buses like PCI Express [5].

The flexibility and parallelism of FPGAs can solve many challenges that high-speed communications face. For example, the UltraScale+ VU13P FPGA has 128 GTY transceivers with a raw aggregated full-duplex throughput of over 4 Tbps [6]. The pipelined stream processing of data frames, which is typical for high-performance FPGA designs, has the benefit of huge and predictable throughput with very little circuitry overhead compared to instruction processing engines (CPUs). At the same time, the structural programmability of FPGAs hides the inflexibility of mostly fixed-function pipelined processing. But employment of the massive parallelism needed for speeds above a certain threshold can easily hit limits of convenient design, causing a need for some extra care, and perhaps a level of data bus design formalism. This is especially apparent when the minimal length of protocol transaction becomes equal to, or even smaller than, the data bus width.

Let's take 100 Gbps Ethernet as an example here. The shortest allowed length of L2 Ethernet frame is 64 B (512 b). At the same time, a typical implementation of FPGA bus transferring 100 Gbps of data is 512 b wide and running at a clock frequency of at least 195 MHz. The shortest 64 B Ethernet frame fits nicely into a single data bus word. But what about 65 B, 66 B and similar frames? When a few frame bytes spill into the second word, the rest of that word may remain unused, yielding effective throughput of only around 50 % of the bus raw capacity. Alternatively, that second word may be somehow shared with the beginning of the next frame. This improves achievable effective throughput significantly, but also brings new complications into the design of processing logic – suddenly fragments of two separate frames are present in the same data word and need to be dealt with in the same clock cycle. Furthermore, the frames must be allowed to start at arbitrary offsets within the data word, not only at its first byte.

The problem of growing complexity gets much more severe when even faster protocols are considered. Without a significant increase in working frequency, buses become wider than minimal transaction lengths. Therefore, processing of multiple transactions or their fractions per clock cycle must be possible to make effective use of raw bus capacity. Due to the requirement of transferring multiple transactions per cycle, we named such buses Multi Buses.

When creating cores for Multi Buses, the design cost of the actual processing logic can be easily overshadowed by the cost of just the parallelization for multiple transactions. Therefore, questions like the following should be answered, considering optimal resources utilization and achievable frequency:

- How many transactions per clock cycle to process?

- What alignment rules (if any) should the bus follow?
- How to design reasonably simple processing replication?
- How to effectively route data to processing modules?

This paper aims to provide a consistent background theory and guidelines for FPGA design decisions like these so that FPGA processing pipelines and cores for high bandwidth interfaces can be easily designed according to proposed Multi Buses method. The contribution of our work is four-fold:

- Formalization of data buses capable of processing multiple frames per clock cycle and related metadata buses carrying multiple values per cycle (Chapter II)
- Worst-case throughput analysis of Multi Buses to show their superiority over single-transaction ones (Chapter III)
- Guidelines for how to design and implement efficient processing cores for Multi Buses (Chapter IV)
- Evaluation of the proposed concept on simple but reusable processing cores and an overview of achieved results for selected complex engines (Chapter V)

Note that we do not claim wide buses design itself to be our contribution, quite the opposite. This work stems from our experience with various existing buses and is motivated by the need for common ground (language) in this area, backed by solid definitions, equations, and experimental results.

Since this paper is motivated mostly by the lack of relevant theory and guidelines for effective wide buses design, we are not able to provide the typical Related Work section. Instead, we use the AMBA AXI4-Stream [7] protocol to highlight the pieces that are often missing. The AXI4-Stream bus is of arbitrary width in Bytes, which are grouped into Packets (frames). There is a single-bit TLAST signal determining that the current bus word carries the end of a packet. The next packet can only begin in the next data word at the earliest. This makes perfect sense for buses up to a certain width, but after the further widening of the bus (for throughput), we encounter severe problems. There is no native support for carrying multiple packets within a single bus word, which limits throughput especially for packets shorter than bus width. There are also no rules to limit packet start position, which in turn can be completely arbitrary thanks to Null and Position bytes. Therefore, data processing cores must be able to start the processing from any byte, resulting in overly complex logic.

Overall, if we were to use AXI4-Stream in a very wide configuration, we would most likely end up defining a custom specification on top of it, using optional TUSER signals to convey additional information, such as multiple positions of packet starts/ends. We would probably also have to define custom packet alignment rules to reduce processing logic complexity, and therefore, we wouldn't be able to make use of standard AXI4-Stream infrastructure modules. We would be left with intuitive work and ad-hoc design of a new bus, with adherence to the original AXI4-Stream specification being more of an obstacle rather than an advantage. It becomes clear that very wide buses require dedicated design effort. Our paper aims at simplification and formalization of this effort.

II. Multi Buses Concept

The key feature of our Multi Buses is their capability to transfer multiple data transactions in each clock cycle. By *transaction* we mean a data frame formed as a variably long sequence of some homomorphic data *elements* (smallest addressable/distinguishable pieces of transferred information). For example, a network frame is a variably long sequence of bytes, a memory burst is a variably long sequence of read/written rows. Furthermore, we consider frames to be transferred continuously over the bus – no holes inside the frames are allowed so each frame is explicitly delimited just by the position of its first and last element (start/end position).

The main objective when designing an effective Multi Bus is to find a structure of its data word, which appropriately constraints frames positioning. The constraints must be restrictive enough to maximally decrease the number of allowed position combinations (to keep the processing logic as simple as possible), while maintaining bus transaction aliasing and alignment overhead acceptably low. The most general version is, obviously, allowing frames to start and end at totally arbitrary positions. It is clear that such word structure does not create any alignment overhead. But on the other hand, it leads to extremely complex architectures of cores in bus processing pipelines, simply because they must expect that each word element can potentially be a start of yet another new frame.

A potential way how the mentioned complexity can be significantly reduced without considerable overhead is to take advantage of the fact that frame lengths are usually limited. The idea is to split elements of bus data word into multiple *regions* and then allow only at most one frame to start and at most one (not necessarily the same) frame to end in each region. Now the number of potential frame start positions in each bus word is reduced significantly – from the number of elements to the number of regions. Also, when region size is picked properly, considering the shortest transferred data frames, none or minimal alignment overhead is added.

Another possibility to reduce processing complexity is to constrain frame start positions inside regions. Each region can be split into multiple *blocks* of data elements where frames are allowed to start only aligned to these blocks. This way, the number of potential frame start positions in each region is reduced, thus the processing logic is simplified. On the other hand, some alignment overhead is added for each frame. Tuning the size of blocks enables a trade-off between throughput efficiency and chip resource usage.

To summarize, each bus data word consists of regions restraining the maximal number of frames per cycle, and each region is separated into blocks of data elements to further constraint positioning of frame starts. We call this arrangement of the data word and each bus following it *Multi Frame Bus* (MFB). Illustration of MFB word structure is shown in Figure 1 (upper part). A word with 16 data elements organized into 4 regions (4 elements per region) and 8 blocks (2 elements per block, 2 blocks per region) can be seen. The lower part of the figure shows an example of possible positioning of six frames

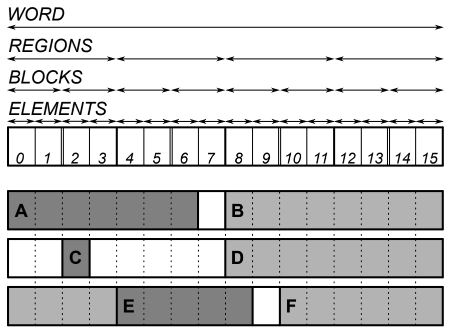

Fig. 1. Structure of MFB word with illustration of possible frame placements.

(A to F) transferred over three words of such MFB. Notice that each frame must start aligned to start of a block, but can end in any element (e.g. C and E ending in the middle of blocks). In the first word, there is an unused data element 7 between A and B which cannot be utilized (block alignment rule), showing a slight throughput inefficiency of MFB. On the other hand, frames in the second word can be packed tighter – D can start four elements sooner (at element 4). C can be similarly moved by two elements (at element 0), but with no effect to overall throughput, since the rule of at most one start of a frame in each region prevents D to be moved further left. This illustrates that MFB has rather high overhead when frames are too short compared to region size (not optimal MFB configuration as we explain later). Lastly, in the third word a legal sharing of one region between parts of two frames is shown – E ends in the same region in which F starts.

General description of MFB data word structure enables definition of multiple buses with different parameters. We formally describe them by the following four attributes:

- *Number of regions (n)* adjusts the maximal number of frames transferred in each word.
- *Region size (r)* defines the number of blocks in each region, thus affects overhead for very short frames.
- *Block size (b)* states the number of elements in each start alignment block, thus controls the alignment overhead.
- *Element width (e)* defines the size of the smallest distinguishable piece of data in bits. Usually, buses work with bytes (octets of bits), but other values can be also utilized.

To simplify the processing logic complexity, r and b should be powers of 2. Using the main attributes of MFB, we derive other useful parameters: *data word width* describing the actual width of the whole bus data word in elements as $w_e = n \times r \times b$ or in bits as $w_b = n \times r \times b \times e$, and *region elements* showing the size of each region in number of elements as $r_e = r \times b$. Furthermore, we define a short notation of specific word structure as $MFB(n, r, b, e)$. For example, the bus in Figure 1 is $MFB(4, 2, 2, *)$ ($*$ denotes any value).

That concludes the concept description of Multi Buses specifically designed for general data frames. Analogous to these, buses that can transfer multiple single-value transactions per clock cycle can be also defined. Such buses come in handy especially as complements to MFB for relaying

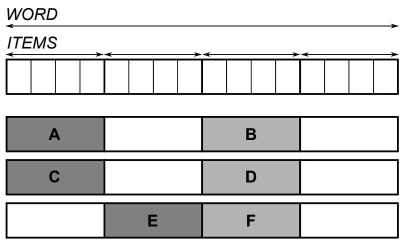

Fig. 2. Structure of MVB word with illustration of transferred values.

metadata associated with the frames. Because only single-value transactions are considered here, the word structure is pretty straightforward – each bus data word consists of multiple homogeneous data items (values) without any further constraints. We call this arrangement of the data word and each bus following it *Multi Value Bus* (MVB). Illustration of MVB word structure is shown in Figure 2 (upper part). A word of 16 bits organized into 4 items can be seen. The lower part of the figure shows an example of six values (A to F) transferred over three words of such MVB. Notice that their positions are chosen to correspond with starting regions of frames from Figure 1. As no constraints for value placements are present, MVB does not create any throughput overhead.

Analogous to MFB, we describe versions of MVB by:

- *Number of items (m)* adjusts the maximal number of values transferred in each word.
- *Item width (i)* represents the size of the values in bits.

Using these attributes, data word width is $w_i = i \times m$ and a short notation $MVB(m, i)$. For example, bus shown in Figure 2 can be now labeled as $MVB(4, *)$.

Description of MVB concludes the Multi Buses concept proposition. To wrap things up, we just want to point out that $MFB(1, 1, *, *)$ and $MVB(1, *)$ configurations behave like ordinary single-transaction buses, so single-transaction buses (e.g. AXI4-Stream) can be viewed as a special case in our proposed concept. Or in other words, the Multi Buses concept can be seen as an extension of the single-transaction buses.

III. Achievable Throughputs

The Multi Buses design introduces some constraints on frame positioning, leading to possible transfer overheads that reduce achievable throughput. Without deeper understanding of characteristics of mentioned overhead, a simple way for rule of thumb MFB worst-case throughput (T) estimate is:

$$T \approx F \times n \times \min(L_{min}, r_e) \times e \quad [b/s] \tag{1}$$

Where n, r_e and e are as defined in previous section, L_{min} is the length of the shortest frame (in data elements) and F is the bus clock frequency. To break down the equation, e is there just for the transformation of units – from data elements to actual bits. Term $F \times n$ defines the maximum theoretical number of transactions per second that MFB can transfer because in each of its n regions per word (clock cycle) at most one frame can start. Finally, the $\min()$ clause represents the fact that if the frames can be very short, the throughput for the shortest ones is simply limited by achievable framerate. On the other hand, if regions are very small compared to frames, the throughput is limited by the actual maximal amount of data that can be transferred over the bus (known $T_{max} = F \times w_b = F \times n \times r_e \times e$). This leads to an assumption that it is generally recommended to set MFB region size to be the same as the length of the shortest legal frames.

The approximation 1 accurately accounts for the throughput limitations created by the concept of regions in MFB. To further generalize the equation, the other source of overhead must be analyzed – concept of blocks. When a frame can start only at the start of a block, then as soon as the first element of that block is already used, a new frame cannot start there. In other words, when at least one element of a frame spills into a block, the whole block is used up, forcing the next frame to start in the next block at the earliest. This feature leads to block related MFB overhead of at most $b - 1$ elements per frame varying based on the actual frame length.

When both sources of MFB overhead are put together, the total number of elements required (allocated) on MFB (cost $C(L)$) when transferring a frame of length L is:

$$C(L) = \begin{cases} r_e & \text{if } L < r_e \\ \lceil L/b \rceil \times b & \text{otherwise} \end{cases} \tag{2}$$

The first case is derived from region constraint overhead for very short frames. Note that thanks to enabled sharing of a region between two frames, the actual cost can be smaller than r_e for some patterns of consecutive frames and their specific positions. However, this does not have to be considered here, as the worst-case scenario is examined. The second case of the function represents the allocation of the whole blocks by any last elements of the frame in them. Now, achievable efficiency ($\eta(L)$) of MFB when transferring frames of length L is relation between transferred amount of data and cost of the transfer:

$$\eta(L) = \frac{L}{C(L)} \tag{3}$$

Worst-case efficiency (H) is just a minimal value of $\eta(L)$:

$$H = \min_{L=L_{min}}^{L_{max}} (\eta(L)) \tag{4}$$

And finally, worst-case throughput of MFB (T) is simply:

$$T = H \times T_{max} = H \times F \times n \times r_e \times e \tag{5}$$

A demonstration of the achievable efficiency of different MFB groups can be seen in Figures 3 and 4. The first graph shows only the effect of block sizes (b) without region sizes taken into consideration ($r = 1$, so $r_e = b$). Constantly repeating drops in effectivity are getting more severe for shorter frames. Also, drops are deeper as the value of b raises. The second graph illustrates how the utilization of larger regions degrades achieved efficiency for the shortest frames. Here, b is set to 8 similarly as for $MFB(*, 1, 8, *)$ (squared) line in the previous graph. A steady linear drop of efficiency towards zero can be seen for all frames shorter than r_e. For

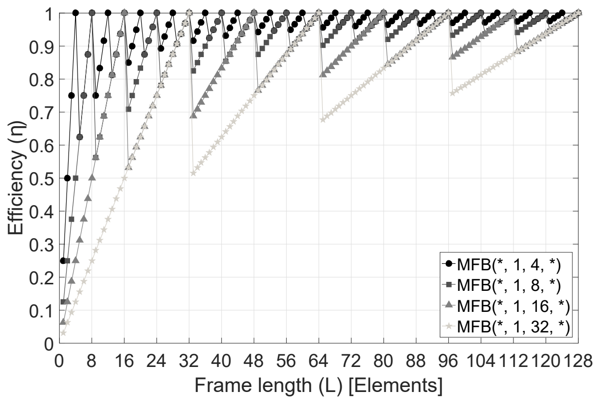

Fig. 3. Efficiency of MFBs with different block sizes (values of b).

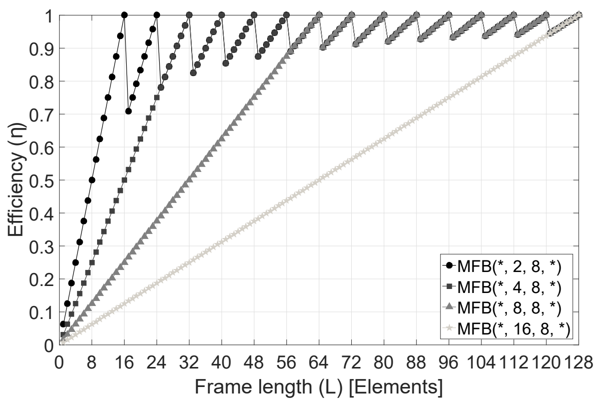

Fig. 4. Effect of larger regions on achievable efficiency of MFBs.

frames longer than r_e the values follow the same pattern as $MFB(*, 1, 8, *)$, resulting in a rather high efficiency.

A very common situation is that input data for the designed bus are decoded (decapsulated) from another protocol with its overhead being stripped away. Therefore, to properly adjust the MFB parameters, a more complex approach needs to be employed than just utilizing the worst-case bus efficiency. Furthermore, our processing protocol can add some kind of its overhead (encapsulation) that must be considered. For example, our input data can be L2 Ethernet frames after PCS/PMA layers decapsulation which removes the L1 overhead of 20 B per frame (Preamble and Interframe Gap). Also, our processing pipeline can append a few extra bytes to each frame with additional useful information like arrival timestamp or ingress interface number.

In the case described above, the aim is not just to achieve the highest worst-case efficiency of frame transfers over the bus, but rather to achieve a *wire-speed* processing of incoming data. As a result, it suffices that the bus achieves at least the same effective throughput as the input bus in every case. To incorporate the described notion of this comparative throughput efficiency, equations 3 and 4 are altered to:

$$\eta_C(L) = \frac{E_{in}(L)}{C(E_{out}(L))} \tag{6}$$

$$H_C = \min_{L=L_{min}}^{L_{max}} (\eta_C(L)) \tag{7}$$

Where $E_{in}(L)$ is the cost of the frame with length L in input encapsulation that has been stripped away and $E_{out}(L)$ stands for the cost of the frame in our specific encapsulation. Note that η_C and H_C can reach values higher than 1 because they just represent a comparative ratio between two efficiencies. Also, a minimum value of η_C can now be present in the longest frame lengths. So, to acquire desirable wirespeed throughput, the following condition derived from equation 5 must be satisfied for the worst-case:

$$T_{in} \leq H_C \times F \times n \times r_e \times e \tag{8}$$

Where T_{in} is the input throughput including E_{in} encapsulation cost. So for example, if 400 Gbps Ethernet is the input, the encapsulation cost function is $E_{in}(L) = L + 20$ and input throughput T_{in} is 400 Gbps.

After specifying the wire-speed throughput condition, it is now possible to derive a technique for MFB parametrization:

1) Value of e should be defined by the type of data that are dealt with. Also, T_{in}, E_{in} and E_{out} should be known.
2) Region size r_e is optimally equal to or slightly lower as minimal legal frame length. This is to avoid considerable efficiency degradation as seen in Figure 4.
3) As an effect of r_e on efficiency η_C and H_C is removed in the previous step, b remains the only parameter that can affect them. Using the largest possible value which preserves a reasonable H_C value (around 1) should be used. Also, after value of b is chosen, main attribute $r = r_e/b$ of MFB can be computed.
4) Finally, compute the required value of the remaining product $F \times n$. Explore trade-off between viable pairs of values and select the best one.

To show the example of the proposed technique in action, an example with Ethernet is again considered. Ethernet operates with bytes ($e = 8$), already mentioned $E_{in}(L) = L + 20$ and shortest frames of 64 B ($r_e = 64$). The optimal value of block size to achieve $H_C \approx 1$ is $b = 16$ ($H_C = 0.885$ for $b = 32$), considering MFB overhead of at most $b - 1$ bytes and given E_{in} with overhead of 20 B. Therefore, $r = 4$ and the optimal group of buses for Ethernet is $MFB(*, 4, 16, 8)$. Note that $MFB(*, 8, 8, 8)$ may be more convenient as PCS/PMA layers already align frames at 8 B blocks.

Continuing with the Ethernet example, the contribution of proposed MFB design is illustrated in Figure 5. The graph shows the efficiency of 400 Gbps Ethernet frames transferred over buses with different features. The darkest line with circles is the optimal MFB configuration that can achieve wire-speed throughput utilizing clock frequency of 196 MHz. The line with squares represents what happens when the unaligned frame starts and region sharing is not allowed – $H_C = 0.6875$ requiring 284 MHz for wire-speed, although it has the same bus word width. The remaining two lines are MFBs with inadequately big regions. They also illustrate the efficiency of buses without the support of multiple frames per clock cycle, but still supporting unaligned frame starts and shared words. Line with triangles shows the same efficiency that a 1024 b single-frame bus would have – $H_C = 0.6563$ requiring 596 MHz for wire-speed. A single-frame bus with a word width of 2048 b (same as optimal MFB) would have efficiency shown by the line with stars – $H_C = 0.3281$ requiring also 596 MHz for wire-speed.

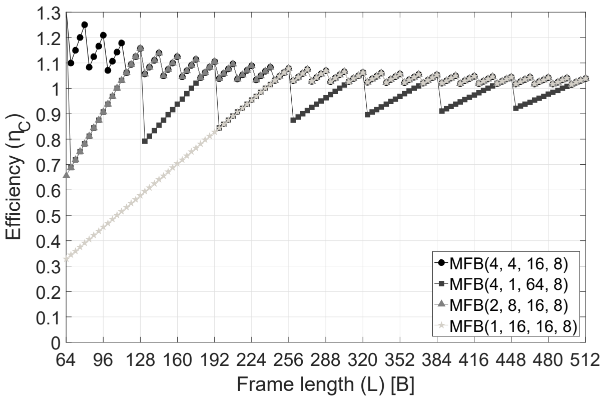

Fig. 5. Comparative efficiencies of different MFBs with 400 Gbps Ethernet.

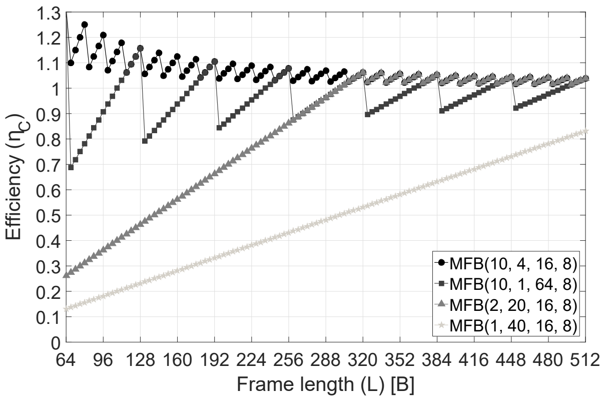

Fig. 6. Comparative efficiencies of different MFBs with 1 Tbps Ethernet.

The efficiency of single-frame buses is even worse in the processing of 1 Tbps Ethernet, as shown in Figure 6. The optimally configured MFB (line with circles) with reasonable region size retains its effectivity and can still achieve wire-speed processing easily. However, effective throughput of single-frame configurations (lines with triangles and stars) plummets to very low values, leading to requirements of around 1.5 GHz frequency for wire-speed processing.

IV. Implementation Methods

This chapter aims to find a general way of how to reduce the design and implementation complexity when creating processing cores for Multi Buses. A very convenient way would be to reduce the problem into two separate steps: create a core for a single-transaction bus processing and then replicate its logic for a given number of regions. To achieve such division, a closer look must be taken on the basic characteristics of the bus processing core structure. A simple

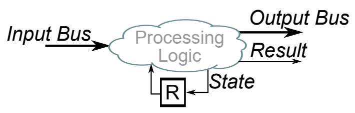

Fig. 7. Basic abstraction of processing core.

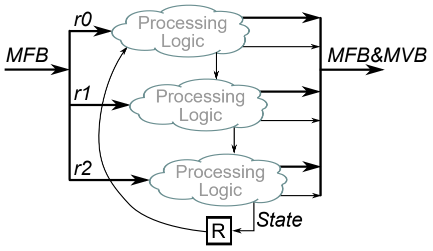

Fig. 8. Abstraction of processing core for MFB.

abstraction is shown in Figure 7. The logic of a processing core (gray cloud) somehow transforms the data on the bus and/or computes some additional information (Result) from them. For proper functionality, the logic also usually requires some kind of context (State) to be retained between the processing of individual words on the bus.

Now, a transition from single-transaction buses to proposed Multi Buses can be viewed as an attempt to process what would be multiple words of the original bus in a single word of the new Multi Bus. This influences the structure of processing architectures as illustrated in Figure 8 on MFB with 3 regions. Original processing logic is simply replicated for each region (gray clouds). The context (State) handling becomes more complex. The logic of each region now requires the state computed in the previous region and only the last one stores its computed state for the next word. This attempt to preform multiple state transitions in a single step leads to longer logic paths on input states of individual regions, possibly reducing the achievable frequency of the whole processing. Furthermore, the state update paths cannot be pipelined as the fully updated state is required directly in the next clock cycle to correctly process the next bus word.

To address the described potential performance degradation, a closer look at the general structure of processing logic should be taken. This leads to a more detailed abstraction as shown in Figure 9. Processing logic cloud is divided into three separate parts. Bus data processing starts with preprocessing, including all operations that do not require the knowledge of the current state. All state-dependent operations with bus data are grouped at the end in the postprocessing part. State update procedures (simplified notation $\oplus$) themselves are shown as a separate block. Preprocessing and postprocessing functionality and also their connections to state information can all be pipelined, as they are not part of the state update loopback. Therefore, the negative effects of longer state update paths in Multi Buses are effectively isolated to affect only $\oplus$ logic itself. Furthermore, all logic of a processing core apart from $\oplus$ is simply replicated for Multi Buses implementations.

Let us take a closer look at state update logic loopback

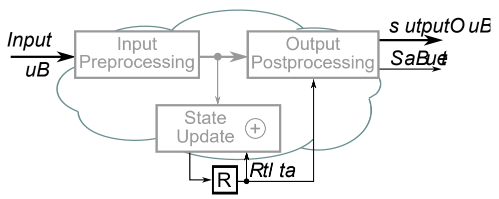

Fig. 9. More detailed abstraction of processing core for single-frame buses.

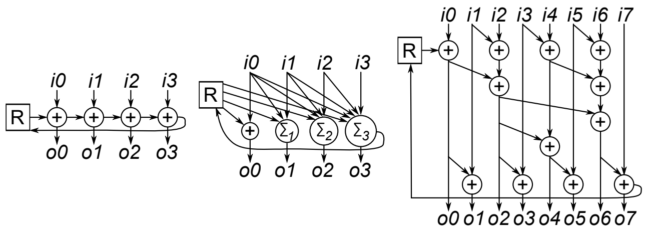

Fig. 10. State updates in Serial, Parallel, and PrefixSum form.

to optimize it. The first thing to do is to minimize the $\oplus$ procedure itself. This is achievable by thoroughly moving all unnecessary functionality into preprocessing or postprocessing stages. Next, complex state update procedures should be divided into multiple simpler parts that can be treated independently. Finally, the state update loopback connection of $\oplus$ procedures itself can be realized in different ways. We identified three possible solutions (Figure 10).

Serial form represents the basic connection schema as described so far. Each region has the same state update logic and the whole loopback is just a sequence of $\oplus$. Serial connection leads to $O(n)$ resource utilization and critical path length.

Parallel form requires a definition of $\sum_i$ functions that perform multiple steps of $\oplus$ at once. Now, each region has its own $\sum_i$ with appropriate size given by the number of data inputs (notice $\sum_0 = \oplus$). This arrangement leads to higher resources utilization but can reduce critical path length. The idea is that, compared to sequences of $\oplus$, a faster implementation of $\sum_i$ can be achieved.

PrefixSum form is based on parallel prefix sum algorithm [8]. It is relevant only when $\oplus$ is associative (the order of state updates does not change the outcome). Because $\oplus$ are connected into two trees (as illustrated) the resources utilization remains in $O(n)$, but critical path is only $O(\log(n))$.

Feasibility of individual implementation options is highly dependent on the structure of $\oplus$. Frequency and resource utilization trade-offs can be analyzed to select the best one.

V. Proof of Concept Results

To prove the feasibility of the proposed Multi Buses methodology, we implemented several cores following the techniques from the previous chapter. In the first part, we chose some basic functions for general processing of MFB frames and some MVB functions that might prove useful as pieces of state update handling (parts of $\oplus$) in more complex MFB engines. By measuring their performance in terms of FPGA resources utilization and achievable working frequency, we demonstrate that the proposed Multi Buses approach is indeed a practical and effective option for high throughput applications. In the second part, we briefly show actual results for selected complex MFB/MVB engines that we have described in detail (including specific related work) in our previous papers [9], [10], [11], [12], [13], [14].

A. Basic Functions

More detailed descriptions of implemented MFB and MVB cores functionality are provided in the following paragraphs, together with results (resources and frequency) for different parameters of MFB/MVB and implementation forms (serial, parallel and prefixsum). We provide results of implementation for Xilinx Virtex UltraScale+ FPGA XCVU9P with speed grade -3 using Vivado 2018.2. We focused on group $MFB(*, 8, 8, 8)$ that is appropriate for the processing of Ethernet frames at wire-speed ($\eta_C(L)$ is always 1 or higher). Tested values of n and m were also chosen based on their usability for 400 Gbps, 1 Tbps, and 2 Tbps Ethernet. In the tables, we displayed only results of the best implementation forms for each value of n or m.

Frame Counter (Table I) counts the number of frames transferred over the MFB. The results were measured for 64 b wide counter. For 32 b wide counter the frequencies are a bit better and resources are roughly halved. Also, the implementation is not influenced by most of the MFB parameters, so the same results are valid for all MFBs and MVBs (as Item Counter) with the same n or m. The nature of this task (only the value of the 'state' register is used) favors the parallel form.

Frame Tagger (Table II) assigns consecutive tags (sequence numbers) to frames transferred over MFB. These tags are then provided on the output MVB with valid items aligned with starts or ends of analyzed MFB frames. Transaction tags can be used for round-robin distribution (e.g. to multiple CPU cores) or as a mechanism for requests/responses pairing in architectures with inconsistent latency (e.g. hierarchical memory system). The results were measured for 16 b wide tags, which should provide enough unique values for most applications. Also, the implementation is not influenced by most of the MFB parameters, so the same results are valid for all MFBs and MVBs (as Item Tagger) with the same n or m.

Frame Measurer (Table III) acquires lengths of individual frames transferred over MFB. These lengths are then provided on the output MVB aligned with ends of analyzed MFB frames. Shown results were measured for 16 b wide representations of frame lengths, enough to support jumbo frames.

Element Extractor (Table IV) extracts data elements at given offset from MFB frame start and provides them over MVB on the output. This is an essential building block of packet header parsers. The results were measured for 4 B extraction and 16 b offset signal width. The state update loopback logic in this core consists of multiple simpler functions and their implementation forms are selected independently.

Last Valid Item (Table V) replicates each valid MVB item into all the following empty items until another valid one is reached. This can be useful in some complex MFB cores to retain state-related metadata of MFB frames throughout their whole bodies (e.g. element extraction offset or trimming length). Shown results are measured for 64 b wide items.

n	Best form	LUTs	FFs	F [MHz]
2	all	65	64	1 342
4	parallel	65	64	1 257
5	parallel	65	64	1 204
8	parallel	71	64	943
10	parallel	71	64	830

TABLE I
RESULTS FOR FRAME COUNTER.

n	Best form	LUTs	FFs	F [MHz]
2	all	33	16	1 647
4	parallel	65	16	1 515
5	parallel	81	16	1 447
8	parallel	134	16	1 114
10	parallel	172	16	884

TABLE II
RESULTS FOR FRAME TAGGER.

n	Best form	LUTs	FFs	F [MHz]
2	all	50	47	1 303
4	serial	99	81	1 101
5	serial	128	96	975
8	serial	200	149	873
	parallel	341	186	936
10	serial	265	183	735

TABLE III
RESULTS FOR FRAME MEASURER.

n	Best form	LUTs	FFs	F [MHz]
2	mixed	1 799	512	1 010
4	mixed	3 143	512	1 009
5	mixed	3 815	512	1 011
8	mixed	5 831	512	1 012
10	mixed	7 175	512	966

TABLE IV
RESULTS FOR ELEMENT EXTRACTOR.

m	Best form	LUTs	FFs	F [MHz]
2	all	132	65	1 841
4	serial	262	65	1 607
5	parallel	327	65	1 182
	prefixsum	425	65	1 569
8	serial	522	65	956
10	serial	844	65	900

TABLE V
RESULTS FOR LAST VALID ITEM.

Results of all described cores support the feasibility of the proposed Multi Buses approach. Achievable frequency is always well above the values required for considered Ethernet speeds. Also, the resources utilization scales reasonably with the widening of the data bus (raising value of n or m). Finally, proposed implementation forms (serial, parallel, prefixsum) provide various trade-offs between resources and frequency, where different form leads to the most effective results for individual cores.

B. Complex IP Cores

Using the proposed methodology, we designed and implemented several complex IP cores for key networking operations. A brief overview of the achieved results is presented in Table VI. For each IP core, we selected the best configurations (n or m and level of registering) to achieve at least a given

IP Core	Speed	n/m	F_{max}	LUTs	FFs	BRAMs
parser	400 GbE	2	441 MHz	18 476	24 900	-
		4	230 MHz	36 097	29 301	-
	1 TbE	8	334 MHz	73 395	77 653	-
	2 TbE	16	247 MHz	133 922	154 107	-
match table (IPv4)	400 GbE	2	545 MHz	1 818	887	46
	1 TbE	4	524 MHz	3 050	1 530	93
	2 TbE	8	451 MHz	5 510	2 818	186
match table (IPv6)	400 GbE	2	543 MHz	4 797	1 847	111
	1 TbE	4	521 MHz	8 092	3 237	222
	2 TbE	8	447 MHz	14 992	6 077	445
CRC	400 GbE	2	426 MHz	8 815	4 980	-
		4	294 MHz	16 711	6 057	-
	1 TbE	4	531 MHz	17 859	6 829	-
		8	320 MHz	36 738	14 401	-
	2 TbE	8	500 MHz	38 525	26 589	-
	0.2 Tbps	8*	798 MHz	6 340	3 728	-
deparser (switch)	200 GbE	2	280 MHz	5 559	8 451	-
	400 GbE	4	281 MHz	13 630	16 533	-
deparser (GRE)	200 GbE	2	296 MHz	22 906	9 781	15
	400 GbE	4	250 MHz	95 852	19 478	30

TABLE VI
RESULTS FOR THE SELECTED COMPLEX IP CORES.

Ethernet speed. We again provide results for the same MFB configuration, FPGA, and tool version. More detailed information for individual cores, especially specific architecture design and related work, can be found in their respective papers.

Parser [9] of frame headers has a modular design that allows specification of different protocol stacks. The shown results are for stack of Ethernet, multiple VLAN, multiple MPLS, IPv4/IPv6 (with extension headers), and TCP/UDP. Thanks to extensive configurable pipelining, the parser can scale up to 2 Tbps while utilizing only a few percents of FPGA resources.

Match table [10], [11] is based on Cuckoo hashing principle and enables exact matching of configurable keys. The shown results are for flow matching (the key is a 5-tuple of IP addresses, ports, and protocol) in IPv4 and IPv6 networks. Also, the effective rule capacity is configured to be above 10 000 flows with 32 bits of data for each. Thanks to effective memory replication scheme, the throughput can scale up to 2 Tbps (around 3 000 Mpps) in current FPGAs.

CRC [12], [13] computation is a critical operation in receiving and transmitting of Ethernet frames as well as in some memory interfaces. Therefore, CRC core must be optimized for resources as multiple instances are usually present in one FPGA firmware. The results in the table show, that only a fraction of FPGA resources will be used even when multiple instances are present. Note that the last row (denoted by *) do not contain results for Ethernet processing with $MFB(*, 8, 8, 8)$, but shows configuration for Hybrid Memory Cube interface [4] that operates with $MFB(*, 1, 1, 128)$.

Deparser [14] assembles the output frames based on changes required by processing pipeline. The table shows results for deparsing core in two applications: basic L2/L3 switch, and unpacking of GRE tunnels. Only speeds up to 400 Gbps can be achieved as this is currently a work in progress, more optimized architecture will be presented soon.

VI. CONCLUSION

This paper presents a new description method for the design of wide buses and processing cores that deal with multiple transactions per clock cycle. The method enables thorough performance analysis before any implementation effort even begins. It also provides guidelines for obtaining suitable bus configuration for a specific protocol and performance requirements. Also, three approaches for reducing the complexity of core design are presented and evaluated using examples of simple processing elements. Complex processing cores are then designed following the presented guidelines and show their feasibility. These cores perform critical operations of network traffic processing (parsing, matching, CRC checking, deparsing) and achieve throughput sufficient for wire-speed processing of 1 Tbps (and even faster) Ethernet traffic in current FPGAs. Overall, this paper provides formal guides that reduce the intuitive work of the designer and minimizes the chance of early bad design decisions. This method for wide bus design also relieves the need for ever-increasing chip frequency, since it becomes easier to use the chip area as the dimension to scale system throughput. We also see our work as a step towards automated design (high-level synthesis) of high-speed logic, outside of the traditional domain of digital signal processing.

ACKNOWLEDGMENTS

This research is partially supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 by the MEYS of the Czech Republic; the IT4Innovations excellence in science project IT4I XS – LQ1602; and by the Ministry of the Interior of the Czech Republic project VI20172020064.

REFERENCES

[1] I. Ganusov and B. Devlin, “Time-borrowing platform in the xilinx ultrascale+ family of fpgas and mpsocs,” in *26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016.

[2] D. Lewis et al., “The stratix™10 highly pipelined fpga architecture,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: ACM, 2016.

[3] M. Deo, J. Schulz, and L. Brown, *Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge*, Intel Corporation, 2016, white paper.

[4] Hybrid Memory Cube Consortium, *Hybrid Memory Cube Specification 2.1*, Altera Corporation, 2014.

[5] Various authors, *PCI Express®Base Specification Revision 3.0*, PCI-SIG.

[6] Xilinx, *UltraScale and UltraScale+ FPGAs Packaging and Pinouts*, Xilinx Inc., 2016, uG575.

[7] ARM, *AMBA AXI4-Stream Protocol Specification*, ARM, IHI 0051A.

[8] R. E. Ladner and M. J. Fischer, “Parallel prefix computation,” *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, October 1980.

[9] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, “Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput,” in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 249–258.

[10] M. Kekely, L. Kekely, and J. Kořenek, “Memory aware packet matching architecture for high-speed networks,” in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Aug 2018, pp. 1–8.

[11] M. Kekely, L. Kekely, and J. Korenek, “General memory efficient packet matching fpga architecture for future high-speed networks,” *Microprocessors and Microsystems*, vol. 73, 2020.

[12] J. Cabal, L. Kekely, and J. Kořenek, “High-speed computation of crc codes for fpgas,” in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec 2018, pp. 234–237.

[13] L. Kekely, J. Cabal, and J. Kořenek, “Effective fpga architecture for general crc,” in *Architecture of Computing Systems – ARCS 2019*. Cham: Springer International Publishing, 2019, pp. 211–223.

[14] “Scalable p4 deparser for speeds over 100 gbps,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 323–323.