# Embedded Systems Design with FPGAs

Peter Athanas • Dionisios Pnevmatikatos
Nicolas Sklavos
Editors

# Embedded Systems Design with FPGAs

Springer

*Editors*
Peter Athanas
Bradley Department of Electrical
and Computer Engineering
Virginia Tech
BLACKSBURG, Virgin Islands
USA

Dionisios Pnevmatikatos
Technical University of Crete
Crete, Greece

Nicolas Sklavos
KNOSSOSnet Research Group
Informatics & MM Department
Technological Educational Institute
of Patras, Greece

ISBN 978-1-4614-1361-5 ISBN 978-1-4614-1362-2 (eBook)
DOI 10.1007/978-1-4614-1362-2
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2012951421

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

This book presents methodologies for embedded systems design, using field programmable gate array (FPGA) devices, for the most modern applications. This manuscript covers state-of-the-art research from academia and industry on a wide range of topics, including applications, advanced electronic design automation (EDA), novel system architectures, embedded processors, arithmetic, and dynamic reconfiguration.

The book organization is based on 11 chapters, which cover different issues and deal with alternative scientific issues and industrial areas. The description of each chapter in a more analytical manner is as follows:

Chapter 1 presents a lightweight extension to statically scheduled microarchitectures for speculative execution: PreCoRe. Its judicious use of an efficient dynamic token model allows to predict, commit, and replay speculation events. Even if the speculation fails continuously, no additional execution cycles are required over the original static schedule. PreCoRe relies on MARC II, a high-performance multi-port memory system based on application-specific coherency mechanisms for distributed caches, and on RAP, a technique to efficiently resolve memory dependencies for speculatively reordered accesses.

The field which Chap. 2 deals with is decimal arithmetic. The importance of decimal for computer arithmetic has been further and definitely recognized by its inclusion in the recent revision of the IEEE-754 2008 standard for floating-point arithmetic. The authors propose a new iterative decimal divider. The divider uses the Newton–Raphson iterative method, with an initial piecewise approximation calculated with a minimax polynomial, and is able to take full advantage of the embedded binary multipliers available in today's FPGA technologies. The comparisons of the implementation results indicate that the proposed divider is very competitive in terms of area and latency and better in terms of throughput when compared to decimal dividers based on digit-recurrence algorithms.

Chapter 3 presents the design and mapping of a low-cost logic-level aging sensor for FPGA-based designs. The mapping of this sensor is designed to provide controlled sensitivity, ranging from a warning sensor to a late transition detector. It provides also a selection scheme to determine the most aging-critical paths at which the sensor should be placed. Area, delay, and power overhead of a set of sensors mapped for most aging-critical paths of representative designs are very modest.

Chapter 4 is devoted to complex event processing (CEP), which extracts meaningful information from a sequence of events in real-time application domains. This chapter presents an efficient CEP framework, designed to process a large number of sequential events on FPGAs. Key to the success of this work is logic automation generated with our C-based event language. With this language, both higher event-processing performance and higher flexibility for application designs than those with SQL-based CEP systems have been achieved.

Chapter 5 outlines an approach to model the DPR datapath early in the design cycle using queueing networks. The authors describe a method of modeling the reconfiguration process using well-established tools from queueing theory. By modeling the reconfiguration datapath using queueing theory, performance measures can be estimated early in the design cycle for a wide variety of architectures with nondeterministic elements. This modeling approach is essential for experimenting with system parameters and for providing statistical insight into the effectiveness of candidate architectures. A case study is provided to demonstrate the usefulness and flexibility of the modeling scheme.

Chapter 6 is dedicated to switch design for soft interconnection networks. The authors first present and compare the traditional implementations that are based on separate allocator and crossbar modules, and then they expand the design space by presenting new soft macros that can handle allocation and multiplexing concurrently. With the new macros, switch allocation and switch traversal can be performed simultaneously in the same cycle, while still offering energy-delay efficient implementations.

Chapter 7 presents advanced techniques, methods, and tool flows that enable embedded systems implemented on FPGAs to start up under tight timing constraints. Meeting the application deadline is achieved by exploiting the FPGA programmability in order to implement a two-stage system start-up approach, as well as a suitable memory hierarchy. This reduces the FPGA configuration time as well as the start-up time of the embedded software. An automotive case study is used to demonstrate the feasibility and quantify the benefits of the proposed approach.

Chapter 8 looks at the structure of a scalable architecture where the number of processing elements might be adapted at run-time, by means of exploiting a run-time variable parallelism throughout the dynamic and partial reconfiguration feature of modern FPGAs. Based on this proposal, a scalable deblocking filter core, compliant with the H.264/AVC and SVC standards, has been designed. This scalable core allows run-time addition or removal of computational units working in parallel.

Chapter 9 introduces a new domain-specific language (DSL) suited to the implementation of stream-processing applications on FPGAs. Applications are described as networks of purely dataflow actors exchanging tokens through unidirectional channels. The behavior of each actor is defined as a set of transition rules using pattern matching. The suite of tools currently comprises a reference interpreter and a compiler producing both SystemC and synthesizable VHDL code.

In Chap. 10, two compact hardware structures for the computation of the CLEFIA encryption algorithm are presented, one structure based on the existing state of the art and another a novel structure with a more compact organization. The implementation of the 128-bit input key scheduling in hardware is also herein presented. This chapter shows that, with the use of the existing embedded FPGA components and a careful scheduling, throughputs above 1 Gbit/s can be achieved with a resource usage as low as 238 LUTs and 3 BRAMs on a Virtex-4 FPGA.

Last but not least, Chap. 11 proposes a systematic method to evaluate and compare the performance of physical unclonable functions (PUFs). The need for such a method is justified by the fact that various types of PUFs have been proposed so far. However, there is no common method that can fairly compare them in terms of their performances. The authors propose three generic dimensions of PUF measurements and define several parameters to quantify the performance of a PUF along these dimensions. They also analyze existing parameters proposed by other researchers.

Throughout the above chapters of the book the reader has a deep point of view in detailed aspects of technology and science, with state-of-the-art references to the following topics like:

- A variety of methodologies for modern embedded systems design
- Implementation methodologies presented on FPGAs
- A wide variety of applications for reconfigurable embedded systems, including communications and networking, application acceleration, medical solutions, experiments for high energy, cryptographic hardware, inspired systems, and computational fluid dynamics

The editors of the *Embedded Systems Design with FPGAs* book would like to thank all the authors for their high-quality contributions. Special thanks must be given to the anonymous reviewers, for their valuable and useful comments on the included chapters.

Last but not least, special thanks to Charles Glaser and his team in Springer for the best work they all did regarding this publication.

We hope that this publication will be a reference of great value for the scientists and researchers to move forward with added value, in the areas of embedded systems, FPGAs technology, and hardware system designs.

Blacksburg, VA, USA Peter Athanas
Chania, Crete, Greece Dionisios Pnevmatikatos
Pyrgos, Greece Nicolas Sklavos

# Contents

# Widening the Memory Bottleneck by Automatically-Compiled Application-Specific Speculation Mechanisms

**Benjamin Thielmann, Jens Huthmann, Thorsten Wink, and Andreas Koch**

## 1 Introduction

The rate of improvement in the single-thread performance of conventional central processing units (CPUs) has decreased significantly over the last decade. This is mainly due to the difficulties in obtaining higher clock frequencies. As a consequence, the focus of development has shifted to multi-threaded execution models and multi-core CPU designs instead. Unfortunately, there are still many important algorithms and applications that cannot easily be rewritten to take advantage of this new computing paradigm. Thus, the performance gap between parallelizable algorithms and those depending on single-thread performance has widened significantly. Application-specific hardware accelerators with optimized pipelines are able to provide improved single-thread performance but have only limited flexibility and require high development effort compared to programming software-programmable processors (SPPs).

Adaptive computing systems (ACSs) combine the high flexibility of SPPs with the computational power of a reconfigurable hardware accelerator (e.g., using field-programmable gate arrays, FPGA). While ACSs offer a promising alternative compute platform, the compute-intense parts of the applications, the so-called kernels, need to be transformed to hardware implementations, which can then be executed on the reconfigurable compute unit (RCU). Not only performance but also better usability are key drivers for a broad user acceptance and thus crucial for the practical success of ACSs. To this end, research for the past decade has focused not only on ACS architecture but also on the development of appropriate tools which

B. Thielmann (✉) • J. Huthmann • T. Wink • A. Koch
Embedded Systems and Applications Group, Technische Universität Darmstadt, FB20 (Informatik), FG ESA, Hochschulstr. 10, 64289 Darmstadt, Germany
e-mail: thielmann@esa.cs.tu-darmstadt.de; huthmann@esa.cs.tu-darmstadt.de; wink@esa.cs.tu-darmstadt.de; koch@esa.cs.tu-darmstadt.de

enhance the usability of adaptive computers. The aim of many of these projects is to create hardware descriptions for application-specific hardware accelerators automatically from HLL such as C.

To achieve high performance, the parallelism inherent to the application needs to be extracted and mapped to parallel hardware structures. Since the extraction of coarse-grain parallelism (task/thread-level) from sequential programs is still a largely unsolved problem, most practical approaches concentrate on exploiting instruction-level parallelism (ILP). However, ILP-based speedups are often limited by the memory bottleneck. Commonly, only 20 % of the instructions of a program are memory accesses, but they require up to 100x the execution time of the register-based operations [12]. Furthermore, memory data dependencies also limit the degree of ILP from tens to (at the most) hundreds of instructions, even if support for unlimited ILP in hardware is assumed [9].

For this reason, memory accesses need to be issued and processed and dependencies resolved as quickly as possible. Many proposed architectures for RCUs rely on local low-latency high-bandwidth on-chip memories to achieve this. While these local memories have become more common in modern FPGA devices, their total capacity is still insufficient for many applications, and low-latency access to large off-chip memory remains necessary for many applications.

As another measure to widen the memory bottleneck for higher ILP, *speculative* memory accesses can be employed [6]. We use the general term “speculative” to encompass uncertain values (has the correct value been delivered?), control flow (has the correct branch of a conditional been selected and is the access needed in this branch?), and data dependency speculation (have data dependencies been resolved?). To efficiently deal with these uncertainties (e.g., by keeping track of speculative data and resolving data dependencies as they occur), hardware support in the compute units is required. We will describe an approach that efficiently generates these hardware support structures in an application-specific manner from a high-level description (C program), instead of attempting to extend the RCU with a general-purpose speculation block. To this end, we will present the speculation-handling microarchitecture PreCoRe, the HLL hardware compile flow Nymble, and the back-end memory system MARC II, which was tuned to support the speculation mechanisms.

## 2 Overview

The development of a compiler and an appropriate architecture is a highly interdependent task. Most of the HLL to hardware compilers developed so far use static scheduling for their generated hardware datapaths. A major drawback of this approach is its handling of variable-latency operators, which forces a statically scheduled datapath to completely stall all operations on the accelerator until the delayed operation completes. Such a scenario is likely to occur when accessing cached memories and the requested data cannot be delivered immediately. Dynamic scheduling can overcome this issue but has drawbacks such as its complex execution model, which results in considerable hardware overhead and lower clock rates. Furthermore, in itself, it does not address the memory bottleneck imposed by the high latencies and low bandwidth of external memory accesses.

Due to these limitations, RCUs are becoming affected by the processor/memory performance gap that has been plaguing CPUs for years [9]. But since RCU performance depends heavily on exploiting parallelism with hundreds of parallel operators, RCUs suffer a more severe performance degradation than CPUs, which generally have only few parallel execution units in a single core.

The quest for high parallelism in ACSs further emphasizes this issue. Control flow parallelism allows to simultaneously execute alternative precluding branches, such as those in an if/else construct, even before the respective control condition has been resolved. However, such an exploitation of parallel control flows may cause additional memory traffic. In the end, this can even slow down execution over simpler less parallel approaches.

A direct attempt to address the negative effect of long memory access latencies and insufficient memory bandwidth is the development of a sophisticated multi-port memory access system with distributed caches, possibly supported by multiple parallel channels to main memory [16]. Such a system performs best if many *independent* memory accesses are present in the program. Otherwise, the associated coherency traffic would become a new bottleneck. Even though this approach helps to benefit from the available memory bandwidth and often reduces access latencies, stalling is still required whenever a memory access cannot be served directly from one of the distributed caches.

Load value speculation is a well-studied but rarely used technique to reduce the impact of the memory bottleneck [18]. Mock et al. were able to prove by means of a modified C compiler, which forced data speculation on an Intel Itanium 2 CPU architecture where possible, that performance increases due to load value speculation [21] of up to 10 % were achievable. On the other hand, the Itanium 2 rollback mechanism, which is based on the advanced load address table (ALAT), a dedicated hardware structure that usually needs to be explicitly controlled by the programmer [19], produces performance losses of up to 5 % under adverse conditions with frequent misspeculations.

Research on data speculation methods and their accuracy has produced a broad variety of data predictors. History-based predictors select one of the previously loaded values as the next value, solely based on their occurrence probability. Stride predictors do not store absolute values, but determine the offset between the successive loaded values. Here, instead of an absolute value, the most likely *offset* is selected. In this manner, sequences with constant offset between elements can be predicted accurately. Both techniques have proven to be beneficial and do not require long learning time, but both fail to provide good results for complex data sequences. Thus, more advanced techniques, such as context-based value predictors, predict values or strides as the function of a previously observed data *sequence* [23]. Performance gains are achievable if the successful prediction rate is high, or if the penalty to recover from misspeculations is very low.

The load value speculation technique is especially beneficial for statically scheduled hardware units, since now even the variable-latency cached read operations give the appearance of completing in constant time (by returning a speculated value on cache misses). This allows subsequent operations to continue to compute speculatively, instead of stalling non-productively. As the predicted values may turn out to be incorrect, the microarchitecture must be extended to re-execute the affected parts of the computation with correct operands (replayed), and commit only those results computed from values that were either correctly speculated or actually retrieved from memory. In this approach, memory reads are the sole source of speculative data, but intermediate computations may be affected by multiple reads. Even a correct speculation might be poisoned by a later incorrectly speculated read value. Ideally, only those computations actually affected by the misspeculated value need to be replayed. While this could be handled at the granularity of individual operators, it would require complex control logic similar to that of dynamically scheduled hardware units. As an alternative, our proposed approach will manage speculation on groups of operators organized as *Stages*, which are similar to the start cycles in a static schedule.

It is important to note that by continuing execution speculatively, an increased number of memory read accesses are issued and then possibly replayed once or several times, increasing the pressure on the memory system even more. Additionally, data dependency violations are likely to occur in such an out-of-order execution of accesses and also need to be managed. We propose prioritization and data dependency resolution schemes to address these issues at run-time.

The speculation support mechanisms, collectively named PreCoRe, are lightweight extensions to a statically scheduled datapath; they do not require the full flexibility (and corresponding overhead) of datapaths dynamically scheduled at the level of individual operators. PreCoRe focuses on avoiding slow-downs of the computation compared to a nonspeculative version (by not requiring additional clock cycles due to speculation overhead), even if all speculations would fail continuously. The PreCoRe microarchitecture extensions are automatically generated in an application-specific manner using the Nymble C-to-hardware compiler. At run-time, they rely on the MARC II memory subsystem to support parallel memory accesses and handle coherency issues. Together, these components provide an integrated solution to enable efficient speculative execution in ACS.

## 3 The PreCoRe Speculation Framework

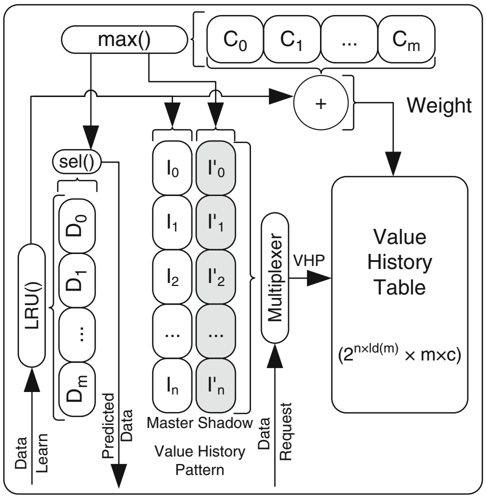
PreCoRe (predict, commit, replay) is a new execution paradigm for introducing load value speculation into statically scheduled data paths: Load values are predicted on cache misses to hide the access latency on each memory read request. Once the true value has actually been retrieved from the memory, one of two operations must happen: If the previous speculatively issued value matches the actual memory data, PreCoRe commits all dependent computations which have been performed using the speculative value in the meantime as being correct. Otherwise, PreCoRe reverts those computations by eliminating speculatively generated data and issuing a replay of the affected operations with corrected values. To implement the PreCoRe operations, three key mechanisms are required. First, a load value speculation unit is needed to generate speculative data for each memory read access within a single clock cycle. Second, all computations are tagged with tokens indicating their speculation state. The token mechanism is also used to commit correct or to eliminate speculative computations. Third, specialized queues are required to buffer intermediate values, both before they are being processed and to keep them available for eventual replays. All three key mechanism will be introduced and discussed in this section.

## *3.1 Load Value Speculation*

Evidently the benefit achieved by speculation is highly dependent on the accuracy of the load value prediction. Fortunately, data speculation techniques have been well explored in the context of conventional processors [3, 27].

It is not possible in the spatial computing paradigm (with many distributed loads and stores) to efficiently realize a predictor with a global perspective of the execution context. This is the opposite of processor-centric approaches, which generally have very few load store units (LSU) that are easily considered globally. On an RCU, the value predictors have a purely local (per-port) view of the load value streams. This limited scope will have both detrimental and beneficial effects: On one hand, a predictor requires more training to accumulate enough experience from its own local data stream to make accurate prediction. On the other hand, predictors will be more resilient against irregular data patterns (which would lead to deteriorated accuracy) flowing through other memory ports.

Using value speculation raises the question of how to train the predictors, specifically, when the underlying pattern database (on which future predictions are based) should be updated: Solely if a speculation has already been determined as being correct/incorrect? Since this could entail actually waiting for the read of main memory, it might take considerable time. Or should the speculated values be assumed to be correct (and entered into the pattern database) until proven incorrect later? The latter option was chosen for the PreCoRe, because a single inaccurate prediction will always lead to the re-execution of all later read operations, now with pattern databases updated with the correct values. The difference to the former approach is that the predictor hardware needs to be able to rollback the entire pattern database (and not just individual entries) to the last completely correct state once a speculation has proven to be incorrect. One of the overarching goals of PreCoRe remains to support these operations without slowing down the datapaths over their non-speculative versions (see Sect. 6).



| | |
|---|---|
| $C_m$ | Probability of value $D_m$ |
| $D_m$ | Stored values |
| $I_n$ | Last confirmed value index |
| $I'_n$ | Last speculative value index |
| m | Number of stored values |
| n | Length of stored value sequence |
| c | Number of bits for storing probability |

**Fig. 1** Local history-based load value predictor

### 3.1.1 Predictor Architecture

The value predictors (shown in Fig. 1) follow a two-level finite-context scheme, an approach that was initially used in branch prediction. The predictions exploit a correlation of a stored history of prior data values to derive future values [27]. The precise nature of the correlation is flexibly parametrized: The same base architecture is used to realize both last-value prediction (which predicts a future value by selecting it from a set of previously observed values, e.g., *23-7-42-23-7-42*) and stride prediction (which extrapolates a new value from a sequence of previously known strides, e.g., from the strides *4-4-8-4*, the sequence *0-4-8-16-20-24-28-36-40* is predicted). A PreCoRe value prediction unit operates parallel last-value and stride sub-predictors in tournament mode, where a sub-predictor is trusted until it mispredicts, leading to a switch to the other sub-predictor. Since both sub-predictors use the same micro-architecture (with exception of the correlation computation), we will focus the discussion on just one mode, namely the value-speculation.

The predictor not only keeps track of the last $m$ different values $D_1, \ldots, D_m$ in a least recently used fashion in its pattern database $D$ but also maintains the $n$-element sequence $I_1, \ldots, I_n$ in which these values occurred (the value history pattern, VHP). Each of the $n$ elements of $I$ is an $\lceil \log_2 m \rceil$ bit wide field holding an index reference to an actual value stored in $D$. $I$ is used in its entirety to index the value history table (VHT) to determine the most likely of the currently known values: Each entry in the VHT expresses the likelihood for all of the known values $D_i$ as a $c$-bit unsigned counter $C_i$, with the highest counter indicating the most likely value (on ties, the smallest $i$ wins). The VHT is thus accessed by a $n \cdot \lceil \log_2 m \rceil$-bit wide address and stores $m \cdot c$-bit-wide words. On start-up, each VHT counter is initialized to the value $2^{c-1}$, indicating a value probability of $\approx 50\,\%$.

To handle mispredictions, we keep two copies of the VHP as $I$ and $I'$: $I$ is the master VHP, which stores *only* values that were already confirmed as being correct by the memory system. However, the stored values may be *outdated* with respect to the actual execution (since it might take awhile for the memory system to confirm/refute the correctness of a value). The shadow VHP $I'$ (shown with gray background in the figure) additionally includes speculated values of *unknown* correctness. It accurately reflects the *current* progress of the execution. Values will be predicted based on the shadow VHP until a misprediction is discovered. The computation in the datapath will then be replayed using the last values not already proven incorrect. A similar effect is achieved in the predictor by copying the master VHP $I$ (holding correct values) to the shadow VHP $I'$ (basing the next predictions on the corrected values). [24] explains the predictor in greater detail and shows a step-by-step example of its operations.

The predictor is characterized by the two parameters $n$ and $m$. The first is the maximum length of the context sequence, the second the maximum number of different values tracked. The state size of VHT and VHP (and thus the learning time before accurate predictions can be made) grows linearly in $n$ and logarithmically in $m$. Note that in a later refinement, optimum values for $n$ and $m$ could be derived by the compiler using profile-guided optimization methods.

## 3.2 Token Handling Mechanisms

The PreCoRe mechanisms are inserted into the datapath and controller of a statically scheduled hardware unit. They are intended to be automatically created in an application-specific manner by the hardware compiler. With the extensions, cache-misses on reads no longer halt execution due to violated static latency expectations, but allow the computation to proceed using speculated values. Variable-latency reads thus give the appearance of being fixed-latency operators that always produce/accept data after a single cycle (as in cache-hit case).

In this manner, predicted or speculatively computed values propagate in the datapath. However, only reversible (side effect-free) operations may be performed speculatively to allow replay in case of a misprediction. In our system, write operations thus form a *speculation boundary*: A write may only execute with operand values that have been confirmed as being correct. If such a confirmation is still absent, the write will stall until the confirmation arrives. Should the memory system refute the speculated values, the entire computation leading up to the write will be replayed with the correct data.

This is outlined in the example of Fig. 2a. Here, the system has to ensure that the data to be written has been correctly predicted in its originating READ node (the sole source of speculated data in the current PreCoRe prototype) before the WRITE node is allowed to execute. This is achieved for the READ by comparing the predicted read result, which is retained for this purpose in an output queue in the READ node, with

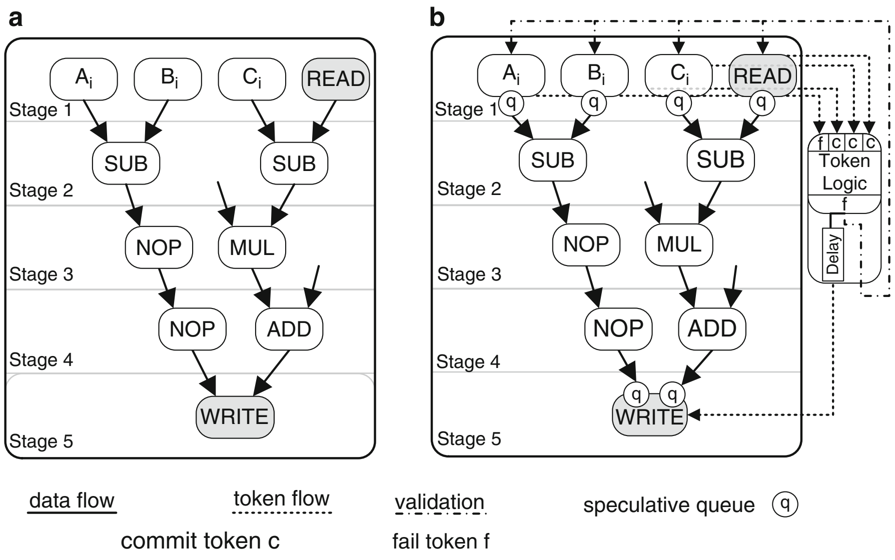


**Fig. 2** Datapath and speculation token processing

the actual value received later from the memory system. Until the comparison has established the correctness of the predicted value, the data to be written (which was computed depending on the predicted read value) is held in an input queue at the WRITE node. This queue also gives the WRITE node the appearance of a single-cycle operation, even on a cache-miss.

Figure 2b sketches the extension of the initial statically scheduled datapath with PreCoRe: Explicit tokens track the speculativity of values and their confirmation/refutation events. This is indicated by additional edges that show the flow of tokens and validation signals.

As an example, if the READ node has confirmed a match between predicted and actual data values, it indicates this by sending a commit-token (shown as C in the figure) to the token logic. However, to reduce the hardware complexity, this token is *not* directly forwarded to the WRITE node waiting for this confirmation, as would be done in operator-level speculation. Instead, the speculativity is tracked per data path stage (corresponding to the operators starting in the same clock cycle in a static schedule). Only if *all* operators in a stage confirm their outputs as being correct is the C-token actually forwarded to the WRITE operator acting as speculation boundary, confirming as correct the oldest WRITE operand with uncertain speculation status. Speculated values and their corresponding C- and F-tokens (indicating failed speculation) always remain in order. Thus, no additional administrative information, such as transaction IDs or similar, is required. Tokens are allowed to temporarily overtake their associated data values up to the next synchronization point (see Sect. 3.3) by skipping stages that lack speculative operators (READ nodes). The speculation output status of a stage depends on that of its inputs: It will be non-speculative, if no speculative values were input, and speculative, if even a single input to the stage was speculative. In the example, stages 2–4 do not contain READs, the C-token can thus be directly forwarded to the WRITE in stage 5, where it will be held in a token queue until the correctly speculated value arrives and allows the WRITE to proceed. In parallel to this, pipelining will have led to the generation of more speculative values in the READ, which continue to flow into the subsequent stages.

If the initial speculation in the READ node failed (the output value was discovered to be mispredicted), all data values which depended on the misspeculated value have to be deleted, and the affected computations have to be replayed with the correct no-longer speculative result of the READ. This is achieved by the token logic recognizing that the misspeculated READ belonged to stage 1, and thus the entire stage is considered to have misspeculated. All stages relying on operands from stage 1 will be replayed. The F-token does not take effect immediately (as the C-token did), but is delayed by the number of stages between the the speculated READ and the WRITE at the speculation boundary. In the example, the F-token will be delayed by three stages, equivalent to three clock cycles of the datapath actually computing. If the datapath were stalled (e.g., all speculative values have reached speculation boundaries but could not be confirmed yet by memory accesses because the memory system was busy), these stall cycles would not count towards the required F-token delay cycles. Delaying the effect of the F-token ensures that the intermediate values computed using the misspeculated value in stages 2–4 have actually arrived in the input queues of the WRITE operation in stage 5 and will be held there since no corresponding C- or F-token for them was received earlier. At this time, the delayed F-token arrives at the WRITE and deletes three sets (corresponding to the three intermediate stages) of potentially incorrect input operands from the WRITE input queues and thus prevents it from executing. The replay of the intermediate computation starts immediately once the last attempt has been discovered to have used misspeculated values. Together with the correct value from the READ (retrieved from memory), the other nodes in stage 1 re-output their last results (which may still be speculative themselves!) from their output queues and perform the computations in stages 2–4 again. A more detailed example of token handling is given in [25].

## *3.3 Queue Management for Speculation*

First introduced in the previous section, operator output queues (re)supply the data to allow replay operations and are thus essential components of the PreCoRe architecture. Note that some or all of the supplied values may be speculative. Data values are retained until all outputs of a stage have been confirmed and a replay using these values will no longer be required. Internally, each queue consists of separate sub-queues for data values and tokens, with the individual values and tokens being associated by remaining strictly in order: Even though tokens may overtake data

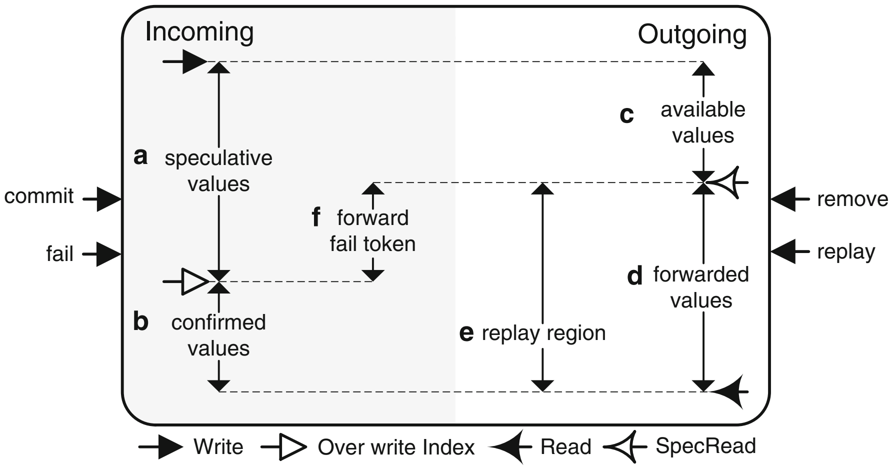


**Fig. 3** Value regions in speculative queue

values between stages, their sequence will not be changed. In our initial description, we will concentrate on the more complex output queues. Input queues are simpler and will be discussed afterwards.

Figure 3 gives an overview of an output queue, looking at it from the incoming (left side) and outgoing (right side) perspectives.

On the incoming side, values are separated into two regions: speculative values (a) and confirmed values (b). Since all data values are committed sequentially and no more committed data may arrive once the first speculative data entered the queue, these regions are contiguous. Similarly, outgoing values are in different contiguous regions depending on their state: (d) is the region of values that have already been forwarded as operands to a consumer node and are just retained for possible replays and (c) is the values that are available for forwarding. Conventional queue behavior is realized using the Write pointer to insert newly incoming speculative data at the start of region (a) and the Read pointer to remove a value from the end of region (d) after the entire stage has been confirmed. Two additional pointers are required to implement the extra regions: Looking into the queue from the outgoing end, SpecRead determines the first value which has *not* been forwarded yet at the end of region (c), and OverwriteIndex points to the last confirmed value at the beginning of region (b).

On misspeculation in a predecessor stage (indicated by an incoming F-token), the speculative values making up region (a) are discarded by setting Write back to OverwriteIndex. If the queue does not hold confirmed values (regions (c) and (d) are empty), the F-token is passed on (f) through the output queue into subsequent stages.

Confirmed values are forwarded to the consumer nodes but retained in the queue for replays (moving from region (c) into region (d) by manipulation of the SpecRead pointer). If operators in the same stage request a replay, SpecRead is reset to Read, making all of the already forwarded but retained values available again for re-execution of subsequent stages, with (f) now acting as a replay region (e). Retained values are removed from the queue *only* if all operators in the stage have confirmed their execution (and thus ruled out the need for a future replay). This final removal is achieved using the Read pointer. For a detailed example of the output queue operation, please refer to [25].

Input queues have a similar behavior but do not need to confirm speculative data (that was handled in their predecessor's output queue).

## 3.4 Dynamic Resolution of RAW Dependencies

The speculative PreCoRe execution scheme enables the prefetching of memory reads: A read which might originally be scheduled after a write is allowed to execute speculatively before the write has finished. This reordering potentially violates a read-after-write (RAW) memory data dependency. Thus, all of the memory read accesses potentially depending on the write must remain in speculative state until the memory write access itself has been committed. Static points-to/alias analysis in the compiler can remove some of the potential dependencies and guarantee that reads and writes will be to non-overlapping memory regions (allowing out-of-order prefetching). However, in most realistic cases, such guarantees cannot be assured at compile time. Instead, dynamic detection and correction of dependency violation due to speculatively prefetched reads must be employed to handle the general case. PreCoRe supports two such mechanisms.

*Universal Replay*: This approach is a straightforward, low-area, but suboptimal extension of the existing PreCoRe commit/replay mechanisms: *All* RAW dependency-speculated reads re-execute as soon as all writes have completed, regardless of whether an address overlap occurred. The number of affected reads is only limited by the PreCoRe speculation depth, which is the number of potentially incorrectly speculated intermediate results that can be rolled back. In PreCoRe, the speculation depth is determined by the length of speculation value queues on the stages between the possibly dependent read and write nodes.

In practice, universal replay is less inefficient as it appears at first glance: Assuming that the data written by all write operations is still present in the cache, the replays will be very quick. Also, in the scheme, *all* writes are initially assumed to induce a RAW violation. If a write is only conditionally executed, the potentially dependent reads can be informed if the evaluation of the control condition prevents the write from executing at all. This is communicated from each write to the reads using a Skip signal (see Fig. 4a). If all writes have been skipped, there no longer is a risk of a RAW violation and the data retrieved by the reads will be correct (and can be confirmed as such). On the other hand the replays in this scheme can become expensive if the write data has been displaced from the cache or if the replayed computation itself is very complex. Thus, it is worthwhile to examine a better dependency resolution scheme.

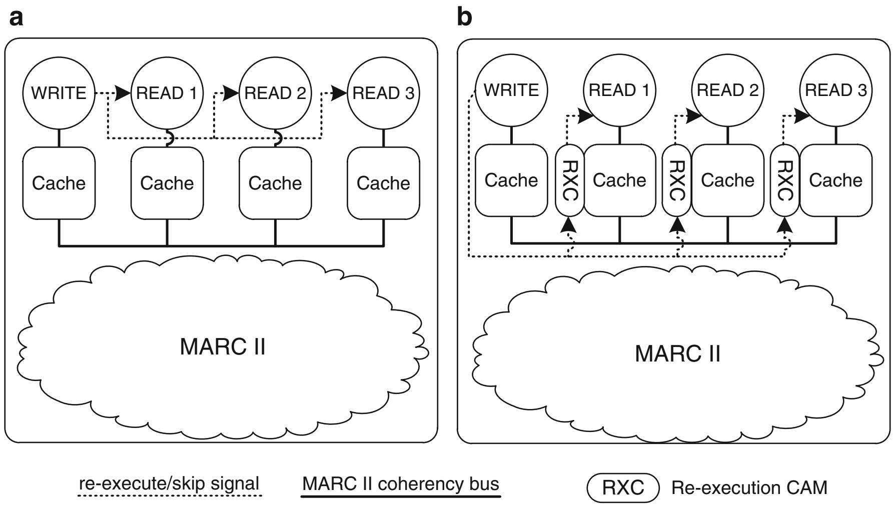


**Fig. 4** Resolution schemes for RAW memory dependencies

*Selective Replay*: This more refined technique avoids unnecessary replays by actually detecting individual read/write address overlaps on a per-port basis and replays only those RAW-speculated reads that were actually affected by writes. To this end, read ports in the memory subsystem are extended with dedicated hardware structures (RXC, see Sect. 5.3) to detect and signal RAW violations. Combined with the Skip signal to ignore writes skipped due to control flow, replays are only started for specific read nodes if RAW violations did actually occur.

## 3.5 *Access Prioritization*

PreCoRe fully exploits the spatial computing paradigm by managing operations on the independent parallel memory ports supplied by the MARC II memory subsystem (see Sect. 5). However, internally to MARC II, time-multiplexed access to shared resources, such as buses or the external memory itself, becomes necessary. By carefully prioritizing different kinds of accesses, the negative performance impact of such time multiplexing can be reduced. PreCoRe influences these priorities not only to best use the available bandwidth on the shared resources for useful accesses, but also to employ spare bandwidth to perform prefetching. A number of techniques are used to manage access priorities.

The simplest approach consists of statically allocating the priorities at compile time. In PreCoRe, the write port always executes with the highest priority, since it will only be fed with nonspeculative data and will thus always be useful. Read

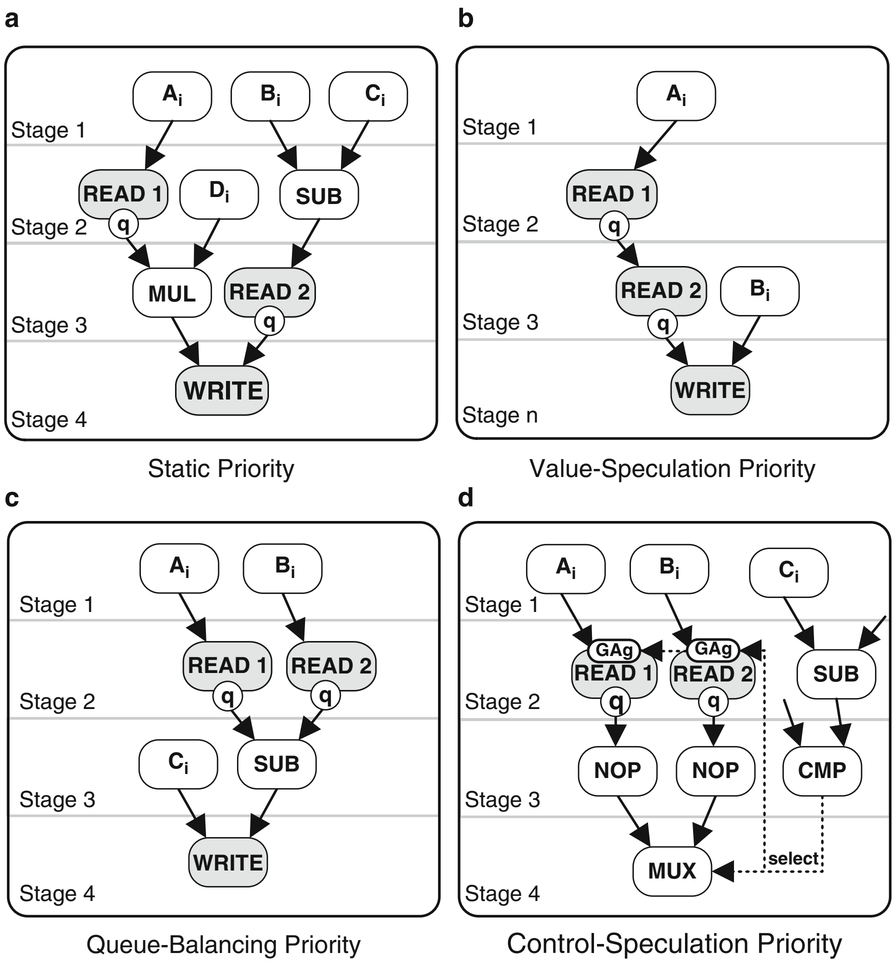


**Fig. 5** Scenarios for priority-based shared resource arbitration

operations placed early in the static schedule will be assigned a higher priority than read operations scheduled later, so their data will already be available when later stages execute. In Fig. 5a, READ1 thus executes with higher priority than READ2.

Figure 5b shows a scenario where the address of READ2 is dependent on the result of READ1. In PreCoRe, READ1 will provide a value-speculated result after a single clock cycle, which READ2 will use as address for prefetching. However, in doing so, it will hog the shared MARC II resources performing a potentially useless access (if READ1 misspeculated). These resources would have been better used to execute the non-address speculated READ1 of the *next* loop iteration, which is an access that will always be useful. Value-speculation priority dynamically lowers the priority of accesses operating on speculated addresses and/or data values, thus giving preferential treatment to accesses using known-correct operands.

In some situations, the simple static per-port priority can even lead to a loss of performance. This occurs specifically if the outputs of multiple reads at the same stage converge at a later operator. An example for this is shown in Fig. 5c. Here, the static priority would always prefer the read assigned to the lowest port number over another one in the same stage. Assuming READ1 had the lower port number, it would continue executing until its output queue was full. Only then would READ2 be allowed to fetch a single datum. A better solution is to dynamically lower the priority of reads already having a higher fill-level of non-speculated values (=actually fetched from memory) in their output queues.

As described above, performance gains may be achieved by allowing read operators to immediately reply with a speculated data value on a cache miss. Orthogonal to this data speculation approach is speculating on whether to execute the read operator at all. Such *control-speculation* is performed on SPP using branch prediction techniques. While this approach is not directly applicable in the spatially distributed computation domain of the RCU (all ready operators execute in parallel), it does have advantages when dealing with shared singleton resources such as main memory/buses: For software, branch prediction would execute only the most likely used read in a conditional, while the RCU would attempt to execute the reads on all branches of the conditional in parallel, leading to heavy competition for the shared resources and potentially slowing down the overall execution (on multiple parallel cache misses).

To alleviate the problem, we track which branch of a parallel conditional actually performed useful computations by recording the evaluated control condition. The read operators in that branch will receive higher priorities, thus preventing reads in less frequently taken branches from hogging shared resources. To this end, we use decision tracking mechanisms well established in branch prediction, specifically the GAg scheme [29], but add these to the individual read operators of the parallel conditional branches (see Fig. 5d). The trackers are connected to the controlling condition for each branch (see [16] for details) and can thus attempt to predict which branch of the past branching history will be useful next, prioritizing its read operators. In case of a misprediction, all mistakenly started read operations are quickly aborted to make the shared resources available for the actually required reads.

To exploit the advantages of the different schemes, they are all combined into a general dynamic priority computation:

$$\mathsf{P}_{\mathsf{dyn}}(r) = (\mathsf{W}_{\mathsf{q}} \cdot \mathsf{P}_{\mathsf{q}}(r) + (1 - \mathsf{W}_{\mathsf{q}}) \cdot \mathsf{P}_{\mathsf{hist}}(r)) \cdot 2^{-(\mathsf{W}_{\mathsf{spec}} \cdot \mathsf{IsSpec}(r))}.$$

The dynamic priority $\mathsf{P}_{\mathsf{dyn}}(r)$ for each read operator $r$ is thus computed from the queue-balancing priority $\mathsf{P}_{\mathsf{q}}(r)$, the control-speculation priority $\mathsf{P}_{\mathsf{hist}}(r)$ based on its GAg predictor, and its speculative predicate $\mathsf{IsSpec}(r)$, which is $= 1$ if $r$ is dynamically speculative for *any* reason (input address speculated and not yet confirmed, control condition not yet evaluated, still outstanding writes for RAW dependency checks), and $= 0$ otherwise. This predicate will be used to penalize the priority of speculative accesses. $\mathsf{W}_{\mathsf{x}}$ are static weights that can be set on a per-application basis, potentially even automatically by sufficiently advanced analysis in the compiler. $\mathsf{W}_{\mathsf{q}}$ is used to trade-off between queue balancing and control history prediction, while $\mathsf{W_{spec}}$ determines the priority penalty for speculative accesses. See [26] for a more detailed discussion and an evaluation of the performance impact of these parameters.

## 4 The Nymble C-to-Hardware Compiler

To discuss the integration of PreCoRe into the Nymble compile flow, we will first give an overview of the initial C-to-hardware compilation process. It relies on classical high-level synthesis techniques to create synthesizable RTL descriptions of the resulting statically scheduled hardware units.

### *4.1 Control-Data Flow Graph Generation*

We use a simple program computing the factorial function (Listing 1) as running example for the compilation flow.

The Nymble front end relies on traditional compiler techniques (specifically, those of the Scale framework [7, 28]) to lex and parse the input source code and perform machine-independent optimizations, finally representing the program as control flow graph (CFG) in static single assignment (SSA) form [1]. Figure 6 shows the SSA-CFG of our sample program. SSA-CFGs are a commonly used intermediate representation in modern software compilers and well suited for the actual hardware compilation.

In SSA form, each variable is written only once, but may be read multiple times. Multiple assignments to the same variable create new value instances (versions of the variable, often indicated by subscripts). A $\Phi$ function selects the current value instance when multiple value instances converge at a CFG node. This happens, e.g., for conditionals for the true and false branches or for the entering and back edge of a loop.

**Listing 1** Sample program for hardware compilation

```
int factorial(int n) {
  int a = 1;
  int i;

  for(i=1; i<n; ++i}
    a = a * i;

  return a;
}
```

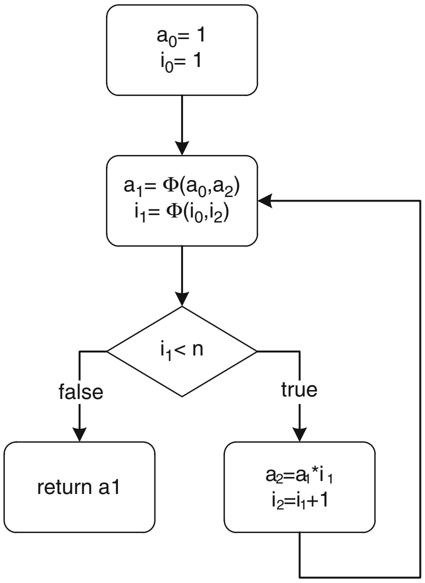


**Fig. 6** SSA-CFG of sample program

Given the abundance of flip-flops on most current reconfigurable devices, the value instances of the SSA form could be mapped directly to hardware registers (but also see comment below). To build the computations between the registers, the data flow from source variables through operators to destination variables has to be extracted. This is easily achievable in SSA form, since a value can originate only from one specific assignment. The flow of values is expressed as a data flow graph (DFG), shown in Fig. 7 for the example.

While it would suffice for the synthesis of the datapath of the hardware unit (by mapping the operators to compute nodes and the edges to appropriate wiring), the control flow (e.g., the loop termination condition) must still be considered when synthesizing the controller. This is achieved by extending the DFG with control edges (shown as dotted lines in Fig. 8, labeled on which boolean value of the controlling condition they activate). Control edges carry the boolean results of conditions to either activate specific nodes (e.g., the end node indicating the completion of hardware execution) or select which value instance to pass through the multiplexers representing the $\Phi$ functions. For the loops shown here, the $\Phi$ functions at the loop heads are controlled by a dedicated init node that outputs true on its control edge if loops are being entered for the first time and false otherwise..

As a refinement of mapping SSA value instances to registers, it is possible to remove purely intermediate variables and replace them by simple wiring to their computing operator in the DFG, instead of allocating a hardware register to hold the intermediate result.

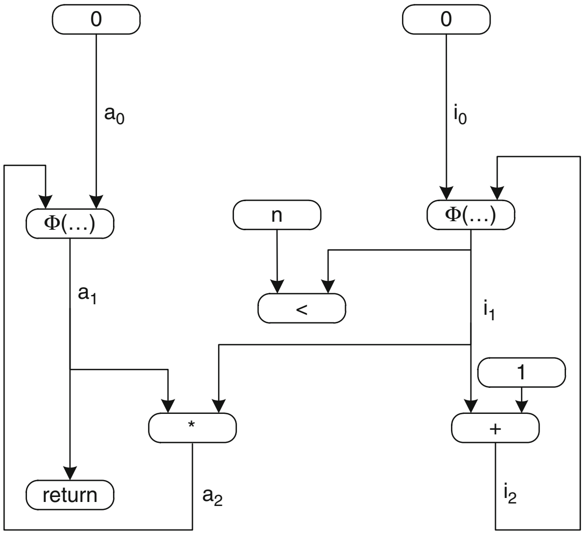


**Fig. 7** Data flow graph (DFG) of sample program

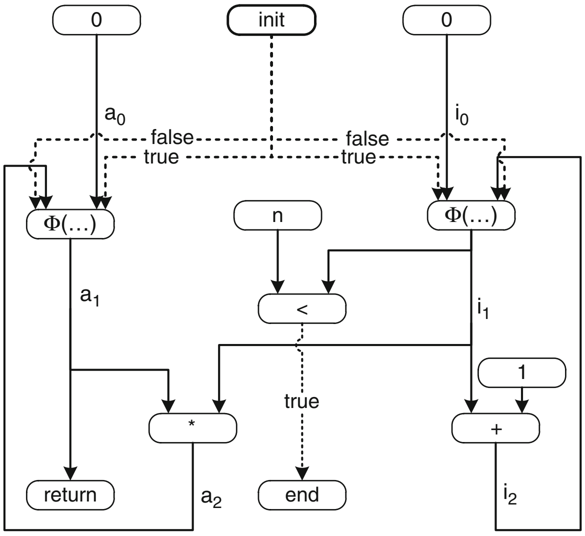


**Fig. 8** Control data flow graph (CDFG) of sample program

## *4.2 Operation Scheduling*

An acyclic CDFG could be mapped directly to a purely combinational datapath, evaluating the entire computation in a single clock cycle. However, this approach would lead to slow clock frequencies and not allow the execution of loops. Thus, the conventional solution is to distribute the computation over multiple clock cycles, leading to both faster clocks and allowing cycles (with inserted registers). In a simple implementation of this approach, hardware registers could be inserted after each operation. Note that further optimizations from high-level hardware synthesis might deviate from this scheme (e.g., packing multiple operators into a clock cycle by operator chaining [20]).

After realizing the computation in sequential logic, the question remains on how to control its execution (e.g., when to assert the registers' load inputs to accept newly computed values). This decision is called *scheduling* and can be performed both statically (at compile time) or dynamically (at execution time).

Dynamic scheduling does have numerous advantages: It can easily handle variable-latency operators, such as cached memory accesses, as the decision to store the read value is made only when the read port has indicated that the datum is available. Similarly, conditionals with differing computation times in their true and false branches can also consider the specific path taken at execution time to load the newly computed values at the correct time. Due to these advantages, dynamic scheduling has been used in a number of hardware compilers, such as COMRADE [5], CHiMPS [22], or CASH [2].

On the other hand, the additional logic required to make scheduling decisions at run-time potentially carries a large area overhead, especially when complex control flows have to implemented. In static scheduling, the times when to load newly computed values into registers and when to start new operations are determined at compile time. This is easy for fixed-latency operators, and the case of imbalanced conditional paths can be addressed by padding the shorter path with additional registers to the length of the longer path, equalizing the lengths. However, variable-latency operators pose a significant problem. In practice, they are assumed to execute in a fixed expected latency (e.g., single cycle on a cache hit). Dedicated logic detects at execution time when this assumption does not hold (e.g., on a cache miss), and halts (stalls) the entire datapath until the outstanding datum is actually available. Only then is execution allowed to proceed, giving the rest of the datapath the impression that variable-latency operators always provide their results within a fixed time. As an advantage, the control logic for orchestrating the execution of a statically scheduled hardware unit can be implemented in a compact and fast fashion (often just using multi-tapped shift registers). Hardware compilers using static scheduling include GarpCC [4], ROCCC [8], and the base microarchitecture in the Nymble flow.

## *4.3 Hardware Synthesis in Nymble*

With the fundamentals of the hardware synthesis now established, this section will consider some of the details of the Nymble compilation process in greater detail. Nymble actually partitions the SSA-CFG into a hierarchical CDFG, with each loop appearing as a single variable-latency node in the parent CDFG. In this manner, arbitrarily nested loop structures are supported. This is shown in Fig. 9: The top-level CDFG is the entire factorial function, which accepts a parameter $n$ from software. At this level, the loop has been encapsulated as a single operation. When

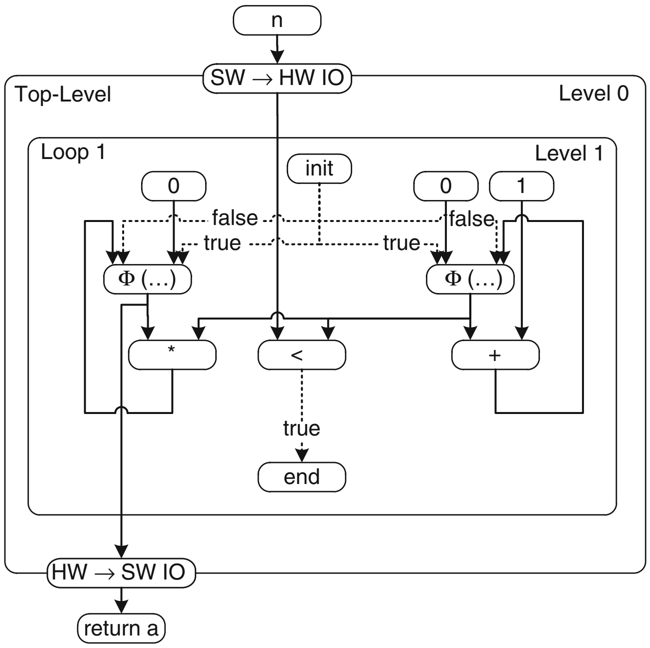


**Fig. 9** Hierarchically scheduled CDFG for sample program

it detects the loop termination condition, it signals the end of hardware execution to the hardware/software interface layer [15] and passes back the computed factorial from hardware to software.

Since we compile for the ACS target to a fully spatial hardware implementation with no operator reuse, we can employ a variant of the classical as-soon-as-possible (ASAP) static scheduling algorithm [20], adding just minor extensions to obey explicit constraints (discussed in Sect. 4.4).

Start times of operations are computed from the start times and expected latencies of their predecessor operations. Outer loops are stalled until nested inner loops explicitly signal their completion to the outer loop.

The hardware controller, sketched in Fig. 10, consists of a simple sequencer Reg 0–Reg 2 that just asserts the start signals (if required) of operators scheduled in the same cycle (called a stage in PreCoRe terminology) and loads the intermediate results of each operator into registers the expected latency number of cycles later. To support pipelining, the sequencer allows multiple stages to be active at the same time. This is limited by backward data dependencies in the DFG, though, which will lead to a longer initiation interval (II) between datapath starts. As a second function beyond the sequencing, a stall controller also detects violations of expected latency for variable-latency operators and stops the sequencing of all other operations until the variable-latency operator has actually completed. In the base

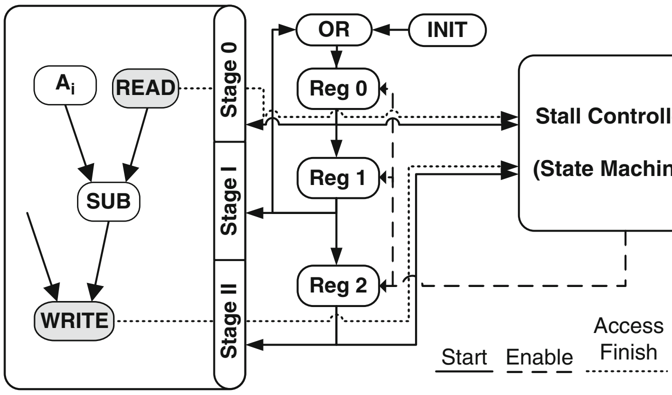


**Fig. 10** Synthesized controller for a non-speculative datapath

version of Nymble, this applies to nested loops (treated as single operators) and cached memory accesses. The latter will be handled differently with the PreCoRe mechanisms described in the next section.

## 4.4 *Compiling for the PreCoRe Microarchitecture*

PreCoRe requires the extension of the pure statically scheduled execution model of the base version of Nymble to a semi-statically scheduled version that makes more scheduling decisions at execution time, but far fewer than would be made in fully dynamic scheduling. In this section, we will discuss the changes required to the Nymble controller microarchitecture to integrate PreCoRe token handling (Sect. 3.2) and speculative queues (Sect. 3.3).

The stage-based nature of PreCoRe speculation has an impact on the static scheduling of multi-cycle operators in Nymble. In general, such multi-cycle operators will not support a partial replay, especially if they are obtained as third-party IP blocks (e.g., floating-point cores ), and will lack the required functionality (injection of preserved state data into the internals of the operator on a replay). Thus, all such operators are constrained in Nymble to be ASAP-scheduled either *completely* before or after any reads (which initiate replays on a misprediction).

Some parts of the controller actually are simplified by using PreCoRe. Since memory reads now become single-cycle operations due to value speculation, the stall controller on a read cache miss is no longer required. However, the need to support replays adds extra complexity. The microarchitecture of a controller supporting PreCoRe is sketched in Fig. 11; the key changes will be discussed next.

The simple sequencing registers in the original statically scheduled controller are replaced by so-called Flow Control nodes in the PreCoRe controller. During normal execution (no mispredicts), their behavior corresponds to those of the simple shift register controller–the Start signal is delayed by a single clock cycle and passed to

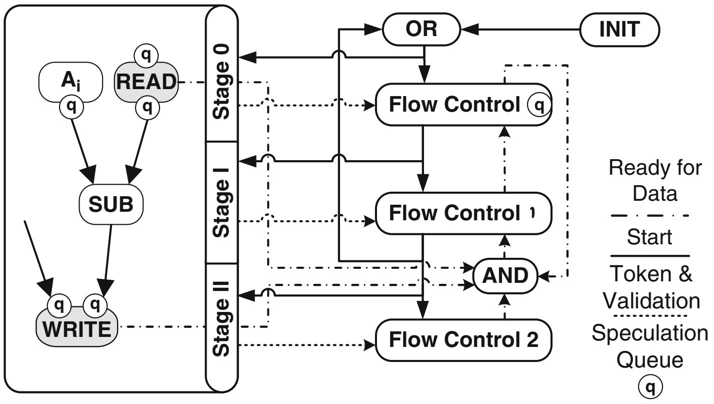


**Fig. 11** Synthesized controller for PreCoRe-speculative datapath

the subsequent stage. However, special logic is required to handle replays and to halt further computations in the operation pipeline as soon as a read is discovered to have mispredicted.

The easier of the extensions deals with the management of the input queues in read and write operators: Execution sequencing is only allowed to proceed if *all* input queues in the entire datapath have space (indicated by asserting their Ready for Data signal) for the operands that would be incoming in the next cycle. Lacking such space, sequencing at the datapath level is stopped, but all memory operators are allowed to proceed internally, draining their input queues. Once queue space has become available once more, datapath sequencing continues.

Flow control nodes of stages holding speculative operators (such as memory reads) have another extension over the simple sequencing registers: They have internal queues to buffer incoming start tokens. If their corresponding datapath stage requests a replay (a read discovered it mispredicted), the start tokens are reissued from the flow control token queue to restart the subsequent stages. If multiple mispredictions occur, the re-issue rate of the replayed start tokens is throttled to match the original Initiation Interval, thus keeping the static parts of the schedule valid. Only once a stage is confirmed in its entirety (precluding the need for a future replay) is the start token removed from the flow control token queue. Analogously to the capacity check for input queues in the datapath, execution in the controller is only allowed to proceed if all flow control nodes with queues have space available for incoming tokens. Otherwise, the controller is stopped, but the speculative reads continue to execute and will (at some point in time) output and confirm the correct data, removing a token from the flow control node responsible for their stage, and thus freeing up queue space. Please see [24] for further details.

Additional hardware (queues, token transition logic) will be inserted by Nymble into the statically scheduled controller only at the places required by the current application. This selective approach avoids the high overhead of relying on a general-purpose speculation support unit.

Now that we have discussed the PreCoRe microarchitecture and its automatic generation during hardware compilation, we can proceed to the last component of the solution, namely the multi-port memory system specialized to support speculative execution.

## 5 The MARC II Memory System

The multi-port cached memory system MARC II, initially presented in [16], has since been extended to support efficient operation of the PreCoRe mechanisms. PreCoRe relies on the memory subsystem to quickly satisfy the increased number of accesses due to execution replays. Note that MARC II deals strictly with nonspeculative data; all value speculation occurs in PreCoRe itself. Furthermore, even though PreCoRe gives the appearance of single-cycle memory reads (due to the value speculation), the scheme depends on low-latency replies from MARC II to quickly determine whether to commit a computation on confirmed values or replay it due to a discovered misprediction.

### *5.1 Overview*

The use of the spatially distributed computing paradigm on the adaptive computer also requires an appropriate parallel memory system. While some approaches rely purely on local on-chip memories (BlockRAMs), their limited size and lack of coherency protocols for shared accesses limit the scalability of the technique. Instead, we propose to use a shared memory system that gives the appearance of independent memory ports by providing each port with a distributed cache. Internal coherency mechanisms ensure a consistent view of all ports on the shared memory. Implementation-wise, we combine parallel on-chip BlockRAMs to realize fast caches, but still access the external off-chip main memory (shared with the SPP on the ACS) for bulk data.

The MARC line of memory systems has always aimed to provide multi-port operation supported by a dedicated cache infrastructure. In contrast, other ACS architectures often have at most a single port to external memory which is then explicitly allocated during scheduling to single memory operations. If they can actually serve multiple ports, they often have only very limited buffers (e.g., holding a DRAM row) as port-local storage. In contrast, MARC I [14] already gave multiple independent memory ports a coherent view of a shared multi-bank multi-port cache, allowing up to four parallel accesses. While the central shared cache avoided all coherency issues, it did not scale to larger numbers of ports and also limited the available clock frequency due to its fully associative organization.

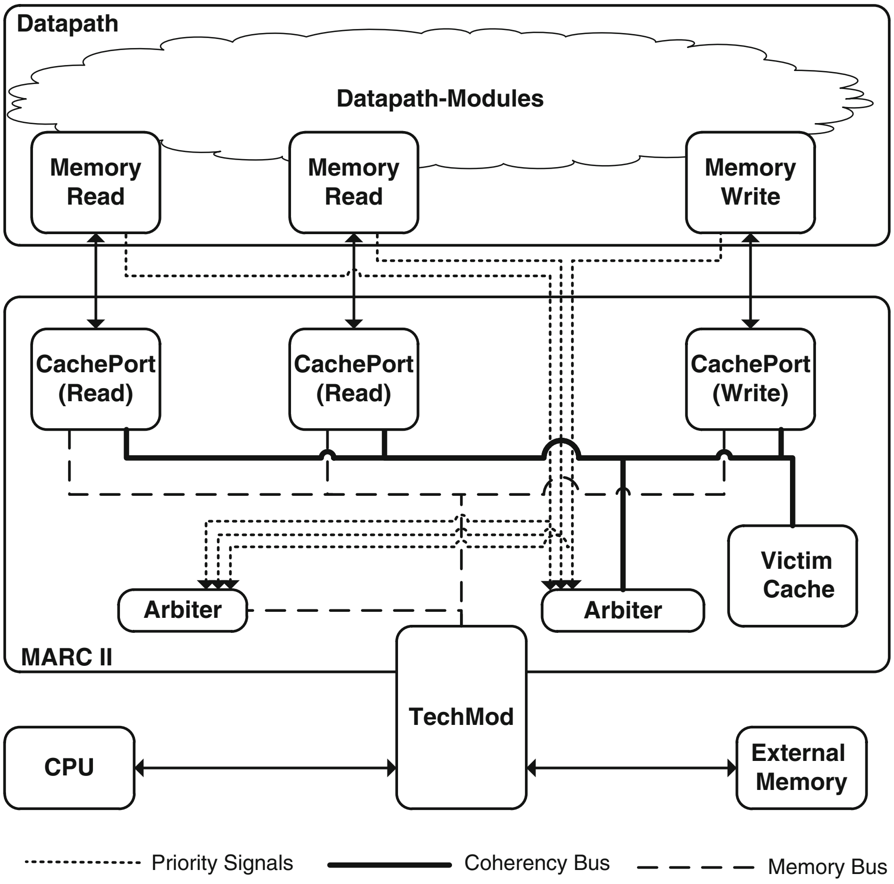


**Fig. 12** Overview of the MARC II cache system

To lift both restrictions, MARC II (shown in Fig. 12) instead relies on distributed per-port caches with a simpler but faster direct-mapped organization in on-chip BlockRAM. Since each MARC II per-port-cache is larger than the MARC I central cache, the lower cache hit rates due to the direct mapped organization do not lead to slow downs. Since all of the caches operate independently, a large number of memory accesses can be served in parallel. Interport coherency is managed explicitly by a dedicated coherency bus (CB, described in the next section). As MARC I, MARC II is designed to isolate the hardware-independent core of the system from the device-dependent memory controllers (QDR2-SSRAM, DDR2/3-SDRAM, etc.), which are implemented as so-called TechMods. This allows the easy retargeting of MARC II-based accelerators to different ACS platforms.

## 5.2 Cache System and Coherency Protocol

Ensuring coherency between distributed caches is a difficult problem that has been the subject of much research, leading to protocols such as MSI, MESI, and MOESI. However, by tailoring the MARC II coherency mechanisms to the requirements of PreCoRe, we can employ a much simpler, low-overhead solution.

PreCoRe relies on load value speculation and does not support speculative writes. Thus, a single Write port suffices in the memory system. All memory writes (being non-speculative) will have to be serialized through that port in program order (to avoid violating WAW dependencies). This limitation is less severe than it appears, since conventional programs execute 3x–6x as many loads as stores (measured in [10] for SPEC CPU 2006).

With the restriction to a single write port, we can employ a lightweight coherency protocol. Cache lines in a read port are either *valid* or *invalid*. In the write port, they are either *invalid* (the cache line is not present), *shared* (the cache line is present and also present in at least one other read port cache) or *exclusive* (the cache line is present and no other cache has it). Note that the explicit *modified* state, common to general-purpose coherency protocols, is not required here, since the write port cache *only* holds modified lines.

Figure 13 sketches how requests from the datapath are handled by MARC II caches on memory reads (a) and memory writes (b).

Whenever a read request is executed, it is checked first whether it is a cache hit. If so, data can be provided in a single cycle from the read port cache without having to interact with any other shared resource. Thus, cache hits can be served completely independent of the actions of other ports. If the requested data is not available from its cache (local cache miss), the request is forwarded to all other caches connected to the CB by broadcast. Only if the request cannot be served by any of the other caches (remote cache miss) must the external memory be accessed.

The behavior for writes is slightly more complex. Again, the port first determines whether the necessary cache line is present. If so, the new data is inserted into the cache. If the modified line was shared with other read ports, coherency must be ensured. This can happen in one of two user-selectable modes: In invalidate mode, the write port tells the read ports holding an affected shared cache line to invalidate it. If the read ports later require the cache line again, it will be requested over the CB from the write port (which now holds the only copy). In update mode, the write port immediately transmits its modified cache line over the CB to all read ports holding the shared old versions (here, multiple copies of the line exist). If the write is a local cache miss, the read port caches are accessed via the CB. On a remote hit, the line is then marked as shared in the write port. Only if no port cache holds the data is the external memory accessed.

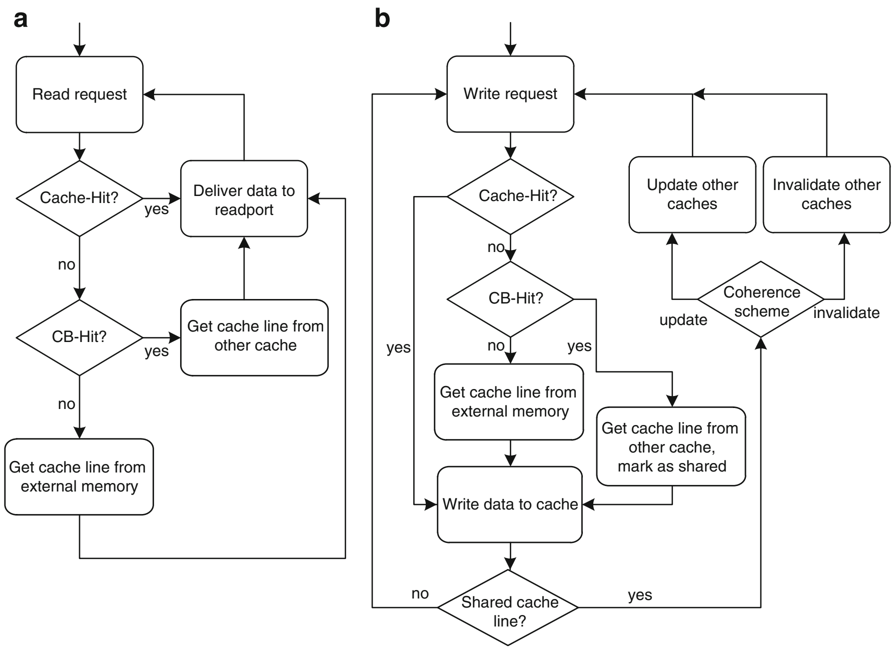


**Fig. 13** Processing an access in a read port (**a**) and a Write Port (**b**)

## *5.3 MARC II Support for PreCoRe Operations*

Obviously, the paradigm of spatially distributing computation can only be maintained in the MARC II front-end. The rest of the infrastructure consists of time-multiplexed shared resources (coherency bus, memory bus, TechMod, the actual external memory). The ports compete for access to these resources: In case of a local cache miss, the shared coherency bus must be accessed. On a remote cache miss, the request is forwarded to the shared external memory bus. If the external memory is in use (e.g., by the CPU), the access will have to wait until the memory becomes available for the RCU.

MARC II allows the accelerator to provide additional information on the priority of each access on a per-port basis: Each cache-port has its own priority-input, and an arbitration mechanism considers the given priorities of all pending requests when arbitrating the use of shared MARC II resources. This feature is used to apply the dynamic priority PreCoRe computes for each access (see Sect. 3.5) to influence the processing order of requests.

The displacement of cache lines in the distributed caches does not affect other caches, and thus is a less severe issue compared to cache line displacement in a single shared cache. However, the direct-mapped cache organization may cause frequent, undesirable cache displacements for some address sequences. In this case, the memory bus must be requested repeatedly to transfer the data from the external memory. Given the frequent memory accesses in PreCoRe, such displaced lines would lead to significantly longer replay times. By adding a small, fully associative victim cache, these drawbacks can be reduced. The impact of a victim cache on performance and where it should be placed (L2 or L1) has been studied in detail for conventional processors [11]. In context of MARC II, the victim cache can be integrated seamlessly by attaching it to the coherency bus, where it just acts as another remote cache. This avoids the need for yet another communication network and also keeps the access latency low by maintaining a single level of cache hierarchy.

MARC II also provides special support for the selective replay RAW dependency resolution mechanism introduced in Sect. 3.4. Each read port has a (relatively small) re-execution CAM (RXC, see Fig. 4b) that holds the last *n* read addresses, where *n* is the PreCoRe speculation depth. The write port broadcasts the write addresses over the coherency bus (see Sect. 5.2) to all read ports. If a RAW-speculated read was performed for an address overlapping a write address (as determined by a RXC lookup), a RAW violation is detected and signaled to the datapath in order initiate a replay.

## 6 Experimental Results

The ACS infrastructure proposed in this work has been implemented on the Xilinx ML507 development board using the Verilog hardware description language. Its core is a Virtex-5 FX FPGA, which is connected to various peripheral components, with a DDR2-SDRAM bank acting as external main memory. The reconfigurable fabric on the FPGA is used as RCU and the embedded PowerPC 440 as SPP. All benchmarks were compiled from C using the Nymble C-to-hardware compiler. The resulting RTL description was then synthesized using Synopsys Synplify Premier DP 9.6.2 and placed and routed with Xilinx ISE 11.1.

For evaluating the different components of the system, we used selected application benchmarks from well-known benchmark suites (e.g., MediaBench [17], Honeywell ACS Suite [13]). The samples include the gf_multiply kernel from the Pegwit elliptic curve cryptography application, the quantization and wavelet transformation of the Versatility image compression application, and a luminance median filter. While the application benchmarks provide a good overview on the performance of the overall system, we also used synthetic hardware kernels to test specific features and characteristics of the system. The following paragraphs just summarize the actual results, please see [24–26] for the detailed measurements.

Each kernel was compiled twice, once with PreCoRe enabled and once with the original purely statically scheduled datapath. As previously discussed in Sect. 4.2, in the static version, a single cache miss stalls the entire datapath. Thus, all differences in performance are due to making better use of the hardware operators that are already present in the datapath.

Depending on the regularity of the input data, performance gains of up to 23 % have been observed by employing load value speculation alone. Although successful speculation effectively hides the memory access latency of its particular access, even unsuccessful speculation may result in an improved execution time: The latency of later accesses may potentially be hidden by allowing them to execute earlier (instead of being stalled with the rest of the datapath). If the access executed early used a non-speculative read address, the data will be prefetched into the cache for use not only by the specific read port executing the early access, but also by all other ports that can retrieve it using the coherency bus (instead of accessing main memory again).

The dynamic priority computation discussed in Sect. 3.5 can lead to speed-ups 2–25.5 %. However, the best choice of weights for the computation is highly application dependent. Compiler support for selecting appropriate parameters automatically would be highly desirable here.

Despite being only a secondary effect of the actual read value speculation, the impact of prefetching should not be underestimated. As an experiment, we disabled the value predictor, forcing it to always mispredict. Even in this crippled form, PreCoRe still executes reads as single-cycle operations and avoids datapath-wide stalls, thus allowing prefetching to be performed. This prefetching-only version of PreCoRe yields a speed-up of 1.43x. Re-enabling the value predictor reduces the execution time further to a total speed-up of 1.58x over the original statically scheduled version. For this specific benchmark, the prefetching made possible by the non-stalling mispredicting reads, and not the successful speculation, is actually responsible for most of the performance gain.

These benchmarks were constructed so that no RAW dependencies existed between accesses. If such dependencies cannot be ruled out (e.g., by using the C `restrict` keyword), the dynamic resolution mechanisms described in Sect. 3.4 need to be employed. For a synthetic benchmark that has a third of all speculative accesses violating RAW dependencies, the selective resolution method (detecting overlapping addresses) requires up to 4 % fewer clock cycles than the universal resolution (that assumes all executed writes interfere with all reads). Adding a victim cache (Sect. 5.3) to speed up replays further gains up to another 9 % of clock cycles.

Combining the various features of PreCoRe, it was possible to achieve wall-clock improvements of up to 2.59x in our examples, *without* incurring any slow downs. This is a significant improvement over prior work such as [21] discussed in Sect. 1.

However, enabling PreCoRe has both an area and a clock frequency cost. The latter is not relevant for our experiments, since the maximum clock slowdown we observed (11 % over the non-speculative versions) was either more than compensated by the PreCoRe speed-ups, or lead to a clock frequency that still exceeded the 100 MHz limit of the ML507 reference design. Since most of the critical path lies inside of the MARC II memory system, the achievable maximum clock frequency is almost independent of whether a speculative or nonspeculative execution model is chosen.

In contrast to the negligible clock slowdown, PreCoRe carries a significant area overhead (in our benchmarks: 1.45x–3.22x, counting slices). Much of this is due to the current Nymble hardware back-end not exploiting the sharing of queues across multiple operators in a stage and the pipeline balancing registers automatically inserted by the compiler not being recognized as mappable to FPGA shift-register primitives by the logic synthesis tool. Both of these issues could be addressed by adding the appropriate low-level optimization passes to Nymble.

## 7 Conclusion

We have presented a comprehensive approach to widening the memory bottleneck that is also starting to affect reconfigurable computing. It encompasses the microarchitectural mechanisms of the PreCoRe value speculation framework, the automatic generation of application-specific controllers implementing these techniques from C programs by the Nymble hardware compiler, and the run-time support for parallel memory accesses and quick execution replays provided by the MARC II memory system.

Our approach embraces the paradigm of spatially distributed computation, preferring to expend reconfigurable silicon area on application-specific computation support structures such as PreCoRe, instead of on general-purpose support mechanisms with diminishing efficiency, such as classical caches. With the ongoing trend towards ever larger reconfigurable devices, continued research in this area seems very promising.

**Acknowledgements** This work was supported by the German national research foundation DFG and by Xilinx Inc.

## References

1. Aho AV, Lam MS et al (2006) Compilers: principles, techniques, and tools, 2nd edn. Prentice Hall, New Jersey
2. Budiu M, Goldstein SC (2003) Optimizing memory accesses for spatial computation. In: Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization, CGO '03, IEEE Computer Society, Silver Spring, MD, pp 216–227
3. Burtscher M, Zorn BG et al (2002) Hybrid load-value predictors. IEEE Trans Comput 51:759–774
4. Callahan TJ, Hauser JR et al (2000) The Garp architecture and C compiler. IEEE Comput 33(4):62–69
5. Gädke-Lütjens H (2011) Dynamic scheduling in high-level compilation for adaptive computers. Ph.D. thesis, Technical University Braunschweig
6. González J, González A (1999) Limits of instruction level parallelism with data value speculation. In: International conference on vector and parallel processing. VECPAR '98, Springer, London, UK, pp 452–465
7. Scale Compiler Group (2006) Scale. A scalable compiler for analytical experiments. Department of Computer Science University of Massachusetts, http://www.cs.utexas.edu/users/cart/Scale/
8. Guo Z, Najjar W et al (2008) Efficient hardware code generation for FPGAs. ACM Trans. on Architecture and Code Optimization (TACO) 5(1):1–26

9. Hennessy JL, Patterson DA (2003) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann Publishers, San Francisco, CA, USA
10. Isen C, John LK et al (2009) A tale of two processors: revisiting the RISC-CISC debate. In: Proceedings of SPEC Benchmark Workshop, pp 57–76
11. Jouppi NP (1990) Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: Proceedings of the 17th annual international symposium on computer architecture, ISCA ’90, ACM, New York, NY, USA, pp 364–373
12. Kaeli D, Yew P-C (2005) Speculative execution in high performance computer architectures. CRC Press, Boca Raton, FL
13. Kumar S, Pires L et al (2000) A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In: FPGA, ACM, New York, NY, USA, pp 126–134
14. Lange H, Koch A (2007) An execution model for hardware/software compilation and its system-level realization. In: International conference on field programmable logic and applications (FPL), 2007, pp 285–292
15. Lange H, Koch A (2010) Architectures and execution models for hardware/software compilation and their system-level realization. IEEE Trans Comput 59(10):1363–1377
16. Lange H, Wink T et al (2011) MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers. In: 2011 Conference on design, automation & test in Europe (DATE)
17. Lee C, Potkonjak M et al (1997) MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of 30th annual IEEE/ACM international symposium on microarchitecture, 1997, pp 330–335
18. Lipasti MH, Wilkerson CB et al (1996) Value locality and load value prediction. ACM, New York, NY, USA, 31(9):138–147
19. McNairy C, Soltis D (2003) Itanium 2 processor microarchitecture. IEEE Micro 23:44–55
20. Micheli GD (1994) Synthesis and optimization of digital circuits, 1st edn. McGraw-Hill Higher Education, New York, USA
21. Mock M, Villamarin R et al (2005) An empirical study of data speculation use on the intel itanium 2 processor. In: Proceedings of workshop on interaction between compilers and computer architectures, IEEE Computer Society, Washington, DC, USA, pp 22–33
22. Putnam A, Bennett D et al (2008) CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In: 2008 international conference on field programmable logic and applications (FPL), pp 173–178
23. Sazeides Y, Smith JE (1997) The predictability of data values. In: Proceedings of international symposium on microarchitecture, MICRO 30. IEEE Computer Society, Washington, DC, USA, pp 248–258
24. Thielmann B, Huthmann J et al (2011) Evaluation of speculative execution techniques for high-level language to hardware compilation. In: 6th international workshop on reconfigurable communication-centric systems-on-chip (ReCoSoC) 2011, pp 1–8
25. Thielmann B, Huthmann J et al (2011) Precore—a token-based speculation architecture for high-level language to hardware compilation. In: 2011 international conference on field programmable logic and applications (FPL), pp 123–129
26. Thielmann B, Wink T et al (2011) RAP: More efficient memory access in highly speculative execution on reconfigurable adaptive computers. In: 2011 international conference on reconfigurable computing and FPGAs (ReConFig)
27. Wang K, Franklin M (1997) Highly accurate data value prediction using hybrid predictors. In: Proceedings 30th annual IEEE/ACM international symposium on microarchitecture, 1997, pp 281–290
28. Weaver G, Cahoon B et al (1997) Common language encoding form (clef) design document. Technical report, Department of Computer Science, University of Massachusetts
29. Yeh T-Y, Patt YN (1992) Alternative implementations of two-level adaptive branch prediction. In: Proceedings of the 19th annual international symposium on computer architecture, pp 124–134

# Decimal Division Using the Newton–Raphson Method and Radix-1000 Arithmetic

**Mário P. Véstias and Horácio C. Neto**

## 1 Introduction

Computer arithmetic is predominantly performed using binary arithmetic because the hardware implementations of the operations are simpler than those for decimal computation. However, many decimal fractions cannot be represented exactly as binary fractions with a finite number of bits. The value 0.1, for example, can only be represented as an infinitely recurring binary number. If a binary approximation is used instead of the exact decimal fraction, the results will not be exact even if the arithmetic is exact. Therefore, many applications, such as financial and commercial, where the results must be exact, matching those obtained by human calculations, must be performed using decimal arithmetic. Until very recently, the adopted solution was to implement decimal operations using software algorithms based on binary arithmetic. However, these software solutions are typically three or four orders of magnitude slower than binary arithmetic implemented in hardware [4]. To speed up the execution of decimal arithmetic, a few processors, such as the IBM Power6 [1], already include dedicated hardware for decimal floating-point operations.

Decimal division is one of the fundamental operations for hardware-based decimal arithmetic. Division techniques based on digit-recurrence algorithms are the most used in (binary) hardware dividers, and have also been considered in most decimal division proposals. Nikmehr et al. [14] proposed a decimal floating-point division algorithm based on high-radix SRT division. Lang and Nannarelli [10] have also implemented a decimal division unit based on the digit-recurrence

M.P. Véstias (✉)
INESC-ID/ISEL/IPL, Lisbon, Portugal
e-mail: mvestias@deetc.isel.ipl.pt

H.C. Neto
INESC-ID/IST/UTL, Lisbon, Portugal
e-mail: hcn@inesc-id.pt

algorithm. Their work was compared to a division unit based on the Newton–Raphson (NR) iterative method concluding that the implemented digit-recurrence seemed advantageous in terms of latency compared to the NR method. Vázquez et al. [19] proposed an algorithm based on the SRT digit-recurrence method and the corresponding architecture for decimal division. Their implementation has a comparable delay to that from [10] using lower hardware complexity. Wang et al. [23] proposed an arithmetic algorithm and hardware design for decimal floating-point division using an initial piecewise linear Taylor series approximation followed by modified Newton–Raphson iterations.

Digit-recurrence algorithms have been the primary choice for decimal division mainly because of the complexity of the decimal multipliers required to implement the NR iterations. However, digit recurrence algorithms only produce one decimal digit at each iteration, while the NR method ensures quadratic convergence. Also, dividers based on the NR algorithm can take advantage of existing decimal multiplier hardware.

The performance and the area of a divider based on the NR method depend mainly on the efficiency of the decimal multipliers. Two main lines have been considered in the design of the multipliers: iterative and parallel. In the iterative approach, the multiplicand is iteratively multiplied by one digit of the multiplier to generate a partial product. Partial products are then added to produce the final decimal result. A few decimal multipliers have been proposed based on iterative units, such as [7, 8, 18]. Parallel decimal multipliers have also been recently proposed in [5, 9, 20, 21]. While parallel decimal multipliers are faster, they require much more hardware resources than iterative ones.

In this work, a new iterative decimal hardware divider is proposed. The division algorithm used is based on the calculation of the reciprocal of the divisor using the Newton–Raphson method, followed by a final multiplication by the dividend to obtain the quotient. The NR algorithm is implemented with two major optimizations, a new initial approximation and a new iterative decimal multiplier. Instead of using the Taylor series expansion, as in [23], the initial approximation of the reciprocal is calculated using piecewise minimax polynomials. This allows to reduce the number of NR iterations and therefore the size and the latency of the multipliers and consequently of the divider. All multiplications are done using very efficient iterative radix-1000 arithmetic. The partial multiplications of the radix-1000 multipliers are directly implemented using the available binary multipliers, therefore significantly improving the overall performance. Further, the use of an internal radix-1000 representation significantly simplifies the binary to/from decimal conversions.

This chapter is organized as follows. Section 2 describes the algorithm used to compute the reciprocal of a decimal number and the error analysis of the method. Section 3 describes the design of the decimal divider. Section 4 provides results with and without embedded multipliers and presents a comparison with a state-of-the-art iterative divider.

## 2 Reciprocal Computation

The reciprocal of the divisor, $\frac{1}{x}$, is calculated using the Newton–Raphson iterative method. The first iteration uses an initial seed, herein obtained using a piecewise linear approximation based on minimax polynomials. The method converges quadratically, that is, the error of the approximation decreases quadratically with the number of iterations.

### 2.1 *Initial Polynomial Approximation*

The minimax polynomial is the approximating polynomial which has the smallest maximum error from the given function. According to the Chebyshev alternation theorem [17], the minimax degree-1 approximation to a given function in a specific interval is the 1st order polynomial that has maximum error at the interval extremes and, with alternate sign, in one interior point.

The 1st order minimax polynomial

$$p^{(1)}(x) = b + m(x - X0) \tag{1}$$

that approximates the function $\frac{1}{x}$ in the interval $[X0, X1]$ can be obtained in direct form [12], and its coefficients are

$$b = \frac{1}{2\,X0} - \frac{1}{2\,X1} + \frac{1}{\sqrt{X0\,X1}} \tag{2}$$

$$m = -\frac{1}{X0\,X1} \tag{3}$$

and its maximum error is

$$|E_{p1}| = \frac{1}{2\,X0} + \frac{1}{2\,X1} - \frac{1}{\sqrt{X0\,X1}}.x \tag{4}$$

### 2.2 *Newton–Raphson Iterations*

Given the divisor $x$, the Newton–Raphson calculates its reciprocal $\frac{1}{x}$ by finding the zero of the equation $f(y) = \frac{1}{y} - x$ using an iterative process. The NR iterations to obtain $\frac{1}{x}$ are given by

$$y^{(i+1)} = y^{(i)}\,(2 - y^{(i)}\,x), \tag{5}$$

**Table 1** Maximum approximation errors for different interval sizes

| Interval size | $\max(E_{p1})$ | $\max(E_{\mathrm{NR}_1})$ | $\max(E_{\mathrm{NR}_2})$ |
|---|---|---|---|
| $10^{-2}$ | $0.11 \times 10^{-1}$ | $0.13 \times 10^{-4}$ | $0.19 \times 10^{-10}$ |
| $10^{-3}$ | $0.13 \times 10^{-3}$ | $0.16 \times 10^{-8}$ | $0.24 \times 10^{-18}$ |
| $10^{-4}$ | $0.13 \times 10^{-5}$ | $0.16 \times 10^{-12}$ | $0.25 \times 10^{-26}$ |

where the initial value $y^{(0)}$ is the initial approximation to $\frac{1}{x}$, herein computed by (1). If the error $E_{\mathrm{NR}}^{(i)}$ (theoretical error without truncation) at iteration $i$ is given by

$$E_{\mathrm{NR}}^{(i)} = \frac{1}{x} - y^{(i)}$$

then the error $E_{\mathrm{NR}}^{(i)}$ at iteration $i+1$ is given by

$$E_{\mathrm{NR}}^{(i+1)} = x\,(E_{\mathrm{NR}}^{(i)})^2 \tag{6}$$

[15].

As, for each piecewise approximation, $x$ is a number in the interval $[X0, X1[$ then $x < X1$, and the NR iteration errors (in this case, the first two iterations) are upper-bounded by

$$E_{\mathrm{NR}}^{(1)} < X1\ (E_{p1})^2 \tag{7}$$

$$E_{\mathrm{NR}}^{(2)} < X1^2\ (E_{p1})^4. \tag{8}$$

Table 1 shows the upper bounds for the reciprocal approximation errors considering that the normalized divisor interval $[0.1, 1[$ is divided into subintervals of size $10^{-1}$.

As shown, the use of subintervals of size $10^{-3}$ is sufficient to provide an error lower than $10^{-8}$ after one NR iteration, and an error lower than $10^{-18}$ after two NR iterations.

## 2.3 Truncation Errors

In practice and to reduce the size of the lookup table and of the multipliers, the arithmetic operators are implemented with less than full precision. In this case (1) becomes

$$p^{(1)}(x) = b + \varepsilon_{T0} + (m + \varepsilon_{T1})((x - X0) + \varepsilon_{T2}), \tag{9}$$

where $\varepsilon_{T0}$, $\varepsilon_{T1}$, and $\varepsilon_{T2}$ are the truncation errors of $b$, $m$, and $(x - X0)$, respectively.

Given the input subinterval $[X0, X1[$, the maximum truncation error of the first-order polynomial approximation becomes therefore upper bounded by

**Table 2** Number of fractional digits used for each operand

| 1st order polynomial | $b=5$ | $m=2$ | $(x-X0)=7$ |
|---|---|---|---|
| 1st NR iteration | $y^{(0)}=5$ | $x=11$ | $y^{(0)}x=10$ |
| 2nd NR iteration | $y^{(1)}=9$ | $x=16$ | $y^{(1)}x=19$ |

**Table 3** Upper-bounds for maximum reciprocal errors

| Divisor digits | NR iterations | Error upper-bound |
|---|---|---|
| 8 | 1 | $0.34\times10^{-8}$ |
| 16 | 2 | $0.42\times10^{-17}$ |

$$\varepsilon_T < \varepsilon_{T0} + (X1-X0)\,\varepsilon_{T1} + \frac{\varepsilon_{T2}}{X1X0} + \varepsilon_{T1}\varepsilon_{T2}. \tag{10}$$

If the operand size is reduced in the computation of the NR iterations (5), the iteration error $E_{\mathrm{NR}}$ becomes

$$E_{\mathrm{NR}}^{(i+1)} = x\,(E_{\mathrm{NR}}^{(i)} + \varepsilon_{TY})^2 + y^{(i)}\,\varepsilon_{TP} - (y^{(i)})^2\,\varepsilon_{TX}, \tag{11}$$

where $\varepsilon_{TY}$, $\varepsilon_{TX}$, and $\varepsilon_{TP}$ are the truncation errors of the operands $y^{(i)}$, $x$, and of the product $y^{(i)}\,x$, respectively.

The iteration error can therefore be upper bounded by

$$E_{\mathrm{NR}}^{(i+1)} \le \max\left(X1\,(E_{\mathrm{NR}}^{(i)} + \varepsilon_{TY})^2 + \frac{1}{X0}\varepsilon_{TP}, \frac{1}{X0^2}\varepsilon_{TY}\right). \tag{12}$$

Tables 2 and 3 show the number of fractional digits used for each operand to ensure faithful rounding after the final reciprocal computation. The resulting precision for the reciprocal, according to the error upper bounds provided by (12), is also indicated. The reciprocal result will be a number in the interval $[1,10]$ and, therefore, will have seven fractional digits, in the case of the 8-digit result, and 15 fractional digits, in the case of the 16-digit result. Faithful rounding guarantees that the computed result is one of the two floating-point neighbors of the exact result [11, 16].

## 3 Implementation of the Divider

Two dividers have been implemented, one divider for 8-digit operands and another for 16-digit operands. In both cases, the quotient $q=\frac{z}{x}$ is obtained by calculating the reciprocal $\frac{1}{x}$ using the method described in Sect. 2, and then multiplying it by $z$.

The piecewise first-order minimax polynomial approximation of the reciprocal, $y^{(0)}$, calculated according to (1), as

$$y^{(0)} = p^{(1)}(x) = b + m\times(x-X0)$$

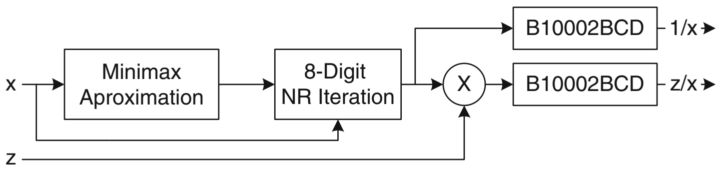


**Fig. 1** Iterative 8-digit decimal divider

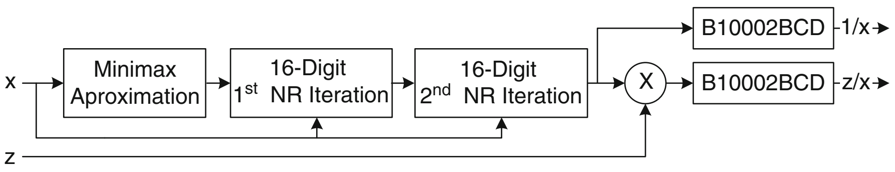


**Fig. 2** Iterative 16-digit decimal divider

is the starting point of the iterative NR method for both the 8-and the 16-digit dividers. The coefficients, $b$ and $m$, of the approximation polynomial are stored in a ROM and then used to calculate the initial linear approximation.

The calculation of the reciprocal requires a single iteration of the NR method, for the 8-digit division, and two iterations for the 16-digit division. This guarantees the required precision as indicated by the error upper bounds shown in Table 3.

The block diagrams for the 8-and 16-digit dividers using an initial minimax approximation and the NR method are illustrated in Figs. 1 and 2.

The outputs of both dividers are the reciprocal of the divisor and the quotient. The outputs of the NR iterations and of the last multiplier used to calculate $\frac{1}{x} \times z$ are numbers represented in radix-1000. Therefore, a final radix-1000 to decimal conversion is used to obtain the reciprocal and the final division result in *binary-coded decimal* (BCD).

A radix-1000 number, $r$, is represented with radix-1000 digits, $r = r_n r_{n-1} \ldots r_1 r_0$, where each $r_i$ digit is a decimal number from 0 to 999. Therefore a radix-1000 number has the following decimal value:

$$r = r_n \times 10^{3\times n} + r_{n-1} \times 10^{3\times(n-1)} + \ldots + r_1 \times 10^{3\times 1} + r_0.$$

This radix has some important characteristics. A radix-1000 number can be easily converted from/to a BCD number and radix-1000 arithmetic can be efficiently implemented using binary arithmetic. Also, radix-1000 (base $10^3$) is close to $2^{10}$, an important characteristic whenever binary from/to radix-1000 conversion has to be done, as will be explained in the following sections.

In the following sections, all blocks are described in detail.

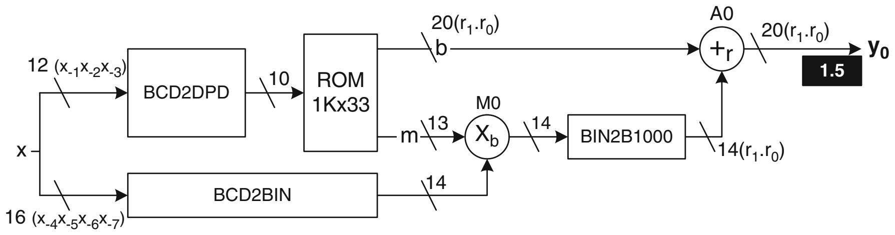


**Fig. 3** Block diagram of minimax approximation unit

## *3.1 Minimax Approximation*

To determine the initial approximation of the reciprocal, approximation intervals of size $10^{-3}$ have been chosen (Table 1). For each interval, the pair of coefficients ($b$, $m$) of the minimax polynomial is stored in a ROM. So, given a $x \in [0,1[$ in the form $0.x_{-1}x_{-2}x_{-3}x_{-4}x_{-5}x_{-6}x_{-7}x_{-8}$, whose reciprocal is to be calculated, its three most significant digits, $x_{-1}x_{-2}x_{-3}$, are used to address the ROM to retrieve the coefficients $b$ and $m$ to be used in the polynomial calculation as follows (see Fig. 3):

$$y^{(0)} = p^{(1)}(x) = b + m \times (x - X0),$$

where $X0$ is $x$ truncated to three fractional digits ($0.x_{-1}x_{-2}x_{-3}$).

A direct implementation of the method would require a ROM with size $2^{3\times4} \times k$, where $k$ is the number of bits needed to represent the pair of coefficients ($b$, $m$). With six digits for $b$ and four digits for $m$, the ROM would have a size of $2^{12} \times 40$. To reduce the size of the ROM, the three input digits of X0 are first converted to a 10-bit *densely packed decimal* (DPD) representation [3], to be detailed in the following section.

The operands of the multiplication in the minimax approximation are as follows

$$m \times (x - X0) = m \times (x - 0.x_{-1}x_{-2}x_{-3}) = m \times (0.000x_{-4}x_{-5}x_{-6}x_{-7}x_{-8}).$$

One of the multiplier operands, the $m$ coefficient, is stored in the ROM with four digits (two digits for the integer part and two digits for the fractional part). The other operand, $(x - X0)$, is truncated to seven fractional digits, $0.000x_{-4}x_{-5}x_{-6}x_{-7}$, enough to guarantee the required precision of the initial approximation (see Table 2).

So, the product will have the format $0.0u_{-2}u_{-3}u_{-4}u_{-5}u_{-6}u_{-7}u_{-8}u_{-9}$, which is then truncated to $0.0u_{-2}u_{-3}u_{-4}u_{-5}$ (the least four significant digits are ignored).

To implement this multiplication, we ignore the fractional points and multiply the decimal digits with an integer decimal multiplier. In this case, we need a $4 \times 4$ multiplier whose result is truncated to four digits. The final four digits are the digits $u_{-2}u_{-3}u_{-4}u_{-5}$ of the minimax multiplication.

The proposed design uses a binary multiplier to implement this multiplication. Therefore, the four digits of $x$ used in the multiplication, $x_{-4}, x_{-5}, x_{-6}, x_{-7}$, must be converted to binary (BCD2BIN component) while the $m$ operand is retrieved from the ROM already in binary format.

The result of the multiplication is then added to the coefficient $b$, which is represented with one digit for the integer part and five digits for the fractional part. A radix-1000 adder is used, since the NR iterations use radix-1000 multipliers. Therefore, the $b$ coefficient is stored in the ROM in radix-1000 format. However, the output from the binary multiplier is a binary number, and therefore, it must be converted to radix-1000 and truncated as stated above.

To optimize the binary to radix-1000 conversion of the output of the multiplier, followed by the truncation of the least four significant digits, the four digits of the $m$ operand are first multiplied by a constant. The constant is determined as explained next. Given two decimal operands, op1 and op2, with four digits each, the four digits truncation of $op1 \times op2$ is given by

$$\frac{op1 \times op2}{10^4}$$

which can be rewritten as

$$op1 \times \frac{2^{13}}{10^4} \times \frac{op2}{2^{13}} = mc \times \frac{op2}{2^{13}}, \tag{13}$$

where $mc = op1 \times \frac{2^{13}}{10^4}$.

While the original coefficient $m$ (op1) is representable with 14 bits, the coefficient $mc$ needs only 13 bits. So, $mc$ is stored instead of $m$. Besides, the 27-bits at the multiplier output are simply shifted 13 bits to do the four digits truncation. The remaining 14 bits from the multiplier output are then converted to radix-1000 (*BIN2B1000* component) and added to $b$.

Considering these optimizations, the size of the ROM is reduced to $2^{10} \times 33$.

### 3.1.1 BCD to DPD Conversion

The *DPD* encoding [3] is a specific format to compress three decimal digits into 10 bits, instead of the 12 bits required using simple BCD (one digit in 4 bits). The DPD encoding has been developed such that the compression can be implemented using only very simple boolean operations.

The DPD converter transforms a group of three BCD digits ($D_2D_1D_0$, where $D_2 = abcd$, $D_1 = efgh$ and $D_0 = ijkm$) into 10-bit numbers of the form "$pqr\ stu\ v\ wx\ y$" according to logic equations (14) (as proposed in [3]). All the required logic functions have five or less variables, and therefore each function can be implemented using at most a single 6-input LUT (e.g., in Virtex6 FPGA) or two 4-input LUTs and a multiplexer (e.g., in Virtex4 FPGA).

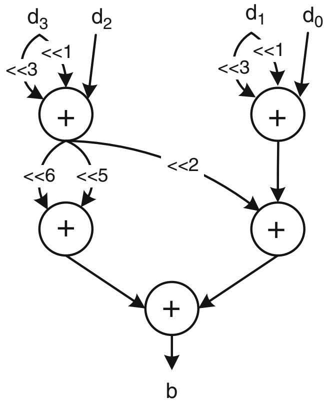


**Fig. 4** Block diagram of the decimal to binary converter

$$\begin{aligned}
p &= b + a \cdot j + a \cdot f \cdot i \\
q &= c + a \cdot k + a \cdot g \cdot i \\
r &= d \\
s &= f \cdot \overline{a} + f \cdot \overline{i} + \overline{a} \cdot e \cdot j + e \cdot i \\
t &= g \cdot \overline{a} + g \cdot \overline{i} + \overline{a} \cdot e \cdot k + a \cdot i \\
u &= h \\
v &= \overline{a} \cdot \overline{e} \cdot \overline{i} \\
w &= a + e \cdot i + j \cdot \overline{e} \\
x &= e + a \cdot i + k \cdot \overline{a} \\
y &= m. \qquad (14)
\end{aligned}$$

### 3.1.2 BCD to Binary Converter: BCD2BIN

The conversion of a 4-digit decimal, $d = d_3d_2d_1d_0$, to binary, $b$, is done using binary arithmetic according to

$$b = (d_3 10 + d_2)10^2 + d_1 10 + d_0 \qquad (15)$$

which is implemented with only adders and shifts by factoring the constants as powers of two, as

$$b = (d_3 \times 8 + d_3 \times 2 + d_2)(64 + 32 + 4) + d_1 \times 8 + d_1 \times 2 + d_0. \qquad (16)$$

The final implementation requires five adders, as shown in Fig. 4 (in this figure, $<<sh$ represents a left shift of $sh$ bits).

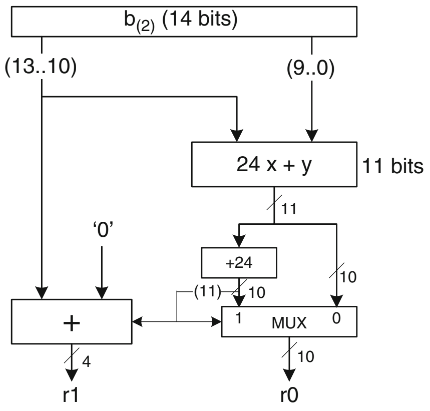


**Fig. 5** 14-bit binary to radix-1000 converter

### 3.1.3 Binary to Radix-1000 Converter

The binary to radix-1000 converter (BIN2B1000 block in Fig. 3) is based on the architecture proposed in [13] to convert a 20-bit binary number to a two-digit radix-1000 number. The BIN2B1000 converter used in the minimax polynomial calculation only needs to convert a 14-bit binary number (see Fig. 5).

The circuit converts a binary number $b \in [0, 9999]$ to one decimal digit plus a digit base-1000 number $r$, that is

$$b = r_1 \cdot 10^3 + r_0 = r.$$

Considering that

$$b = b_1 \cdot 2^{10} + b_0$$

it follows that

$$\begin{aligned} b &= b_1 \cdot 1024 + b_0 & b_1 \leq 10 = \tfrac{9999}{1024} \\ &= b_1 \cdot 1000 + \underbrace{b_1 \cdot 24 + b_0}_{c} & b_0 \leq 1023, \end{aligned} \tag{17}$$

where

$$c = b_1 \cdot 24 + b_0 \qquad c \leq 1215 \leftarrow 11\,\text{bits}. \tag{18}$$

From Eqs. (17) and (18), $b$ is given by

$$b = (b_1) \cdot 1000 + c \tag{19}$$

and a first approximation for the two digits is:

$$\hat{r}_1 = b_1 \leq 9 \qquad \leftarrow 4\,\text{bits} \tag{20}$$

$$\hat{r}_0 = c \leq 1215 \qquad \leftarrow 11\,\text{bits}. \tag{21}$$

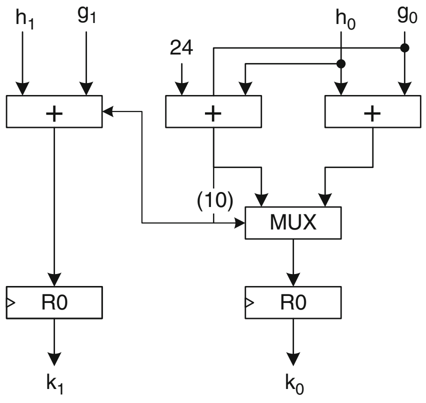


**Fig. 6** Adder radix-1000 for numbers with two radix-1000 digits

From these, the final step is to test if $\hat{r}_0$ is higher than 999, that is, if $\hat{r}_0 + 24$ is higher then 1023. If yes, $\hat{r}_1$ must be incremented by one and $\hat{r}_0$ must be increased by 24 to adjust the result. Otherwise, the result is already correct.

### 3.1.4 Adder Radix-1000

A radix-1000 adder sums two radix-1000 numbers, $g = g_2 g_1 g_0$ and $h = h_2 h_1 h_0$. The result, $k = k_2 k_1 k_0 = g + h$, is calculated as follows:

$$\begin{aligned} k_0^{'} &= g_0 + h_0 \\ k_1^{'} &= g_1 + h_1 + Cy_0 \\ k_2^{'} &= g_2 + h_2 + Cy_1 \\ k_m &= k_m^{'},\ \ Cy_m = 0,\ \text{ if } k_m^{'} < 1000 \\ &= k_m^{'} + 24,\ \ Cy_m = 1,\ \text{ otherwise.} \end{aligned} \tag{22}$$

The radix-1000 adder in the circuit for minimax polynomial calculation adds two operands with two radix-1000 digits each. In this particular case, the result can also be represented with only two radix-1000 digits. Therefore, the more general case given in Eq. (22) simplifies into Eq. (23).

$$\begin{aligned} k_0^{'} &= g_0 + h_0 \\ k_1^{'} &= g_1 + h_1 + Cy_0 \\ k_0 &= k_0^{'},\ \ Cy_0 = 0,\ \text{ if } k_0^{'} < 1000 \\ &= k_0^{'} + 24,\ \ Cy_0 = 1,\ \text{ otherwise} \\ k_1 &= k_1^{'}. \end{aligned} \tag{23}$$

A block diagram of the implementation is sketched in Fig. 6. As shown, to verify if the addition of the lowest digits, $k_0^{'}$, is lower than 1000, $k_0^{'}$ is added with 24 and the most significant bit of the result is checked. If it is '0' it means that result is lower than 1024 and so $k_0^{'}$ is lower than 1000. This avoids the utilization of a comparator.

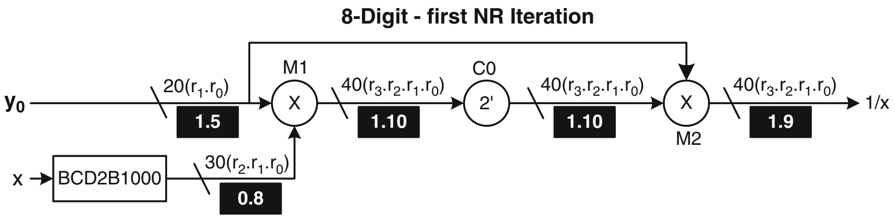


**Fig. 7** NR iteration unit for the 8-digit reciprocal

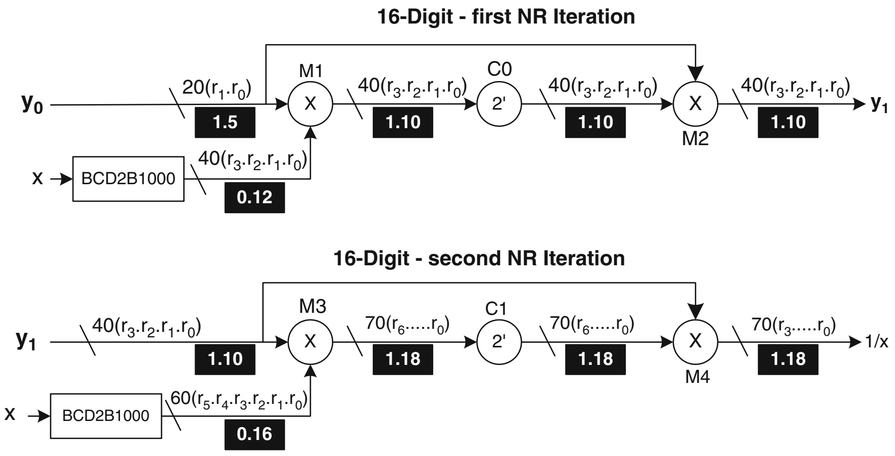


**Fig. 8** First and second NR iteration units for the 16-digit reciprocal

## *3.2 NR Iterations*

The 8-digit reciprocal is calculated with a single NR iteration:

$$\frac{1}{x} = y^{(0)} \times (2 - x \times y^{(0)})$$

while the 16-digit reciprocal calculation requires two NR iterations:

$$y^{(1)} = y^{(0)} \times (2 - x \times y^{(0)})$$
$$\frac{1}{x} = y^{(1)} \times (2 - x \times y^{(1)}).$$

Each NR iteration needs two multipliers and a circuit to calculate $(2 - w)$. However, the precision (number of digits) considered at each intermediate calculation is different for each case (see Figs. 7 and 8).

The inputs of the NR iteration unit, for the 8-digit reciprocal, are the output of the minimax approximation unit and the eight fractional digits of x in radix-1000. The block *BCD2B1000* converts the decimal number, *x*, to three radix-1000 digits.

The first multiplication, M1, must keep ten fractional digits plus an integer one (11 digits), and therefore four radix-1000 digits ($4 \times 3$ decimal digits) are kept at the output of the multiplier. Next, the block 2' implements $2 - x \times y^{(0)}$ and maintains the same precision. Finally, the operands of the last multiplier, M2, are two radix-1000 numbers with two and four radix-1000 digits, respectively, and four radix-1000 digits are kept at the output to guarantee a result with one integer and nine fractional digits.

For the 16-digit reciprocal case, the number of fractional digits of $x$ used in the first multiplication increases from eight to twelve, and the final result of the first iteration unit must keep ten fractional digits instead of nine. The output of the second iteration unit keeps one integer digit and 18 fractional digits.

In all implementations, the multiplications are done with iterative radix-1000 multipliers. The dimensions of the multipliers in terms of radix-1000 operands are as follows:

| Divider | M1 | M2 | M3 | M4 |
|---|---|---|---|---|
| 8-Digit | $2 \times 3$ | $2 \times 4$ | – | – |
| 16-Digit | $2 \times 4$ | $2 \times 4$ | $4 \times 6$ | $4 \times 7$ |

The following sections describe the implementation of the iterative radix-1000 multipliers, the decimal to radix-1000 converter, and the $(2 - w)$ block.

## Radix-1000 Multiplier

The radix-1000 multiplications of the divider are implemented with iterative multipliers. As referred, using parallel multipliers would turn the solution very expensive in terms of resources.

In each iteration of the multiplication, two radix-1000 digits are multiplied and the result is accumulated. This allows the utilization of a binary multiplier since a radix-1000 digit is in binary format. The results published in [22] suggest that this method is very efficient as long as binary to decimal conversions of large numbers are avoided since the size of binary to decimal converters increases more than linearly [21] with the size of the operands.

Formally, given two radix-1000 numbers, $g$ and $h$, in the form

$$g = g_{n-1}g_{n-2}\cdots g_2g_1g_0$$
$$h = h_{n-1}h_{n-2}\ldots h_2h_1h_0$$

or

$$g = \sum_{i=0}^{n-1} g_i 10^{3\times i}, \quad h = \sum_{i=0}^{n-1} h_i 10^{3\times i}$$

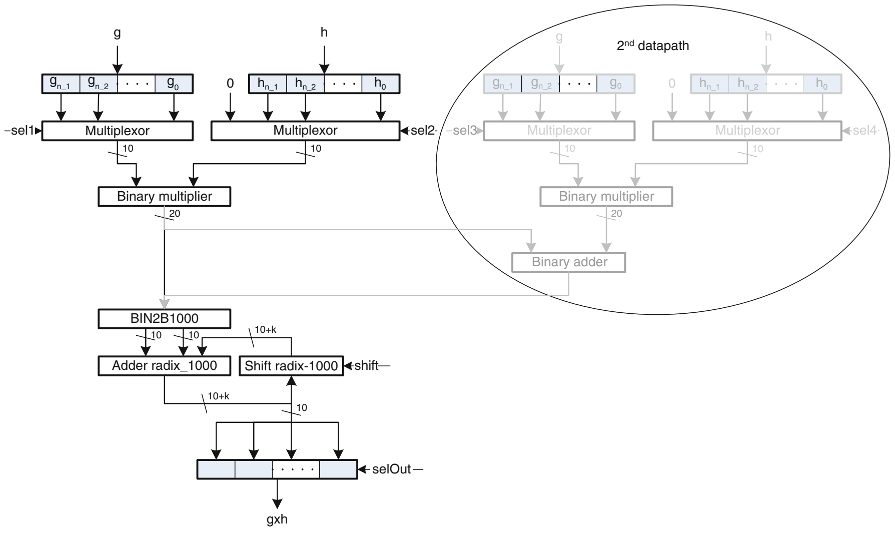


**Fig. 9** Iterative radix-1000 multiplier using binary arithmetic

the iterative multiplication is given by

$$g \times h = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} g_i \times h_j 10^{3\times(i+j)}. \tag{24}$$

The architecture of the iterative multiplier is illustrated in Fig. 9.

At each step, a pair of radix-1000 digits is selected at the input multiplexers and multiplied. The result of the multiplication is a 20-bit binary number that is converted to two radix-1000 digits, which are then accumulated also in radix-1000 with the previous result. Working with radix-1000 reduces the division by 1000 to a shift of a digit radix-1000.

The shifts take place according to the multiplication algorithm. According to Eq. (24), the first shift happens after one accumulation ($g_0h_0$), the second shift after two accumulations ($g_0h_1 + g_1h_0$), until a maximum of $n$ accumulations, followed by $n-1$, $n-2$ accumulations, and so on.

To reduce the number of iterations, a second internal path for the parallel calculation of partial products may be used. This second path is only used in the second NR iteration, as explained later.

The following sections describe the implementation of the binary to radix-1000 converter (*BIN2B1000* component) and the radix-1000 adder with accumulation.

## Binary to Radix-1000 Converter

In the single path multiplier, the input of the binary to radix-1000 converter is 20 bits large, and in the double path multiplier is 21 bits large.

The input of the 20 bit converter is a binary number $b \in [0, 999999]$ and the output is a two radix-1000 digits number, $r_1, r_0$, that is

$$b = r_1 \cdot 10^3 + r_0 = r.$$

Considering that

$$b = b_1 \cdot 2^{10} + b_0$$

it follows that

$$\begin{aligned} b &= b_1 \cdot 1024 + b_0 & & b_1 \leq 974 = \tfrac{999\times 999}{1024} \\ &= b_1 \cdot 1000 + \underbrace{b_1 \cdot 24 + b_0}_{c} & & b_0 \leq 1023, \end{aligned} \tag{25}$$

where

$$\begin{aligned} c &= b_1 \cdot 24 + b_0 & & c \leq 24399 & & \leftarrow 15\,\text{bits} \\ c &= c_1 \cdot 1024 + c_0 \\ &= c_1 \cdot 1000 & & c_1 \leq 23 & & \leftarrow 5\,\text{bits} \\ &\quad + c_1 \cdot 24 & & c_1 \times 24 \leq 23 \times 24 \\ &\quad + c_0 & & c_0 \leq 1023 & & \leftarrow 10\,\text{bits.} \end{aligned} \tag{26}$$

From equations (25) and (26), $b$ is given by:

$$b = (b_1 + c_1) \cdot 1000 + c_1 \cdot 24 + c_0 \tag{27}$$

and a first approximation for the two digits is

$$\hat{r}_1 = b_1 + c_1 \leq 997 \quad \leftarrow 10\,\text{bits} \tag{28}$$

$$\hat{r}_0 = c_1 \cdot 24 + c_0 \leq 1585 \quad \leftarrow 11\,\text{bits}, \tag{29}$$

where

$$\begin{aligned} \hat{r}_1 &\in [r_1 - 1, r_1] \\ \hat{r}_0 &\in [0, 1575]. \end{aligned}$$

From these, the final step is to test if $\hat{r}_0$ is higher than 999, that is, if $\hat{r}_0 + 24$ is higher than 1023. If yes, $\hat{r}_1$ must be incremented by one and $\hat{r}_0$ must be increased by 24 to adjust the result. Otherwise, the result is already correct.

The converter consists of a set of adders and a ROM to convert a $k \times 2^{10}$ number to radix-1000, as shown in Fig. 10.

The input of the 21-bit converter is a binary number $b \in [0,\ 2 \times 999 \times 999]$, and the outputs are a two radix-1000 digits number, $r_1 r_0$, and a carry out (Cout). The converter is similar to the case with 20 bits, except that an extra step is needed to extract the carry out from the second digit (see Fig. 11).

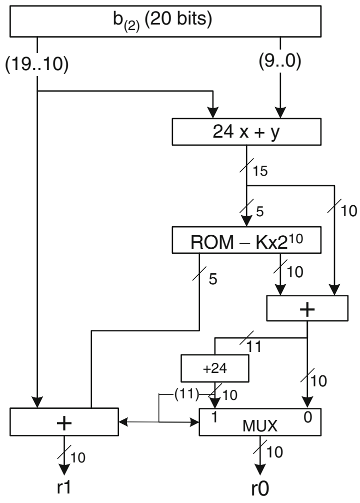


**Fig. 10** 20-bit binary to radix-1000 converter

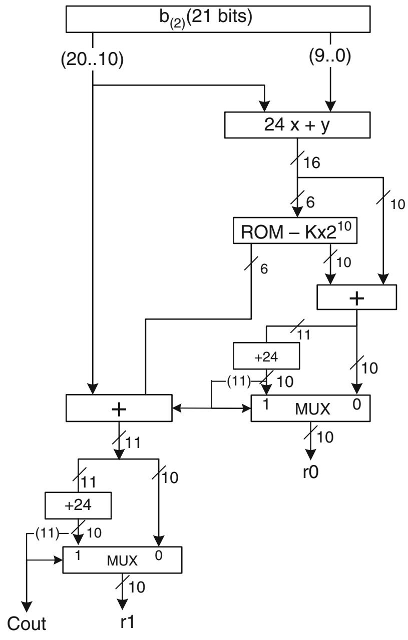


**Fig. 11** 21-bit binary to radix-1000 converter

### 3.2.1 Radix-1000 Accumulator

The radix-1000 accumulator adds radix-1000 digits in the interval $[0,\ 999\times999]$ for the single datapath and in $[0,2\times999\times999]$ for the double datapath multiplier. Considering an $m\times n$ radix-1000 multiplier, $\max(n,m)$ accumulations can occur before a shift takes place. The biggest multiplier used to implement the divider is

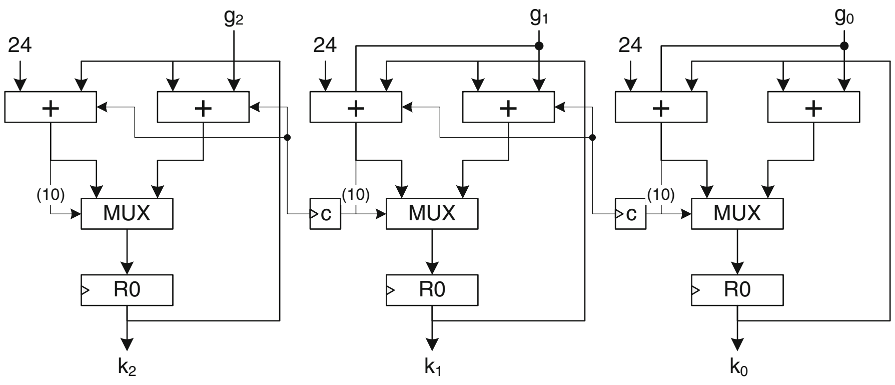


**Fig. 12** Radix-1000 accumulator

a $6 \times 7$ multiplier. Therefore, a three radix-1000 digits accumulator is enough to implement any of the iterative multipliers utilized to implement the divider.

Given two radix-1000 numbers, $g = g_2g_1g_0$ and $h = h_2h_1h_0$ the addition $k = k_2k_1k_0 = g + h$, is calculated as follows:

$$\begin{aligned} k_0^{'} &= g_0 + h_0 \\ k_1^{'} &= g_1 + h_1 + Cy0 \\ k_2^{'} &= g_2 + h_2 + Cy1 \\ k_m &= k_m^{'},\ \ Cym = 0,\ \text{ if } k_m^{'} < 1000 \\ &= k_m^{'} + 24,\ \ Cym = 1,\ \text{ otherwise.} \end{aligned} \tag{30}$$

According to the block diagram of the iterative decimal multiplier (Fig. 9), the radix-1000 number obtained after conversion of the result from the binary multiplier is added to the previous accumulation. A direct implementation of the algorithm would have a high carry propagation delay. To improve the performance, we have implemented a carry save adder and used a similar scheme to test if the number is higher than 999.

The final implementation is shown in Fig. 12. Each input digit is added to both the previously accumulated digit of the same arithmetic weight and with 24. The results from both adders are multiplexed according to the most significant bit of the plus 24 adder. The result and the carry out are registered to implement carry save addition. In the case of a single datapath multiplier, $g_2$ = “0”. For a double datapath, $g_2 = Cout$, where *Cout* comes from the binary to radix-1000 converter.

### 3.2.2 Decimal to Radix-1000 Converter

Decimal to radix-1000 conversion is straightforward. Each radix-1000 results from the conversion of three decimal digits to binary. Formally, given a decimal number

$d_{n-1}d_{n-2}\ldots d_1d_0$, the radix-1000 equivalent, $r_{n-1}r_{n-2}\ldots r_1r_0$, is determined as follows:

$$r_i = DEC2BIN(d_{2+3\times i}d_{1+3\times i}d_{0+3\times i}),\ i = 0,1,2,\ldots \tag{31}$$

The decimal to binary conversion of three digits is obtained, using binary arithmetic, as

$$b = d_2 10^2 + d_1 10 + d_0 \tag{32}$$

whose implementation is straightforward.

### 3.2.3 Two's Complement of a Radix-1000 Number

Each iteration of the NR has a multiplicative factor of the form $(2-w)$, where $w = w_{n-1}w_{n-2}\ldots w_1w_0$ is the output of the first multiplier of the iterative unit represented in radix-1000. $w$ has one integer digit equal to 0 or 1. Given a number in radix-1000 with $n$ digits, $w$, and knowing that the most significant digit is only 0 or 1, $(2-w) = c = c_{n-1}c_{n-2}\ldots c_1c_0$ is calculated as follows:

$$\begin{aligned}
c_0^{'} &= 1000 - w_0\\
c_0 &= 0,\ Cy_0 = 1,\ if\ c_0^{'} = 1000\\
&= c_0^{'},\ Cy_0 = 0,\ \text{otherwise}\\
c_1^{'} &= 999 - w_1 + Cy_0\\
c_1 &= 0,\ Cy_1 = 1,\ if\ c_1^{'} = 1000\\
&= c_1^{'},\ Cy_1 = 0,\ \text{otherwise}\\
\ldots &= \ldots\\
c_{n-1} &= 1 - w_{n-1} + Cy_{n-2}.
\end{aligned} \tag{33}$$

## 3.3 *Binary to BCD Converter*

The BCD converter transforms a 10-bit binary number, $b$, into a BCD number with three digits, $d_2$, $d_1$, and $d_0$. Binary to decimal conversion is fundamentally the calculation of the following polynomial, using decimal arithmetic:

$$b = b_{n-1}\cdot 2^{n-1} + b_{n-2}\cdot 2^{n-2} + \ldots + b_0\cdot 2^0 \tag{34}$$

or

$$b = (((b_{n-1}\cdot 2) + b_{n-2})\cdot 2 + b_{n-3})\cdot 2 + \ldots + b_0. \tag{35}$$

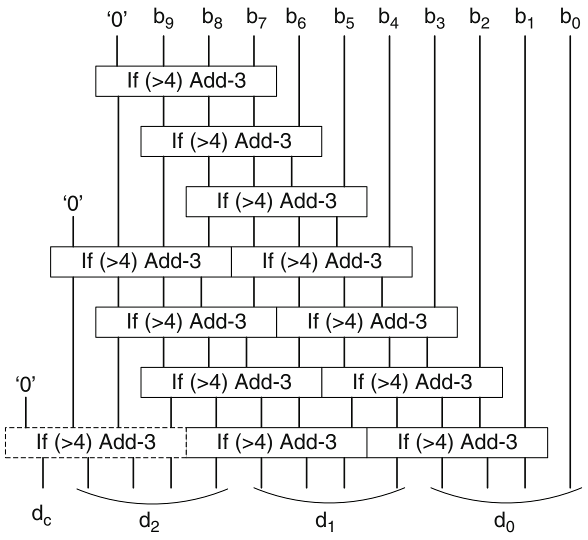


**Fig. 13** Add-3 and shift algorithm for a 10-bit binary number

Multiplication by two is achieved with a shift towards the most significant bit. However, since the operations are in decimal whenever a bit shifts across a boundary of a digit, the digit must be corrected by adding three before the shift takes place [2] (or six after the shift). The three-digit binary to decimal converter (see Fig. 13) was implemented using this algorithm, which is usually known as the add-3 and shift algorithm .

In the converter used in the divider, the binary number to be converted is always less than 1000. Whenever this is the case, the left bottom block of the converter (dashed block in Fig. 13) can be removed since $d_c$ is zero.

## *3.4 Improving the Performance of the Divider*

The performance of the decimal divider depends mainly on the efficiency of the iterative multipliers. The maximum operating frequency of the multiplier is constrained by the initial datapath (multiplexer, binary multiplier and binary to radix-1000 converter) that calculates the partial products, which are successively accumulated. To improve its performance, and consequently the performance of the divider, the multiplier datapath is pipelined such that each stage is optimally balanced with the internal loop longest path. Consequently, the maximum frequency becomes constrained by the internal loop consisting of the radix-1000 accumulator.

Also, a multiplier in the datapath of the complete divider can start calculating as soon as the first partial product of the previous operation is available. Therefore, the execution of the multipliers can overlap in time. A more detailed accounting of the number of cycles is provided in the following section.

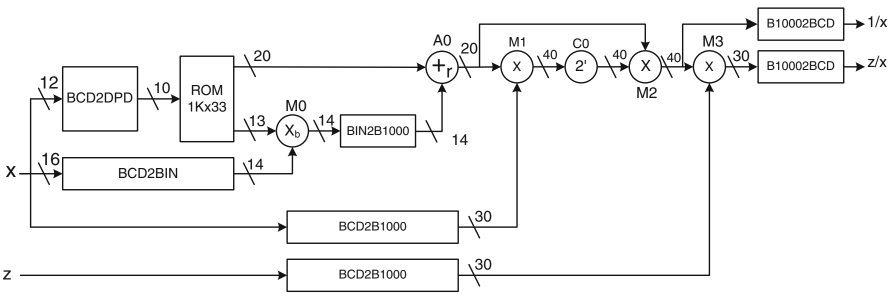


**Fig. 14** Final architecture of the 8-digit divider

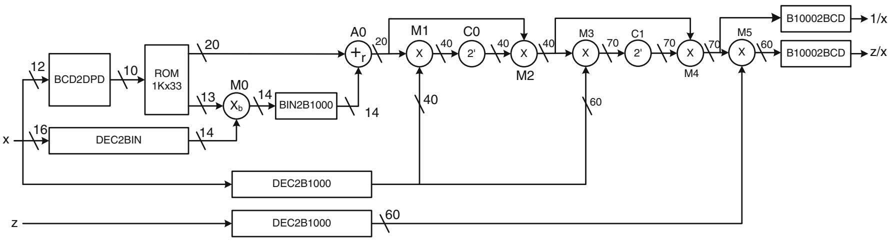


**Fig. 15** Final architecture of the 16-digit divider

The complete divider architecture comprises the minimax approximation block, the NR iterations components, the final multiplier ($\frac{1}{x} \times z$), and the radix-1000 to decimal converters, as shown in Figs. 14 and 15.

The dividers provide as output results both the reciprocal and the quotient.

## 4 Results

The architectures of the dividers for operands of size 8 and 16 were specified in VHDL. The circuits were synthesized, placed, and routed with Xilinx ISE13.1 and implemented in a Virtex-4 SX35-12 FPGA and in a Virtex-6 VLX75T. The results were compared with an SRT-like divider from [6], which proposed a nonrestoring algorithm (alg1) and an SRT-like algorithm (alg2), both implemented in a Virtex-4 FPGA.

Two different architectures were considered for the 8-digit divider, A8Single and A8Double, with the following characteristics:

- A8Single—All multipliers use a single datapath
- A8Double—The final multiplier uses a double datapath

Three different architectures were considered for the 16-digit divider, A16Single, A16Double and A16OneDouble, with the following characteristics:

**Table 4** Results for the 8-digit divider in a Virtex-4 (times in ns)

| $A/B$ | FF | LUT | BRAMs | DSPs | Cycles | $T_{clk}$ | Exec (cycles $\times$ $T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|---|---|---|---|---|---|---|---|---|---|
| A8Single | 1371 | 2016 | 2 | 0 | 51 | 3.4 | 173 | 14 | 21 |
| A8Single | 1101 | 1605 | 2 | 5 | 51 | 3.4 | 173 | 14 | 21 |
| A8Double | 1489 | 2224 | 2 | 0 | 47 | 3.4 | 160 | 12 | 24.5 |
| A8Double | 1267 | 1783 | 2 | 5 | 47 | 3.4 | 160 | 12 | 24.5 |
| alg1[6] | 151 | 2008 | 0 | 0 | 10 | 20.5 | 205 | 10 | 4.9 |
| alg2[6] | 231 | 2612 | 0 | 0 | 10 | 16.4 | 164 | 10 | 6.1 |

**Table 5** Results for the 16-digit divider in a Virtex-4 (times in ns)

| $A/B$ | FF | LUT | BRAMs | DSPs | Cycles | $T_{clk}$ | Exec (cycles $\times$ $T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|---|---|---|---|---|---|---|---|---|---|
| A16Single | 2098 | 2756 | 2 | 0 | 118 | 3.4 | 401 | 43 | 6.8 |
| A16Single | 1820 | 2091 | 2 | 7 | 118 | 3.4 | 401 | 43 | 6.8 |
| A16Double | 2361 | 3768 | 2 | 0 | 96 | 3.4 | 326 | 25 | 11.7 |
| A16Double | 2172 | 2718 | 2 | 10 | 96 | 3.4 | 326 | 25 | 11.7 |
| alg1[6] | 246 | 2974 | 0 | 0 | 18 | 21 | 386 | 18 | 2.6 |
| alg2[6] | 342 | 3799 | 0 | 0 | 18 | 16.6 | 300 | 18 | 3.3 |

**Table 6** Results for the 8-digit divider in a Virtex-6 (times in ns)

| $A/B$ | FF | LUT | BRAMs | DSPs | Cycles | $T_{clk}$ | Exec (cycles $\times$ $T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|---|---|---|---|---|---|---|---|---|---|
| A8Single | 1355 | 1549 | 1 | 0 | 51 | 2.6 | 135 | 14 | 27 |
| A8Single | 1009 | 987 | 1 | 4 | 51 | 2.6 | 135 | 14 | 27 |
| A8Double | 1427 | 1737 | 1 | 0 | 47 | 2.6 | 122 | 12 | 32 |
| A8Double | 1182 | 1166 | 1 | 4 | 47 | 2.6 | 122 | 12 | 32 |

- A16Single—All multipliers use a single datapath
- A16Double—Multipliers in the second NR iteration and the final multiplier use a double datapath
- A16OneDouble—Only the final multiplier uses a double datapath

Also, for each architecture, implementations with different performance/area tradeoffs were considered, including implementations utilizing dedicated DSPs of the FPGA. The results obtained (after place and route) are summarized in Tables 4 and 5, for the Virtex-4, and in Tables 6 and 7, for the Virtex-6 FPGA.

The throughput indicated in the tables is based on the number of cycles after which a new division can start (# CBD).

The total number of cycles depends not only on the number of cycles of each block but also on the overlap of execution times of each multiplier, as shown in Figs. 16 and 17. The figures indicate the execution and the starting times (in clock cycles) of each hardware block in the datapath. Since the execution of the last multiplier overlaps only with the previous block, it is important to speed up its execution. This is the reason why the A16OneDouble architecture, where only the

**Table 7** Results for the 16-digit divider in a Virtex-6 (times in ns)

| $A/B$ | FF | LUT | BRAMs | DSPs | Cycles | $T_{clk}$ | Exec (cycles × $T_{clk}$) | # CBD | Throughput (Mdiv/s) |
|---|---|---|---|---|---|---|---|---|---|
| A16Single | 2545 | 2517 | 1 | 0 | 118 | 2.7 | 321 | 43 | 8.6 |
| A16Single | 2637 | 2822 | 1 | 0 | 118 | 2.6 | 309 | 43 | 8.9 |
| A16Single | 2213 | 1771 | 1 | 6 | 118 | 2.7 | 321 | 43 | 8.6 |
| A16Double | 2875 | 3156 | 1 | 0 | 96 | 2.7 | 262 | 25 | 14,8 |
| A16Double | 2934 | 3452 | 1 | 0 | 96 | 2.6 | 252 | 25 | 15,4 |
| A16Double | 2390 | 2026 | 1 | 9 | 96 | 2.7 | 262 | 25 | 14,8 |
| A16OneDouble | 2801 | 3020 | 1 | 0 | 108 | 2.7 | 294 | 30 | 12.3 |
| A16OneDouble | 2603 | 2720 | 1 | 0 | 108 | 2.6 | 283 | 30 | 12.8 |
| A16OneDouble | 2376 | 2235 | 1 | 7 | 108 | 2.7 | 294 | 30 | 12.3 |

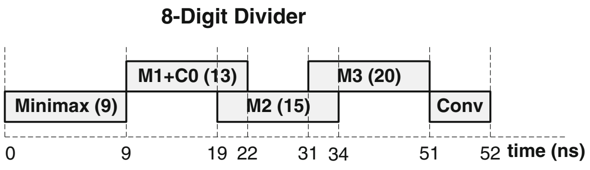


**Fig. 16** Temporal diagram of the execution of the 8-digit divider

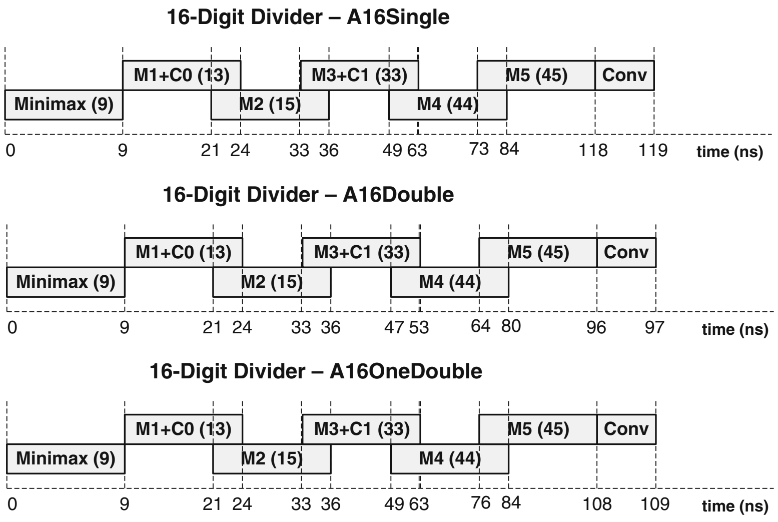


**Fig. 17** Temporal diagram of the execution of the 16-digit divider for the three different architectures

last multiplier uses a double datapath, achieves almost 50 % (10 ns) of the total performance improvement (22 ns) achieved with the architecture A16Double, where both multipliers of the second NR iteration and the last multiplier have a double datapath.

From the tables, we observe and conclude the following:

- The proposed $8 \times 8$ decimal divider without DSPs utilizes about the same area of Alg1 with a 16 % reduction in execution time. The solution using a double datapath in the final multiplier achieves an 8 % improvement in the execution time, with a 10 % increase in area.
- The solutions with DSPs have a 10 % better execution time than the implementations without DSPs and save around 20 % of LUTs.
- The throughput of the proposed $8 \times 8$ decimal divider is 4 to 5 times better compared to those obtained with alg1 and alg2.
- For the $16 \times 16$ divider, the area and execution time of both solutions are similar. In terms of throughput, the proposed divider is almost four times better.
- The execution time in Alg1 and Alg2 increases linearly with the number of digits. However, it increases more than linearly in the proposed divider. On the other hand, the area increases linearly in Alg1 and more than linearly in our proposal and in Alg2. So, for larger operands, the efficiency of the NR method will decrease compared to Alg1 and Alg2.
- Similar conclusions can be taken for the Virtex-6 implementations. These utilize less LUTs since we are dealing with 6-input LUTs instead of 4-input LUTs of Virtex-4 FPGAs. Also, the execution time is from 25 % to 30 % better compared to the implementation using a Virtex-4 FPGA.

## 5 Conclusions

Iterative dividers of sizes $8 \times 8$ and $16 \times 16$ based on the Newton–Raphson method with an initial minimax approximation were proposed and implemented in reconfigurable hardware.

The NR-based method for division has the advantage of producing the reciprocal that can be used for successive divisions by the same number. However, the remainder is not available.

The results show that the division based on the NR iterative method is competitive with SRT-based solutions, achieving significantly higher throughputs, as long as it uses a good initial approximation and efficient multipliers. Our approach, using the minimax polynomial for initial approximation, needs one less iteration than the Taylor-based approximation to achieve the precision required.

Also, the proposed decimal multipliers use binary multipliers. This is specially attractive when the target technology includes embedded binary multipliers, such as FPGAs.

**Acknowledgements** This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

## References

1. POWER6 Decimal Floating Point (DFP) (2009). URL http://www.ibm.com/developerworks/wikis/display/WikiPtype/Decimal+Floating+Point
2. Alfke P, New B (1997) Serial code conversion between BCD and binary. In: Xilinx application note XAPP 029
3. Cowlishaw M (2002) Densely packed decimal encoding. IEE Proc Comput Digit Tech 149(3):102–104. DOI 10.1049/ip-cdt:20020407
4. Cowlishaw MF (2003) Decimal floating-point: algorism for computers. In: Proceedings IEEE 6th IEEE international symposium on computer arithmetic, pp 104–111
5. Dadda L, Nannarelli A (2008) A variant of a radix-10 combinational multiplier. In: Proceedings IEEE international symposium on circuits and systems (ISCAS), pp 3370–3373
6. Deschamps JP, Sutter G (2010) Decimal division: Algorithms and FPGA implementations. In: Proceedings IEEE southern conference on programmable logic, pp 67–72
7. Erle MA, Schulte MJ (2003) Decimal multiplication via carry-save addition. In: Proceedings IEEE 14th IEEE international conference on application specific systems, pp 348–358
8. Kenney RD, Schulte MJ, Erle MA (2004) High-frequency decimal multiplier. In: Proceedings IEEE international conference on computer design: VLSI in computers and processors, pp 26–29
9. Lang T, Nannarelli A (2006) A radix-10 combinational multiplier. In: Proceedings IEEE 40th international asilomar conference on signals, systems, and computers, pp 313–317
10. Lang T, Nannarelli A (2007) A radix-10 digit-recurrence division unit: algorithm and architecture. IEEE Transactions on Computers, 56(6):1–13
11. Louvet N, Muller JM, Panhaleux A (2010) Newton–Raphson algorithms for floating-point division using an FMA. In: Proceedings IEEE 20th international conference on application-specific systems architectures and processors, pp 200–207
12. Muller JM (2006) Elementary functions—algorithms and implementation. Birkhauser, Basel
13. Neto HC, Véstias MP (2008) Decimal multiplier on fpga using embedded binary multipliers. In: Proceedings IEEE 20th international conference on field programmable logic and applications, pp 197–202
14. Nikmehr H, Phillips B, Lim CC (2006) Fast decimal floating-point division. IEEE Trans VLSI Syst 14(9):951–961
15. Parhami B (2000) Computer arithmetic—algorithms and hardware designs. Oxford University Press, Oxford
16. Rump SM, Ogita T, Oishi S (2008) Accurate floating-point summation part i: faithful rounding. SIAM J Sci Comput 31:189–224. DOI http://dx.doi.org/10.1137/050645671. URL http://dx.doi.org/10.1137/050645671
17. Suetin PK (2001) Chebyshev polynomials, Encyclopedia of mathematics edition. Springer, Berlin
18. Sutter G, Todorovich E, Bioul G, Vazquez M, Deschamps JP (2009) FPGA implementations of BCD multipliers. In: Proceedings IEEE international conference on reconfigurable computing and FPGAs, pp 36–41
19. Vázquez A, Antelo E, Montuschi O (2007) A radix-10 SRT divider based on alternative BCD codings. In: Proceedings IEEE international conference on computer design, pp 280–287
20. Vázquez A, Antelo E, Montushi P (2007) A new family of high-performance parallel decimal multipliers. In: Proceedings IEEE 18th symposium on computer arithmetic, pp 195–204
21. Véstias MP, Neto HC (2010) Parallel decimal multipliers using binary multipliers. In: Proceedings IEEE southern conference on programmable logic, pp 73–78
22. Véstias MP, Neto HC (2011) Iterative decimal multiplication using binary arithmetic. In: Proceedings IEEE southern conference on programmable logic, pp 257–262
23. Wang LK, Schulte M (2004) Decimal floating-point division using newton-raphson iteration. In: Proceedings IEEE international conference on application-specific systems, Architectures and processors, pp 84–95

# Lifetime Reliability Sensing in Modern FPGAs

**Abdulazim Amouri and Mehdi Tahoori**

## 1 Introduction

The need for high performance has been the main motivation toward more down-scaling of *complementary metal oxide semiconductor* (CMOS) devices [1]. This down scaling enables the integration of billions of transistors on a single die [2, 3]. FPGAs are in the front line to exploit the latest advancements in CMOS technology, because their high volume, regularity, and scalability, to cope with highest performance demands for digital and some mixed-mode analog applications. Unfortunately, this downscaling does not come cost-free. Some of the challenges include manufacturing variability, sub-threshold leakage, power dissipation, increased circuit noise sensitivity, and reliability concerns due to transient (e.g., radiation-induced soft errors) and permanent (e.g., transistor aging) failures [4, 5]. Transistor aging is one of the most important reliability challenges at nanoscale CMOS technology. Two important causes are *bias-temperature instability* (BTI) and *hot carrier injection* (HCI) [6]. The BTI consists of two independent phenomena: negative BTI (NBTI) that affects the PMOS transistors and positive BTI (PBTI) that affects the NMOS transistors. The NBTI/PBTI occurs when the PMOS/NMOS transistor is negatively/positively biased at elevated temperature. The HCI on the other side happens due to carrier trapping in the interface region between the channel and the gate dielectric. The major effect of the BTI and HCI is that they increase the magnitude of the transistor's threshold voltage and reduce the effective carrier mobility over time. That leads to a reduction in the switching speed of the transistor [7].

A. Amouri (✉) • M. Tahoori
Chair of Dependable Nano Computing (CDNC), Karlsruhe Institute of Technology (KIT), Haid-und-Neu-Str. 7, D-76131 Karlsruhe, Germany
e-mail: abdulazim.amouri@kit.edu; mehdi.tahoori@kit.edu

This means it causes time-increasing *path delay faults* in the circuit. Once the delay of critical paths exceed the clock period, the correct functionality of the circuit is affected. Furthermore, the accumulated effects of NBTI and HCI push toward the wear-out phase, so the transistor is aged at a faster rate [8].

As FPGAs use the latest trends in CMOS technology, it is important to address nanoscale reliability concerns for FPGAs and have counter-measures against them. In this chapter, we present a logic-level circuitry for detection of late transitions that happen due to transistor aging in modern FPGAs. We take advantage of the resources available in FPGAs to design and implement low-cost and highly accurate online aging sensors. We also provide a scheme to select which paths are to be monitored (the most aging-vulnerable paths) in the circuit using the available FPGA tools in order to have a high-efficient monitoring.

By using our sensor mapping techniques, the sensitivity of the sensor can selectively be adjusted to the range from a warning sensor to a late transition detector with a desired window. When used as a warning sensor, it can signal aging when the transitions happen in the timing guards, to be able to detect and mitigate aging of critical paths before it causes erroneous captures.

Unlike most of previous work which are based on ring oscillators and counters to measure variation and aging across the FPGA chip [9, 10], our proposed sensor is *application dependent*, that is it monitors the correct functionality of critical paths in the FPGA-mapped design. To the best of our knowledge, we present the first approach for design and mapping of a logic-level aging sensor for FPGA-based designs.

The rest of the chapter is organized as follows. In Sect. 2, the term of transistor aging is explained, then in Sect. 3, the related work is reviewed. Section 4 presents the main idea of the proposed aging sensor. FPGA mappings of the sensor is discussed in Sect. 5. Section 6 contains the experimental results and analysis of the aging sensors. Finally, Sect. 7 concludes this paper.

## 2 Transistor Aging

Aging is one of the most important reliability issues facing the VLSI devices at nano-scale. It happens gradually on a long time scale, in which the performance of the device decays slowly until reaching a critical limit that causes failures in the circuit. There are several reasons for device aging; two of the most important ones are BTI [11, 12] and HCI [7, 8]. These two phenomena cause the delay of the transistors to increase, which in turn increase the total delay of the paths in the circuit, and once the critical path delay exceeds the timing (defined by the running frequency), the chip starts to fail, and ultimately, the lifetime will be reduced.

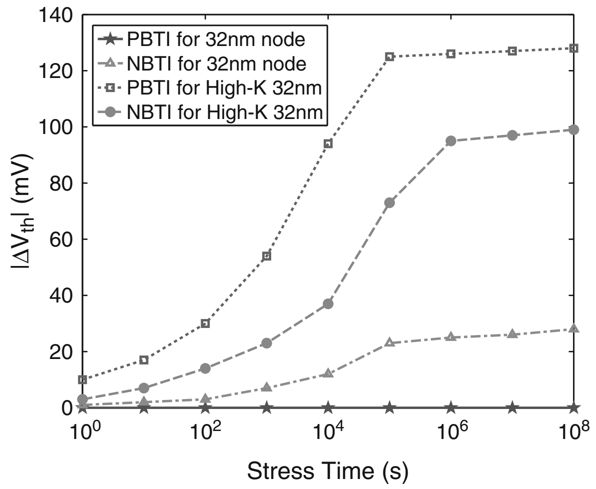


**Fig. 1** $V_{th}$ shift induced by NBTI and PBTI [12]

## 2.1 *Bias Temperature Instability*

The BTI phenomenon causes the magnitude of the threshold voltage of transistors to increase [13]. Hence, the switching delay of the transistor increases, and as a result, once the delay of functional paths exceeds the timing requirements, the circuit starts to fail. This can greatly reduce the operational lifetime of FPGA chips.

BTI consists of two different phenomena: NBTI, which has an effect on PMOS transistors, and PBTI, affecting NMOS transistors. Since in previous technology nodes and fabrication schemes, the PBTI effect was negligible in comparison to NBTI, it was mostly ignored. However, since the introduction of high-$\kappa$/metal gates transistors in sub 45 nm technology, the PBTI effect becomes comparable to the NBTI one [12, 14, 15] (see Fig. 1). Consequently, both effects should be considered in new technologies. NBTI (PBTI) has two phases:

- Stress phase: at which the gate-source voltage is reversely (positively) biased ($V_{gs} = -(+)V_{dd}$).
- Relaxation phase: ($V_{gs} = 0$).

During the stress phase (i.e., when the transistor is on: $V_{gs} = -V_{DD}$ for PMOS under NBTI and $V_{gs} = V_{DD}$ for NMOS under PBTI), some interface traps are generated at the interface of channel and gate oxide. The generated interface traps cause the magnitude of threshold voltage ($V_{th}$) to increase. On the other hand, during the relaxation phase ($V_{gs} \approx 0$), some of the interface traps are removed, and as a result, the magnitude of $V_{th}$ of the transistor decreases [16]. However, it should be noted that this recovery cannot completely compensate the effect of stress phase. Consequently, the overall effect of BTI is an increase in the magnitude of threshold voltage over the time (see Fig. 2).

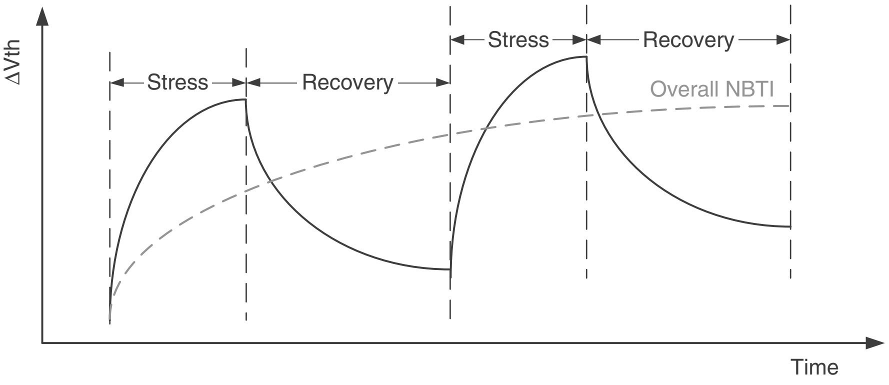


**Fig. 2** BTI induced $V_{\text{th}}$ change during stress and recovery

## *2.2 Hot Carrier Injection*

Similar to the BTI, the HCI phenomenon, causes also the threshold voltage of the transistors to increase, which in turn leads to similar consequences, like BTI, increasing the delay of the functional paths and reducing the lifetime of the FPGA chips.

HCI happens when a high voltage is applied at the drain ($V_D > V_G$) causing the channel carriers (electrons for NMOS and holes for PMOS) to be accelerated into the depletion region of the drain. These accelerated carriers will get collided with the silicon atoms. The effect is that some of them will gain a little more energy than the average, which makes them able to overcome the electric potential barrier between the silicon substrate and the gate oxide, and will get injected into the gate oxide layer where they are sometimes trapped. The worst case is when $V_D = 2V_G$ [17]

Over the time, these trapped carriers will eventually build up electric charge within the dielectric layer, which will increase the threshold voltage needed to turn the transistor on. This mechanism is called *drain avalanche hot carrier* (DAHC) injection (see Fig. 3). In fact, there are other three mechanisms in which HCI is encountered [18]. However, DAHC is the worst among them.

## 3 Related Work

For delay fault testing in FPGAs, a *Built-In Self-Test* (BIST)-based approach has been presented in [19], which stimulates several paths with a same length, then compares their output to detect faults. However, this method targets mainly manufacturing delay faults. In [7], a method is developed for measuring and monitoring degradation in an FPGA, based on measuring the difference on transitions at inputs and outputs and their probabilities for a single path, but it is used offline to find at

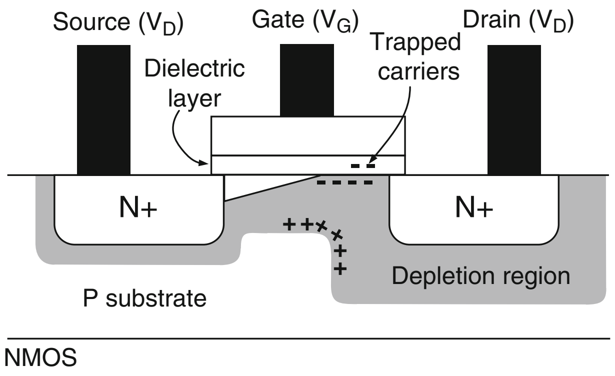


**Fig. 3** HCI–DAHC mechanism [17]

which frequency the circuit starts to fail. This chapter also estimates the effects of the aging phenomena based on an NBTI model. The authors in [9] present a multiuse sensor implemented in reconfigurable logic in order to help estimating variation in delay, static and dynamic power, and temperature. The sensor is based on a ring oscillator and a residue-number-system ring counter, where an array of such sensors are arranged among the FPGA chip in order to measure the needed variation. Each sensor occupies 8 LUT of a Virtex-5 FPGA. The relation between the frequency of the ring oscillator and the desired parameter variation is used to estimate that variation. In [20], a delay measurement method based on transition probabilities (TPs) is presented. A set of test vectors are used to test the desired paths in the circuit, and to calculate the TPs at the output of each path, then based on the TPs, the delay of the paths are determined.

In the scope of reliability analysis for FPGAs, a dynamic thermal-aware reliability management framework has been presented in [10] which estimates the lifetime reliability of FPGAs, based on estimation of several phenomenas and hard errors like *time-dependant-dielectric-breakdown* (TDDB) and *electron migration* (EM), in addition to NBTI impacts. The proposed framework uses some tools and simulators to calculate the temperature variation across the chip, the switching activity of the design, and the static probability of the signals, to do the estimation of TDDB, EM, and NBTI. Performance degradation of FPGAs due to HCI and TDDB has been analyzed in [21], and some load balancing and alternate routing techniques have been proposed to improve the reliability (mean time to failure) of the FPGA chip. The first proposed technique is to use controlled input vectors to optimize the active leakage power of logic blocks and hence reduce the TDDB effect. The second technique is to balance the load on the circuit and hence mitigating the HCI impact. Another technique of using a selective alternate routing is used to reduce the EM impact. In [22], accelerated life tests are performed on FPGAs, to study the effects of the degradation, and three degradation-mitigation strategies

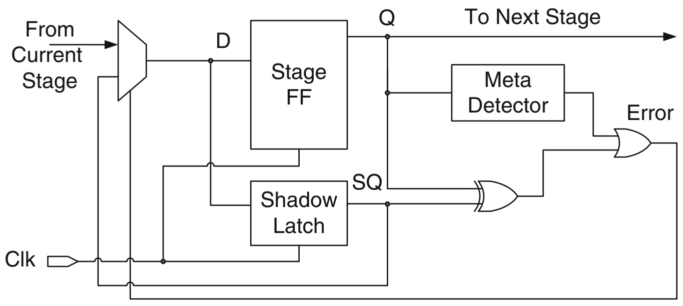


**Fig. 4** Razor I technique [25]

are also discussed. The first strategy is relocating the logic functions to unused LUTs, the second strategy is to reroute signals to unused interconnects, and the third strategy is to exploit the unused regions of LUTs with spare inputs.

Logic-based sensors for *application specific integrated circuit* (ASIC) designs to measure delay degradation in the circuit due to transistor aging are presented in [23] and [24]. In [23], the sensor is based on two ring oscillators, one as a reference and one under stressed conditions. The outputs of both oscillators are passed to a phase comparator to determine the difference. The difference between their frequencies is used to measure the delay. In [24], the sensor composed of a simple inverter and two tri-state buffers to measure the instability in the output of the critical path during a specific period. The sensor is to be placed at the output of the critical path, if the output is unstable within the desired period, due to aging, the output of the two tristate buffers will be the same, and thus an error can be reported.

Techniques for eliminating the design margins in processor pipelines are presented in [25–29]. Although these techniques are intended to be used to detect violation in design margins caused by power saving techniques, or process variation, they can be used, in principle, for aging detection as well. However, the applicability for FPGA designs is not certain, because they require either non-logic type of resources or delay elements, which have relatively large area impact when ported to FPGA. RAZOR [25, 28], for example (Fig. 4), is a simple and good technique to detect delay faults in ASIC designs. However, mapping it to FPGA resources is difficult, as the placement of its shadow latch and the XOR comparator should be very accurate in order to catch the delayed transitions correctly. Given the constraints in type of available logic resources and routing path delay on current FPGAs, such precise timing cannot be met. Hence such sensors cannot be directly mapped to current FPGAs. In addition to these approaches, there are a large set of sensors and techniques for delay detection in ASIC designs that use resources not available on FPGAs.

The main difference of our work with existing techniques is that it exploits the native resources available on FPGA to design a low-cost aging sensor with adjustable sensitivity. Additionally, we provide a scheme to select the places where the sensor should be placed to have a more reliable monitoring.

## 4 Aging Sensor: Main Idea

### 4.1 Critical Path and Aging

The maximum operational frequency of a circuit is determined according to the delay of the longest combinational path (*critical path*). A guard period is also added to this delay to define the minimum clock period ($T_{critical}$) to ensure correct functionality. This guard period is used to allow the signal to be stable before the flip-flop setup time (Fig. 5) and also to consider process variation. Due to transistor aging, gate delays and in turn path delays are increased, causing transitions to arrive later, and ultimately, an incorrect value can be latched in the flip-flops (as illustrated in Fig. 5).

It should be noted that, in the presence of aging, the critical paths may change over time, that is some noncritical paths at the beginning of FPGA lifetime ($t_0$) may become critical after aging ($t_n$). More details are discussed in Sect. 5.4, where a selection scheme to determine the highly vulnerable paths to aging (aging-critical paths) is presented.

Two main concepts should be considered when designing a sensor to detect these late transitions due to aging or delay faults:

- *Warnability*: The sensor/detector should be able to generate a warning signal when the output of the critical path gets very close to the clock edge before

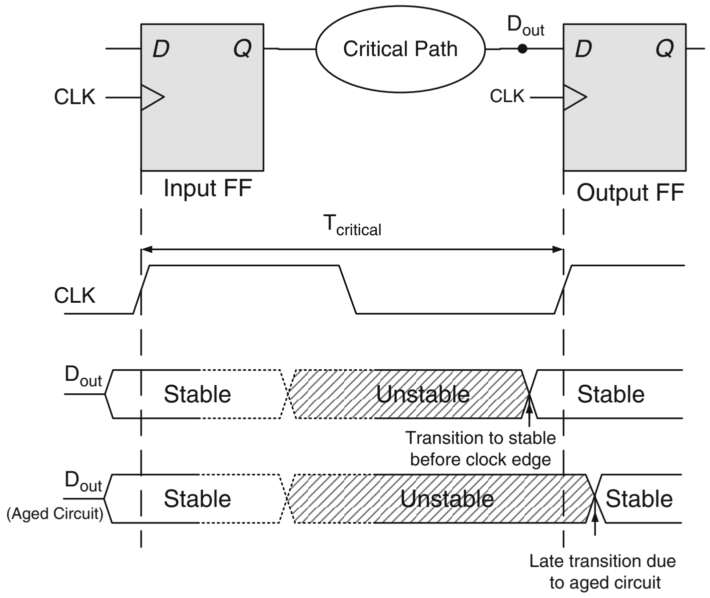


**Fig. 5** Description of the effect of the aged circuit

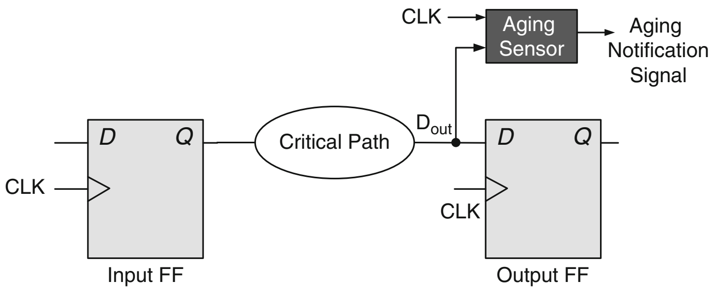


**Fig. 6** The placement of the aging sensor on the critical path

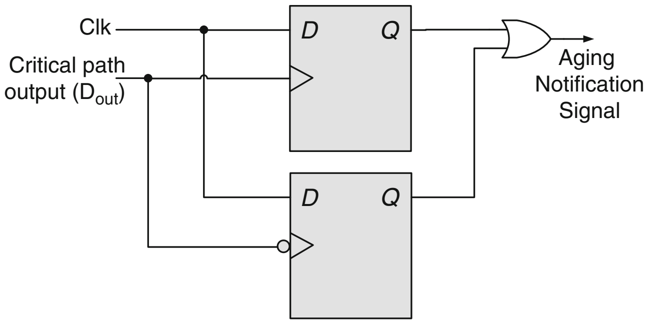


**Fig. 7** Schematic diagram of the proposed aging sensor

exceeding it, which is the case of the aging phenomena that happens gradually. Thus, a suitable action can be taken to mitigate aging (e.g., reducing the operating frequency).

- *Detectability*: When the output of the critical path exceeds the clock edge, a delay fault happens. The sensor/detector should be able to detect this fault and generate an error signal.

## 4.2 *The Proposed Sensor*

The proposed sensor is to be placed in parallel with the aging-critical path output ($D_{out}$), meaning the path that has most (initial delay + aging-induced delay increase), to detect whether the circuit generates late transitions after the clock edge or too close before it, as shown in Fig. 6. The schematic of the sensor is depicted in Fig. 7. The idea is to use the signal on the critical path output itself as a clock input to two different edge-triggered D flip-flops. Assuming that the original design works with the positive-edge transition of the clock, the proposed circuit detects both positive

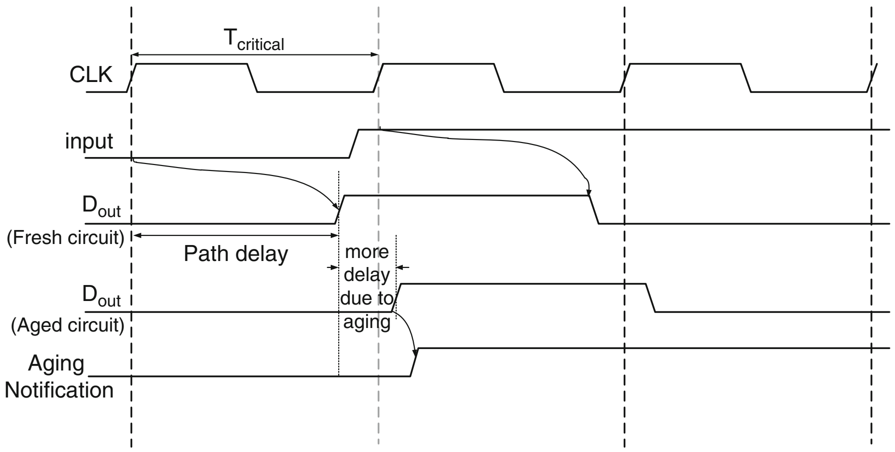


**Fig. 8** Example to show the sensor functionality when the aging happens in the circuit

and negative transitions on the critical path, which happen during the first half (assuming a 50 % duty cycle for the clock) of the next clock cycle (i.e., when the clock level is high). That is why the system clock is fed to the inputs of the D flip-flops. The use of two flip-flops is done because one latches rising transitions and the other one detects falling transitions. When the end point of the critical path makes a transition at the positive level (i.e., the first half) of the (next) clock, it means that the path is aged and makes a late transition.

One can argue that the nonaged (fault-free) path can also make the early transitions during the first half of the clock cycle. This may invalidate the functionality of the sensor as outlined above and cause it to wrongly raise the error signal. Modifications to the sensor to deal with such issues are discussed in Sect. 5, where a narrower *detection window* is generated and used instead of the clock to reduce the detection period, and avoid the false detection of the early transitions that may happen (Fig. 10).

To illustrate the sensor functionality, a chain of odd number of inverters is considered as a basic example for a critical path. The timing diagram in Fig. 8 shows two cases, namely the fault-free (nonaged) circuit (its output is $D_{\text{out}}$), and the aged circuit (with delay fault). In the non-aged case, the input to the inverter chain causes a transition at the output lies in the second half of the clock cycle (clk = 0), and will cause the sensor (Fig. 7) to latch the current state of the clock, that is, “0”, which means no notification of aging. In the second case, the circuit is aged, and the transition in the critical path happens after the clock edge, in the first half of the next clock cycle (clk = 1). This transition causes the sensor to latch the current state of the clock, that is, “1”, and the sensor raises the error signal.

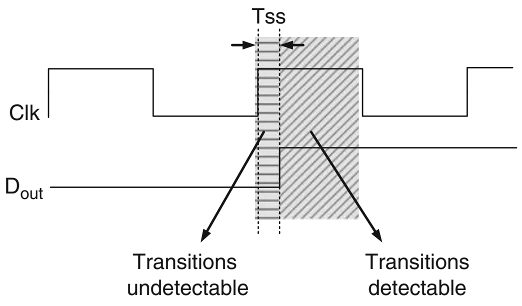


**Fig. 9** The sensitivity of the sensor ($T_{ss}$), negative to the left of the clock edge, and positive to the right

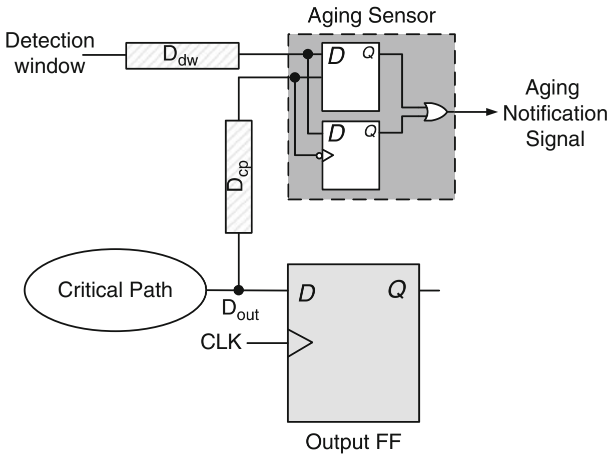


**Fig. 10** The relative delays of the sensor inputs

## *4.3 Sensor Sensitivity Analysis*

The *sensitivity* of the sensor, $T_{ss}$, is defined as the time period from the latching edge of the clock to the earliest time that the path makes a transition after the clock edge and the sensor is able to raise an aging notification signal. In other words, if the clock period is $T_{clk}$, the delay of the critical path must be at least $T_{clk} + T_{ss}$ to be detected by the aging sensor. This means that any aging delay less than $T_{ss}$ would not be detected by the sensor. In case of a warning sensor, $T_{ss}$ is negative, thus the detection can happen before the clock edge (see Fig. 9).

The amount of $T_{ss}$ can be controlled according to the design requirements. To explain how this can be achieved, let's consider Fig. 10 that shows the sensor's flip-flops and the routing delays for its inputs.

The flip-flop requires the transition on its input to be stable before the clock edge, at least in an amount equal to its setup time ($T_{set}$), to latch the input successfully. Another fact that should be considered is that the timing at which the sensor detects must be relative to the timing of the output flip-flop (at which the actual data is latched). Now, if we consider there is no skew between the inputs of the sensor's flip-flops and the inputs of the output flip-flop (Fig. 10), then the sensitivity can be calculated as follows.

$$T_{\text{ss}} = T_{\text{Sensor\_set}} - T_{\text{Output\_set}}, \tag{1}$$

where $T_{\text{Sensor\_set}}$ and $T_{\text{Output\_set}}$ are the setup times of the sensor's flip-flops and the output flip-flop, respectively. If no process variation is considered, the flip-flops will have equal setup times, and as a result, $T_{\text{ss}}$ will be simply zero.

In the real case, there are routing delays that lead to variable amounts of skew. If the two delays are different, the signals will reach the sensor at different timing with respect to the output flip-flop (Fig. 11). Considering these delays, the sensitivity equation (Eq. 1) will become:

$$T_{\text{ss}} = (T_{\text{Sensor\_set}} - T_{\text{Output\_set}}) + (D_{\text{dw}} - D_{\text{cp}}), \tag{2}$$

where $D_{\text{dw}}$ and $D_{\text{cp}}$ are the path delays of the sensor inputs (from the detection window and the critical path) relative to the inputs of the output flip-flop.

Since the setup times cannot be altered, the term $(D_{\text{dw}} - D_{\text{cp}})$ is used to control the amount of the required sensitivity. For an ideal late transition detector, the sensitivity is zero, which means that the late transitions can be detected as soon as they exceed the clock edge. This can be achieved when both terms $(T_{\text{Sensor\_set}} - T_{\text{Output\_set}})$ and $(D_{\text{dw}} - D_{\text{cp}})$ are equal to zero, or when $(T_{\text{Sensor\_set}} - T_{\text{Output\_set}}) = -(D_{\text{dw}} - D_{\text{cp}})$.

To obtain a negative sensitivity (warning sensor), the term $(D_{\text{dw}} - D_{\text{cp}})$ must be negative. That means the amount of the delay from the output of the critical path is larger than that of the clock (Fig. 11).

## 5 Sensor Mapping

We use Xilinx Virtex-6, as one of the state-of-the-art FPGA platforms, for the mapping. As presented in Sect. 4, the sensor uses two different edge-triggered D flip-flops to latch the actual state of the clock. It would be much useful if double-edge-triggered D flip-flops are available on the FPGA, because then, using one flip-flop would be enough to detect both negative and positive transitions, and there will be no need for the OR gate shown in Fig. 7 any more. Unfortunately, double-edge triggered flip-flops are not available on the Virtex-6 series, however, there are similar components that can be used instead, namely, *double-data rate output registers* (ODDR). The ODDR exists near the general purpose input/output pins, and one ODDR is enough to implement the functionality of the proposed sensor. However, because the aging notification signal has to be sent to one of the output pins outside the FPGA, this may cause problems since it might be necessary to

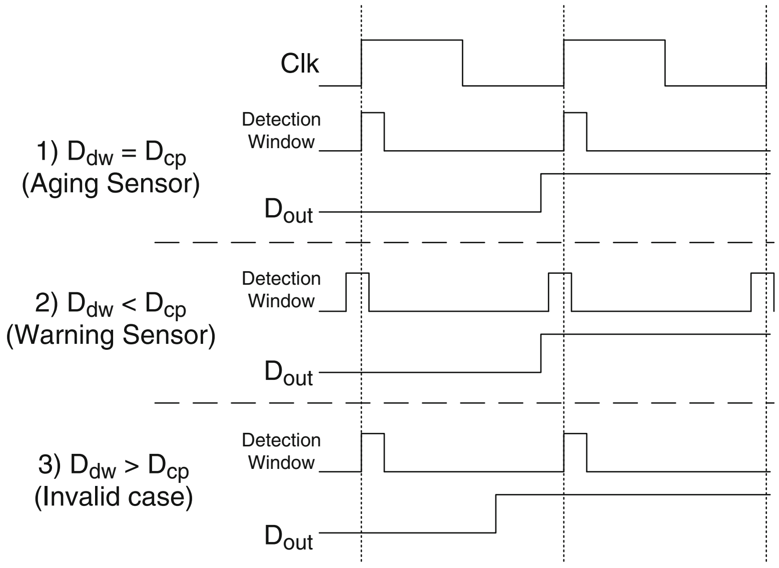

Clk
1) $D_{dw} = D_{cp}$ (Aging Sensor)
Detection Window
$D_{out}$
2) $D_{dw} < D_{cp}$ (Warning Sensor)
Detection Window
$D_{out}$
3) $D_{dw} > D_{cp}$ (Invalid case)
Detection Window
$D_{out}$

**Fig. 11** Different relative values of input delays and the effects of how the sensor sees the inputs

handle and use this signal inside the FPGA. Furthermore, the ODDRs exist only in certain places on the FPGA; so the output of the critical path has to be routed there, which could be very long (through multiple switch matrices and buffers), depending on the original placement of the critical path.

## *5.1 Mapping to Logic Slices*

The other option to map the proposed aging sensor is the normal logic slices that are distributed over the FPGA area. The fact that in each slice of the Virtex-6 FPGA, only one type of clock edge can be defined implies the need to use two different slices to implement the aging sensor: one for the positive-edge D flip-flop and the other one for the negative edge. Although the sensor occupies two slices, it uses the D flip-flops and one LUT, leaving the rest of resources in these slices available for mapping other circuit components.

The basic aging sensor (Fig. 7) can detect any transition happens during the positive level (first half) of the clock cycle. This can generate false notification if an early transition happens in this period in the fault-free operation of the circuit. To alleviate this problem, it is necessary to reduce the window in which the sensor latches the transition. In the basic sensor implementation, this window was generated by the positive level of the clock. Since the clock typically has a 50 % duty cycle, this latching window is 50 % of the clock period. This is an issue for many

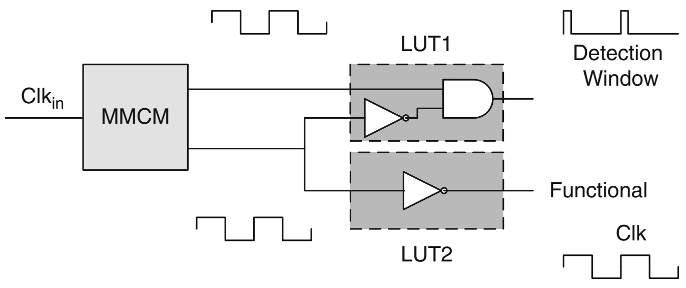


**Fig. 12** The generation of the detection window using the MMCM

functional paths as they make early transitions during the clock cycle. However, if this latching window is reduced such that the monitored path in the nonaged state does not make any transition during this latching window, no false notification can happen.

The Mixed-Mode Clock Managers (MMCM) component in Virtex-6 allows the generation of a controlled-duty-cycle clock. This option fits the need for a flexible aging sensor. By generating a smaller duty cycle period, the latching window can be reduced proportionally. In this case, it is enough to use the controlled duty cycle clock in place of the functional clock in the sensor. However, the MMCM is unable of generating a small duty cycle signal when the functional clock frequency is relatively high (300–600 MHz). This makes this method unsuitable for high-frequency designs. A suitable method for high-frequency designs is introduced in the next section.

## 5.2 *Detection Window Generation*

Using the MMCM, a phase shift version of the clock can be generated, which can be then combined with the original clock using an AND gate to generate the required detection window (Fig. 12). The amount of the phase shift can specify the width of the detection window. It should be noted that the width of the detection window cannot be less than the allowed minimum signal width of the FPGA as described in Sect. 5.3, otherwise, the detection window will be absorbed by the internal buffers and will not reach the flip-flops.

This combination (the original clock with the phase shifted one) must be done before passing the two generated clocks (the functional clock and the generated detection window) to global clock buffers, in order to assure minimum skew. Furthermore, the two clock paths must be balanced using the same amount of components to avoid large delay differences between them. That is the reason why there is a NOT gate on the functional clock path in Fig. 12, so that the two clocks have nearly the same propagation delay of one LUT each.

**Table 1** Minimum allowed pulse widths in different FPGAs

| Simulated device | Minimum allowed pulse width (ps) |
|---|---|
| Spartan-6 LX45 | 240 |
| Virtex-5 LX110t | 369 |
| Virtex-6 LX75t | 450 |

## 5.3 *Glitches in FPGA*

As the proposed sensor uses the data path as a clock source, it may face too many transitions, and hence, the power consumption of the sensor maybe high. Before going further with the analysis of these information, an interesting point must be highlighted. In modern FPGAs, special types of buffers are used for the clocks and the inputs of the flip-flops inside the slices. These buffers allow only pulses with a width greater than a certain amount to be further propagated. Thus, glitches that are very small are internally absorbed by the buffer [30]. Our timing simulations prove that fact as well. Table 1 shows different simulated FPGAs, with the minimum pulse-width that can be propagated through the buffers.

The values in Table 1 are actually defined in the generated post-place-and-route simulation models, under "PATHPULSE => xxx ps" constraint. The possible generated small glitches cannot reach the output flip-flop (the monitored flip-flop, where these glitches are supposed to be latched and cause errors) as they will be absorbed by the buffers, also these glitches cannot reach the sensor's flip-flop. Actually, this is the idea, because the sensor is supposed to generate errors only if the latched data is incorrect; so as long as the output flip-flop does not see these glitches, the sensor would not see them as well. Furthermore, as the possible number of glitches that can reach the sensor clock input within one clock cycle is minimum, its power consumption would be minimum as the results show in Sect. 6.

## 5.4 *Aging Sensor Placement and Calibration*

Choosing the appropriate place (CLB slices) for mapping the sensor with respect to the placement of the *aging-critical paths* (the paths that have the highest post-aging delay) to be monitored, and the number of the sensors for the entire circuit are important issues to be addressed.

### 5.4.1 Selection Scheme of the Paths to be Monitored

The correct choice of where to place the sensors in the circuit plays an important role for achieving a highly reliable monitoring.

To select the paths with the highest post-aging delays, we first need to consider two major aging models, NBTI and HCI, in more details.

The model of the NBTI effect on PMOS transistor's switching delay ($d$) can be estimated and simplified as follows [31, 32]:

$$\Delta d = A_{\mathrm{NBTI}} \times Y^n \times t^n \times d_0, \tag{3}$$

where $d_0$ is the pre-aging delay, $A_{\mathrm{NBTI}}$ is a technology dependent factor, $t$ is the time (FPGA age), and $Y$ is the ratio of the stress time to the total time (duty cycle). $n$ can be 1/6 or 1/4 depending on the fabrication process.

HCI on the other side affects mainly the NMOS transistors. The effect on NMOS transistor's switching delay ($d$) is empirically found and can be simplified as follows [8, 33]:

$$\Delta d = A_{\mathrm{HCI}} \times \alpha \times f \times t^{0.5} \times d_0, \tag{4}$$

where $d_0$ is the pre-aging delay, $A_{\mathrm{HCI}}$ is a technology dependent factor, $t$ is the time, $\alpha$ is the activity factor of the transistor, and $f$ is the frequency.

It can be seen from Eq. 4 that the activity factor and the frequency have a direct effect on the NMOS transistor's switching delay change ($\Delta d$), and from Eq. 3 the effect of the duty cycle ($Y$) on the PMOS transistor's switching delay change. Temperature ($T$) has also exponential effect on the NBTI-induced $\Delta d$ and a piecewise linear effect on the HCI-induced $\Delta d$ [8]. The temperature dependency is implicit in both $A_{\mathrm{HCI}}$ and $A_{\mathrm{NBTI}}$.

As the transistor level implementation of the FPGA is not available, the models in Eqs. 3 and 4 cannot be directly used, and a higher level model is necessary. To obtain a system-level estimation for both HCI and NBTI, the information provided by the FPGA design kit is used. Both the FPGA timing and the power tools provide information at *node* level (a single node can represent a LUT, a path through different switch matrices, an internal signal inside a CLB, etc). In our approach, it is assumed that all the transistors in a single node have the same parameters as the node itself (i.e., same switching activity, same duty cycle, etc). In this way, the models in Eqs. 3 and 4 can be used to calculate the delay change for the entire node $\Delta d$. For the NBTI model, $d_0$ will represent pre-aging delay of the path inside the node at $t_0$ and can be obtained from the timing report for each node. $Y$ is the duty cycle at node inputs, which is estimated by $1 - SP$, where $SP$ is the signal probability and can be obtained using statistical logic simulations for the entire circuit with random inputs. For HCI, $\alpha$ will represent the path activity (number of activation in one cycle), and $f$ is the operational frequency of the node. The multiplication $\alpha \times f$ represents the signal rate and can be obtained from the power report for each node. The selection scheme, for determining the most aging-critical paths in the circuit, can be summarized in the following steps:

1. Using the timing analyzer tool, sort the paths based on their timing slacks in the decreasing order. This way, the critical and near-critical paths are determined.
2. Select top $N$ paths from the list (or all paths that have at most s % time slack, e.g. 5 %).
3. Each path $P_i$ has a delay $d_0^i$ and contains nodes $g_1^i$ to $g_N^i$.

- For each node $g_j^i$, find the activity ratio (signal rate) $R_j^i$ from the power report.
- For each node $g_j^i$, find the duty cycle $Y_j^i = 1 - SP_j^i$ using random logic simulations.

4. For each path $P_i$, obtain:
   - Average activity ratio of its nodes $R_{\text{avg}}^i$.
   - Average duty cycle of its nodes $Y_{\text{avg}}^i$.
5. Re-sort the paths based on post-aging delay $d_0^i + \Delta d^i$:
   - For HCI, find the path $P_j$ with maximum $d_0^j \times R_{\text{avg}}^j$.
   - Assuming $x\%$ maximum HCI delay increase, obtain $K_{\text{HCI}}$ from $K_{\text{HCI}} \times d_0^j \times R_{\text{avg}}^j = x\% \times d_0^j$.
   - Re-sort the paths based on $d_0^i + K_{\text{HCI}} \times d_0^i \times R_{\text{avg}}^i$ .
   - For NBTI, find the path $P_k$ with maximum $d_0^k \times (Y_{\text{avg}}^k)^n$.
   - Assuming $y\%$ maximum NBTI delay increase, obtain $K_{\text{NBTI}}$ from $K_{\text{NBTI}} \times d_0^k \times (Y_{\text{avg}}^k)^n = y\% \times d_0^k$.
   - Re-sort the paths based on $d_0^i + K_{\text{NBTI}} \times d_0^i \times (Y_{\text{avg}}^i)^n$.
6. As the sensor is to be placed at the output of the path, if several paths share the same output node $g_j^i$, keep the one with highest post-aging delay remove the others.
7. Depending on the criticality of the application, and the available space left on the FPGA chip; the number of the paths to be monitored can be determined.
8. Place the sensor circuits at the outputs of selected paths.
9. A calibration to the sensors should then be done to set their sensitivity (negative for warning, and near-zero positive for late transition detector) as detailed below.

### 5.4.2 Sensor Calibration

As mentioned in Sect. 4.3, the sensitivity can be calibrated using the delays at the inputs of the sensor. An option to use programmable delay elements would consume too many resources. Therefore, our proposed approach is to control the routing to specify the amount of the delay in a simple way.

The first knob to calibrate sensitivity is the delay on the detection window path, which can be controlled using a relative location constraint. The locations of the AND and the NOT gates (the LUT to which they are mapped in Fig. 12) can be chosen such that one of them is farther than the other one, with respect to the clock buffers. Thus specifying one of them affects the relative delay of the other one, accordingly. For example, suppose that the names of the two LUTs in Fig. 12 are `MMCM/Detection_window` and `MMCM/Functional_clock`, the relative location constraints for them can then be written as

```
INST "MMCM/Detection_window" U_set="clock_0";
INST "MMCM/Functional_clock" U_set="clock_0";
INST "MMCM/Detection_window" RLOC = X0Y0;
INST "MMCM/Functional_clock" RLOC = X10Y0;
```

which means that the LUT from which the functional clock is passing, is 10 slices farther than the other, with respect to the clock buffers of both, and hence the detection window advances the functional clock. Modifying this distance changes the amount of delay between the two clocks.

The second knob is the relative delay from the critical path output to the sensor. Again, the relative location constraints can be used to determine the place of the sensor relative to the place of the output flip-flop, and thus increasing or decreasing the delay difference between the output flip-flop inputs and the sensor inputs. For example, suppose that the name of the output register in Fig. 10 is `Output_reg`, and the names of the sensor's FFs are `sensor0/pos_edge` and `sensor0/neg_edge`, the relative location constraints can then be written as

```
INST "Output_reg" U_set="sensor0";
INST "sensor0/pos_edge" U_set="sensor0";
INST "sensor0/neg_edge" U_set="sensor0";

INST "Output_reg" RLOC=X0Y0;
INST "sensor0/pos_edge" RLOC=X0Y1;
INST "sensor0/neg_edge" RLOC=X1Y1;
```

By controlling these two knobs, the sensor sensitivity can easily be calibrated using Eq. 2.

### 5.4.3 Pre-used FPGAs

In reconfigurable applications, some other configurations may have been loaded in the FPGA and because of that, the FPGA device is aged based on that usage. When the new configuration is loaded in the FPGA, the effect of pre-aging due to previous configurations need to be considered (in which some FPGA resources are aged with different rates). To handle such cases in our approach, instead of considering $d_0$, as the path delay of "fresh FPGA", one can use $d_0^u$ as the delay of that path in the "used FPGA" (see Eq. 5).

$$d_0^u = d_0 + \Delta d_0, \tag{5}$$

where $\Delta d_0$ is the aging of the path so far. In other words, $d_0$ is updated with the delay increase so far, and the delta delay in future, and hence the post-aging path delay $d$ becomes $d = d_0^u + \Delta d$.

## 6 Experimental Results

In order to evaluate, validate, and analyze the proposed aging sensor, we have performed experimental analysis for representative high-frequency FPGA designs.

### *6.1 FPGA Design Tool Experiments (Simulation Results)*

The simulations are done using Xilinx ISE 12.2, together with Modelsim SE 6.5c for Xilinx Virtex-6 FPGA devices. Post-place-and-route simulation model from ISE is generated for the simulation of each experiment and used together with the generated Standard Delay Format (SDF) file in Modelsim. The maximum delay values for all components were always considered in the following results to reflect the worst-case results. Virtex-6 series FPGAs are chosen for the simulated device since they are one of state-of-the-art FPGA devices from Xilinx fabricated using a 40 nm copper CMOS process technology.

To have a fair analysis of the sensor, we have firstly chosen three circuits with typical operational frequency to reflect real usage of the FPGA. The first circuit is a pipelined 32-bit square root circuit [34]. It contains five pipelined stages and operates at a frequency of about 320 MHz. The second circuit is a pipelined AES encoder [35] that contains 30 stages, and operates at a frequency of about 550 MHz. The third circuit is a self-built typical 8-bit FIR filter with 32 stages, has LUT-based implementation, and operates at a frequency of about 275 MHz. In addition, we have tested the sensor on a set of several circuits from the ITC'99 testbench with different sizes and frequencies.

To assign the sensors to the critical paths in the circuits, a logic-level implementation is necessary, because the behavioral description contains internally generated paths which may not easily be extracted to be monitored. The logic-level descriptions for the circuits were generated using Xilinx ISE synthesis tool. Different number of sensors are placed to the top aging-critical paths of the circuit as described in Sect. 5.4. The sensors are then calibrated as warning sensors. The suitable device size to efficiently fit the requirements of most of the tested circuits is chosen to be XC6VLX75T-FF484. For the AES encoder circuit, XC6VLX240T-FF784 is chosen. To simulate the aging phenomena, the frequency of the circuit is increased gradually and the outputs of critical paths is reported together with the sensors behavior. The sensors were calibrated to work with a sensitivity of almost $-50$ ps. When the critical path transitions fall within 50 ps prior to the clock edge, the sensors generate the aging notification signal. By using the detection window adjustment technique presented in Sect. 5.2, no false notifications happened for the inserted sensors.

The area overhead of the sensors for the tested circuits is reported in Table 2. The area is chosen as an optimization goal in the synthesis phase to have a fair comparison. For power and performance overhead reported in Table 3 the speed

**Table 2** Area overhead for different number of sensors

| Tested circuit | Resource type | Original | With 5 sensors | With 10 sensors | With 20 sensors |
|---|---|---|---|---|---|
| b04 | Slice registers | 62 | 77 (24.19 %) | 90 (45.16 %) | 110 (77.41 %) |
| | Slice LUTs | 108 | 111 (2.77 %) | 113 (4.62 %) | 116 (7.41 %) |
| b05 | Slice registers | 44 | 54 (22.72 %) | 64 (45.45 %) | 84 (90.90 %) |
| | Slice LUTs | 132 | 135 (2.27 %) | 138 (4.55 %) | 141 (6.82 %) |
| b12 | Slice registers | 126 | 136 (7.94 %) | 146 (15.87 %) | 166 (31.75 %) |
| | Slice LUTs | 240 | 243 (1.25 %) | 245 (2.08 %) | 248 (3.33 %) |
| b14 | Slice registers | 165 | 175 (6.06 %) | 185 (12.12 %) | 205 (24.24 %) |
| | Slice LUTs | 782 | 785 (0.38 %) | 787 (0.64 %) | 790 (1.02 %) |
| b17 | Slice registers | 1,334 | 1,344 (0.75 %) | 1,354 (1.50 %) | 1,374 (3.00 %) |
| | Slice LUTs | 5,643 | 5,650 (0.12 %) | 5,652 (0.16 %) | 5,651 (0.14 %) |
| Square root | Slice registers | 924 | 934 (1.08 %) | 944 (2.16 %) | 964 (4.33 %) |
| | Slice LUTs | 997 | 1,009 (1.20 %) | 1,007 (1.00 %) | 1,014 (1.70 %) |
| AES encoder | Slice registers | 7,748 | 7,758 (0.13 %) | 7,768 (0.26 %) | 7,788 (0.52 %) |
| | Slice LUTs | 9,698 | 9,754 (0.58 %) | 9,697 (-0.01 %) | 9,657 (-0.42 %) |
| FIR filter | Slice registers | 499 | 509 (2.00 %) | 519 (4.01 %) | 540 (8.22 %) |
| | Slice LUTs | 962 | 966 (0.42 %) | 968 (0.62 %) | 971 (0.94 %) |

is chosen as an optimization goal. The area overhead of the sensor is very small, as each sensor needs only 2 flip-flops and 1 LUT. The performance overhead is also very small. The power overhead is mainly caused by using the MMCM to generate the detection window. This can be seen from the results in Table 3 where adding extra sensors does not scale the power linearly. Actually, the power can be further reduced by selectively activating the sensors from time to time. The MMCM element is used in all circuits, the original circuits (i.e., without sensors) and with-sensors circuits, to generate the 200+ MHz clock frequency, therefore, the MMCM is not considered in the area comparison. The negative values of area overhead in Table 2 are related to LUTs which have been optimized; however, the area overhead of the registers is always positive.

It needs to be noted that the transistor aging happens at a very large time scale. Therefore, the critical path is not required to be monitored all the time: a periodic (e.g., once every week or month) monitoring of the critical path is enough. Given the runtime reconfigurability of FPGAs, it is possible to turn off the sensor most of the time and only activate it at very infrequent rates. Alternatively, a control signal can be easily asserted to the sensor to enable/disable it. The clock enable (CE) ports of the D flip-flop in the sensor can be used for that purpose. Since the sensor circuitry would be used (switched) much less frequently than the functional path, the aging rate of the sensor circuitry would be multiple times less than the original circuit. Therefore, the aging of the sensor circuitry can be neglected compared to the aging of the original circuit. This periodic activation of the sensor can also reduce the power associated with the sensor circuitry considerably.

**Table 3** Power and performance overhead for different number of sensors

| Tested circuit | Power/Performance | Original | With 5 sensors | With 10 sensors | With 20 sensors |
|---|---|---|---|---|---|
| b04 | Power at 229 MHz | 0.795 W | 0.845 W (6.29 %) | 0.871 W (9.56 %) | 0.876 W (10.19 %) |
| | Performance | 4.332 ns (230 MHz) | 4.347 ns (0.35 %) (230 MHz) | 4.350 ns (0.42 %) (230 MHz) | 4.365 ns (0.76 %) (229 MHz) |
| b05 | Power at 338 MHz | 0.807 W | 0.828 W (2.60 %) | 0.837 W (3.72 %) | 0.850 W (5.33 %) |
| | Performance | 2.877 ns (347 MHz) | 2.942 ns (2.26 %) (340 MHz) | 2.912 ns (1.22 %) (343 MHz) | 2.956 ns (2.75 %) (338 MHz) |
| b12 | Power at 400 MHz | 0.812 W | 0.838 W (3.20 %) | 0.846 W (4.19 %) | 0.868 W (6.70 %) |
| | Performance | 2.458 ns (406 MHz) | 2.491 ns (1.34 %) (401 MHz) | 2.484 ns (1.05 %) (402 MHz) | 2.499 ns (1.67 %) (400 MHz) |
| b14 | Power at 146 MHz | 0.800 W | 0.807 W (0.88 %) | 0.807 W (0.88 %) | 0.811 W (1.38 %) |
| | Performance | 6.588 ns (151 MHz) | 6.835 ns (3.75 %) (146 MHz) | 6.810 ns (3.37 %) (146 MHz) | 6.825 ns (3.60 %) (146 MHz) |
| b17 | Power at 180 MHz | 0.819 W | 0.822 W (0.37 %) | 0.829 W (1.22 %) | 0.829 W (1.22 %) |
| | Performance | 5.513 ns (181 MHz) | 5.527 ns (0.25 %) (180 MHz) | 5.501 ns (−0.21 %) 181 MHz) | 5.511 ns (−0.04 %) (181 MHz) |
| Square root | Power at 310 MHz | 0.951 W | 0.978 W (2.84 %) | 0.976 W (2.62 %) | 0.985 W (3.58 %) |
| | Performance | 3.130 ns (319 MHz) | 3.144 ns (0.44 %) (318 MHz) | 3.173 ns (1.37 %) (315 MHz) | 3.184 ns (1.73 %) (314 MHz) |
| AES encoder | Power at 500 MHz | 3.901 W | 3.900 W (−0.03 %) | 3.928 W (0.69 %) | 3.977 W (1.95 %) |
| | Performance | 1.856 ns (538 MHz) | 1.857 ns (0.05 %) (538 MHz) | 1.896 ns (2.16 %) (527 MHz) | 1.865 ns (0.48 %) (536 MHz) |
| FIR filter | Power at 270 MHz | 1.566 W | 1.581 W (0.96 %) | 1.584 W (1.15 %) | 1.597 W (1.98 %) |
| | Performance | 3.619 ns (276 MHz) | 3.620 ns (0.03 %) (276 MHz) | 3.662 ns (1.19 %) (273 MHz) | 3.633 ns (0.39 %) (275 MHz) |

### 6.2 FPGA Board Experiment (Emulation Results)

To validate the sensor functionality in real-time environment, an XUP-5 board equipped with a Virtex5-LX110t is used. The square root circuit with five attached sensors is tested. The maximum frequency for the mapped circuit is reported as 340 MHz. For the generation of the detection window, two digital clock managers (DCMs) are used, because the MMCM element is not available in Virtex-5. The outputs of the sensors are passed to an OR gate and connected to a LED.

Two input clock frequencies were tested: (1) under the maximum frequency (325 MHz) to test the sensor in normal circuit operation mode, and (2) above the maximum frequency (400 MHz) in order to emulate the aged circuit case. The LED was OFF during the first case, which means that no late transitions were detected, and ON during the second case, which proves that a late transition has been detected.

## 7 Conclusions

Aggressive device downscaling at nanoscale in CMOS technology is one of the main drives for the continuation of Moore's law. State-of-the-art FPGA circuits have taken advantage of most recent CMOS technologies to meet high-performance demands. However, this aggressive downscaling comes at the expense of reduced device predictability, increased parametric variations, and reliability threats. One of the major reliability issues for scaled CMOS technology is transistor aging, mainly due to NBTI and HCI, resulting in performance degradation and delay faults over time.

In this chapter, we have presented the design and mapping of a low-cost logic-level aging sensor for FPGA-based designs. We have taken advantage of FPGA resources to design an efficient aging sensor (controlled to be warning or late-transitions detector) which not only detects transistor aging but can detect erroneous glitches due to intermittent and transient faults. The implementation of such sensors for representative designs shows very low area , performance, and power overhead ($\approx$1.3 % area, $\approx$1.6 % performance, and $\approx$1.5 % power overhead when 10 sensors are placed)

**Acknowledgements** The authors would like to thank both *Saman Kiamehr*, and *Fabian Oboril*, from *Karlsruhe Institute of Technology* (KIT), for the very helpful discussions regarding the selection schemes of the paths to be monitored.

## References

1. Iwai H (2009) Technology roadmap for 22 nm and beyond. In: Electron devices and semiconductor technology, 2nd international workshop on 2009. IEDST '09, pp 1–4
2. Borkar S (2007) Thousand core chips: a technology perspective, In: DAC '07: Proceedings of the 44th annual design automation conference. ACM, New York, NY, USA, pp 746–749
3. Borkar S (2009) Design perspectives on 22 nm cmos and beyond. In: DAC '09: Proceedings of the 46th annual design automation conference. ACM, New York, NY, USA, pp 93–94
4. Borkar S (2006) Tackling variability and reliability challenges. IEEE Des Test Comput 23:520
5. Park SP, Kang K, Roy K (2009) Reliability implications of bias-temperature instability in digital ics. IEEE Des Test 26(6):8–17
6. Bernstein K, Frank DJ, Gattiker AE, Haensch W, Ji BL, Nassif SR, Nowak EJ, Pearson DJ, Rohrer NJ (2006) High-performance CMOS variability in the 65-nm regime and beyond. IBM J Res Dev Adv Silicon Tech 50:433–449
7. Stott EA, Wong JS, Sedcole P, Cheung PY (2010) Degradation in fpgas: measurement and modelling. In: FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on field programmable gate arrays. ACM, New York, NY, USA, pp 229–238
8. Tiwari A, Torrellas J (2008) Facelift: Hiding and slowing down aging in multicores. In: 2008 41st IEEE/ACM international symposium on microarchitecture, 2008. MICRO-41, pp 129–140
9. Zick KM, Hayes JP (2010) On-line sensing for healthier fpga systems. In: FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on field programmable gate arrays. ACM, New York, NY, USA, pp 239–248
10. Mangalagiri P, Bae S, Krishnan R, Xie Y, Narayanan V (2008) Thermal-aware reliability analysis for platform fpgas. In: ICCAD '08: Proceedings of the 2008 IEEE/ACM international conference on computer-aided design. IEEE Press, Piscataway, NJ, USA, pp 722–727
11. Wang W, Reddy V, Krishnan A, Vattikonda R, Krishnan S, Cao Y (2007) Compact modeling and simulation of circuit reliability for 65-nm cmos technology. IEEE Trans Device Mater Reliab 7(4):509–517
12. Zafar S, Kim Y, Narayanan V, Cabral C, Paruchuri V, Doris B, Stathis J, Callegari A, Chudzik M (2006) A comparative study of nbti and pbti (charge trapping) in sio2/hfo2 stacks with fusi, tin, re gates. In: 2006 symposium on VLSI technology, 2006. Digest of technical papers. IEEE, New York, pp 23–25
13. Bhardwaj S, Wang W, Vattikonda R, Cao Y, Vrudhula S (2006) Predictive modeling of the nbti effect for reliable design. In: Custom integrated circuits conference, CICC'06. IEEE, New York, pp 189–192
14. Kim J, Rao R, Mukhopadhyay S, Chuang C (2008) Ring oscillator circuit structures for measurement of isolated nbti/pbti effects. In: IEEE international conference on integrated circuit design and technology and tutorial, ICICDT. IEEE, New York, pp 163–166
15. Stathis JH, Wang M, Zhao K (2010) Reliability of advanced high-k/metal-gate n-FET devices. Microelectronics Reliability, Elsevier, 50(9–11):1199–1202
16. Wang W, Yang S, Bhardwaj S, Vrudhula S, Liu F, Cao Y (2010) The impact of nbti effect on combinational circuit: modeling, simulation, and analysis. IEEE Trans VLSI Syst 18(2):173–183
17. Renesas (2008) Semiconductor reliability handbook. Renesas Electronics Corporation, Japan
18. Rittman D (2005) Nanometer reliability. http://www.tayden.com/publications/Nanometer%20Reliability.pdf
19. Abramovici M, Stroud CE (2003) Bist-based delay-fault testing in fpgas. J Electron Test 19(5):549–558
20. Wong J, Cheung P (2011) Improved delay measurement method in fpga based on transition probability. In: Proceedings of the 19th ACM/SIGDA international symposium on field programmable gate arrays. ACM, New York, pp 163–172
21. Srinivasan S, Mangalagiri P, Xie Y, Vijaykrishnan N, Sarpatwari K (2006) Flaw: Fpga lifetime awareness. In: DAC '06: Proceedings of the 43rd annual design automation conference. ACM, New York, pp 630–635

22. Stott E, Wong J, Cheung P (2010) Degradation analysis and mitigation in fpgas. In: 2010 international conference on field programmable logic and applications (FPL), pp 428–433
23. Keane J, Kim T, Wang X, Kim CH (2010) On-chip reliability monitors for measuring circuit degradation. Microelectronics Reliability, Elsevier, 50(8):1039–1053
24. Omana M, Rossi D, Bosio N, Metra C (2010) Novel low-cost aging sensor. In: CF '10: Proceedings of the 7th ACM international conference on Computing frontiers. ACM, New York, pp 93–94
25. Ernst D, Kim NS, Das S, Pant S, Rao R, Pham T, Ziesler C, Blaauw D, Austin T, Flautner K, and others (2003) Razor: A low-power pipeline based on circuit-level timing speculation. Microarchitecture, 2003, MICRO-36. Proceedings 36th Annual IEEE/ACM International Symposium on, IEEE, pp 7–18
26. Sato T, Kunitake Y (2007) A simple flip-flop circuit for typical-case designs for DFM. Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on, IEEE, pp 539–544
27. Eireiner M, Henzler S, Georgakos G, Berthold J, Schmitt D-Landsiedel (2007) In-situ delay characterization and local supply voltage adjustment for compensation of local parametric variations. IEEE J Solid State Circ 42(7):1583–1592
28. Das S, Tokunaga C, Pant S, Ma W-H, Kalaiselvan S, Lai K, Bull D, Blaauw D (2009) Razorii: in situ error detection and correction for pvt and ser tolerance. IEEE J Solid State Circ 44(1):32–48
29. Bowman K, Tschanz J, Kim NS, Lee J, Wilkerson C, Lu S-L, Karnik T, De V (2009) Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. IEEE J Solid State Circ 44(1):49–63
30. Xilinx synthesis and simulation design guide. http://www.xilinx.com
31. Noda M, Kajihara S, Sato Y, Miyase K, Wen X, Miura Y (2010) On estimation of nbti-induced delay degradation. In: 2010 15th IEEE European Test Symposium (ETS), pp 107–111
32. Wang W, Yang S, Bhardwaj S, Vrudhula S, Liu F, Cao Y (2010) The impact of nbti effect on combinational circuit: modeling, simulation, and analysis. IEEE Trans VLSI Syst 18(2):173–183
33. Takeda E, Suzuki N (1983) An empirical model for device degradation due to hot-carrier injection. IEEE Electron Device Lett 4(4):111–113
34. Angermeier J, Amouri A, Teich J (2009) General methodology for mapping iterative approximation algorithms to adaptive dynamically partially reconfigurable systems. In: International conference on field programmable logic and applications, FPL 2009, pp 302–307
35. Opencores. http://opencores.org. Accessed 2011

# Hardware Design for C-Based Complex Event Processing

**Hiroaki Inoue, Takashi Takenaka, and Masato Motomura**

## 1 Introduction

Recent trends in real-time application domains, such as financial trading, smart city, fraud detection for credit cards, and healthcare, require processing high volumes of time-series events in order to extemporarily extract meaningful information. Complex event processing (CEP) is a new computing paradigm that responds to such application requirements. By definition, CEP generates complex events (i.e., useful information) from a sequence of real-time events and allows events to be both filtered with user-defined patterns and transformed into new data so that applications will be able to quickly and easily handle the events and data. For example, CEP is capable of detecting items which have been shoplifted, and it could raise an alert if a patient had taken an overdose of antibiotics in the past 4 hrs [1].

Many software-based CEP (stream) systems support new event languages as extensions of the structured query language (SQL) used in data-base management systems (DBMSs) [2–11]. SQL is a domain-specific, declarative language widely used in DBMSs, and it enables applications to handle a large volume of data items so as to be efficiently implemented with simple operators. In such systems, an event

H. Inoue (✉)
Green Platform Research Laboratories, NEC Corporation, 1753, Shimonumabe,
Nakahara-ku, Kawasaki 211–8666, Japan
e-mail: h-inoue@ce.jp.nec.com

T. Takenaka
Green Platforms Research Laboratories, NEC Corporation, Nakahara, Kawasaki,
Kanagawa 211-8666, Japan
e-mail: takenaka@aj.jp.nec.com

M. Motomura
Graduate School of Information Science and Technology, Hokkaido University,
Kita-ku, Sapporo 060-0815, Japan
e-mail: motomura@ist.hokudai.ac.jp

```
PATTERN SEQ(News a, Stock+ b[ ])
WHERE   a.type = 'bad' AND
        b[i].symbol = 'NEC'
WITHIN  4 hours
HAVING  b[b.LEN].volume < 80%*b[1].volume
RETURN  parallel_sum(b[ ].volume)
```

**Fig. 1** Example of a SASE+ CEP query [1]

is defined as data with multiple fields (a tuple). The basic functions of an event language include: (1) arithmetic/boolean operations; (2) projection, which discards specified fields from a tuple; (3) selection, which selects tuples whose fields contain at least one match; (4) union, which merges multiple streams into a single stream; (5) partitioning, which classifies a stream contents in terms of fields; (6) windowing, which specifies a range (window) of processed tuples; (7) concatenation, which converts multiple fields into a single tuple; and (8) aggregation, which performs a function (e.g., sum or average) that inputs specified fields of tuples.

The two most powerful functions that such CEP event languages support are (1) regular expressions with Kleene closure (e.g., $*$ or $+$ in regular expressions) and (2) user-defined aggregation functions. A regular expression provides a concise, flexible means for matching sequential events. In particular, Kleene closure enables a finite yet unbounded number of time-series events to be efficiently handled. In addition, user-defined aggregation functions help achieve high throughput with algorithm-oriented applications, such as change-point analysis and cryptography. For example, IBM SPADE [7] is used to allow user-defined aggregation functions written in C++ or Java to be executed with its SQL built-in operators. Thus, the functions enable CEP systems to be widely used in many application domains.

Figure 1 depicts an example of a SQL-based CEP query written in a well-known CEP event language, called SASE+ [8]. This query retrieves the total trading volume of NEC stocks in the 4 h period after some bad news occurred [1].

According to [8], the **PATTERN**, **WHERE** and **WITHIN** clauses form a pattern matching block. The **PATTERN** clause specifies the structure of a pattern to be matched along a **SEQ** construct that specifies an event sequence in a regular expression. In this example, event **Stock b** repeatedly occurs after event **News a** occurs. Here, an array variable **b[]** is declared for each Kleene closure **Stock** component. The **WHERE** clause imposes value-based constraints on the events addressed by the pattern. The **WITHIN** clause further specifies a sliding window over the entire pattern. The evaluation of **PATTERN**, **WHERE**, and **WITHIN** clauses results in a stream of pattern matches; each consists of a unique sequence of events used to match the pattern. The **HAVING** clause further filters each pattern match by applying predicates on the constituent events. Finally, the **RETURN** clause transforms each pattern match into a result event for output. In this example, the **RETURN** clause quickly outputs the sum of stock volumes with user-defined aggregation function **parallel_sum()**, which executes a multi-threaded version of aggregation function **sum()**.

```
PATTERN (A NOTB* C | A NOTC* D | A NOTD* E)
DEFINE
   A    AS ( checkpoint  = 'Staten Island' )
   NOTB AS ( checkpoint != 'Brooklyn'      )
   C    AS ( checkpoint  = 'Queen'         )
   NOTC AS ( checkpoint != 'Queen'         )
   D    AS ( checkpoint  = 'Bronx'         )
   NOTD AS ( checkpoint != 'Bronx'         )
   E    AS ( checkpoint  = 'Manhattan'     )
```

**Fig. 2** Example of a complex event detection query [15]

However, existing software-based CEP systems, which achieve sophisticated event processing with SQL-based declarative languages, suffer from poor event processing performance (at most, 500 Kevents/s) [12]. For example, in financial trading markets, the Options price reporting authority (OPRA) has, in fact, announced that event traffic will reach 6.537 Mevents/s (1.57 Gbps) by July, 2012 [13]. Typically, events are sent in small user datagram packet (UDP) packets in order to reduce the latency required for event processing. Once the arrival rate of the events exceeds a certain threshold, the systems are unable to sustain their event processing since UDP packets begin to be dropped [14].

One promising approach would seem to be use of reconfigurable hardware, such as field-programmable gate arrays (FPGAs), in order to accelerate event processing. The authors of [15] have proposed an epoch-making, FPGA-based CEP system that employs an in-house SQL compiler and have achieved 1 Gbps event processing performance. The FPGA-based CEP system supports regular expression-based complex event detection, unlike traditional CEP languages. Figure 2 depicts an example of a complex event detection query. According to [15], the New York marathon is taken as an example. The runners need to pass an electronic checkpoint in each of the five boroughs: Staten Island (**A**), Brooklyn (**B**), Queens (**C**), the Bronx (**D**) and Manhattan (**E**). While there is nothing wrong with a runner passing any single of the checkpoints, an incorrect order of passing them may indicate cheating. This query is used to describe the complex event where a runner reached one of the checkpoints **C**, **D**, or **E** (from start point **A**), but has not passed the respective predecessor **B**, **C**, or **D**.

This query consists of the **PATTERN** clause and the **DEFINE** clause. In the **PATTERN** clause, the complex event is specified using predicate-based regular expressions with Kleene closure. The predicates are defined in the subsequent **DEFINE** clause. Observe that the absence of checkpoint readings (e.g., as $(\mathbf{A}|\mathbf{C}|\mathbf{D}|\mathbf{E})^*$) can be described in a more readable way by using negation (i.e., **NOTB**, **NOTC**, and **NOTD**). While the FPGA-based CEP system achieves efficient hardware acceleration for complex event detection, it restricts the language specifications of SASE+ only to the **PATTERN** and **WHERE** clauses (i.e., regular expressions with Kleene clo*sure) without any aggregation functions* due to the language's complexity required for building hardware.

In this chapter, we report our hardware-accelerated CEP system, which uses a novel hardware-friendly C-based event language [16]. This C-based approach offers two major benefits: (1) high-throughput regular expressions with Kleene closure and (2) support of various aggregation functions. This is possible because recent advances in high-level synthesis industry tools [17, 18] allow source codes written in C to be directly converted to fully-optimized hardware description language (HDL) source codes. In addition, on the basis of the hardware-friendly C-based event language, we make it possible to support SQL interfaces in the same flexibility as software-based CEP systems since the C-based event language itself has a high descriptive ability to implement other SQL functions.

Major contributions of our work include achievement of the following design objectives:

- *A novel hardware-friendly C-based event language*: We have newly defined a C-based event language which includes new regular expression syntax with Kleene closure in addition to traditional C syntax, using typical SQL-based event languages for reference [8, 19], in order to help detect matched events in chronological order. The resulting event language is able to support a variety of aggregation functions, including those required for our financial trading example (see Sect. 2).
- *An efficient logic construction method*: We have designed a pipelined method which enables both regular expression syntax with Kleene closure and (user-defined) aggregation functions to be efficiently mapped to hardware. The method newly adds data paths required for aggregation functions to control paths used for regular expressions with Kleene closure as an extension of a traditional logic construction method, known as the Sidhu and Prasanna method [20] (see Sect. 3.2).
- *Remarkable speed-ups over CPU software*: We have confirmed that our FPGA-based CEP system is applicable to an actual 20 Gbps FPGA-based network interface card (NIC) by applying it to an example of financial trading (see Sect. 2). On the FPGA NIC, we have achieved 12.3 times better performance than does CPU software in the example of financial trading.

The remainder of this chapter is structured as follows: Sect. 2 depicts our motivating example, Sect. 3 introduces our C-based framework, Sect. 4 describes our CEP system, Sect. 5 presents the results of our evaluation, Sect. 6 illustrates related work and Sect. 7 summarizes our work.

## 2 Motivating Example

One of our target applications is financial trading, which requires real-time processing with respect to various stock prices obtained from stock exchanges. Figure 3 shows a simple stock price analysis. Here, we use the opening prices of NEC

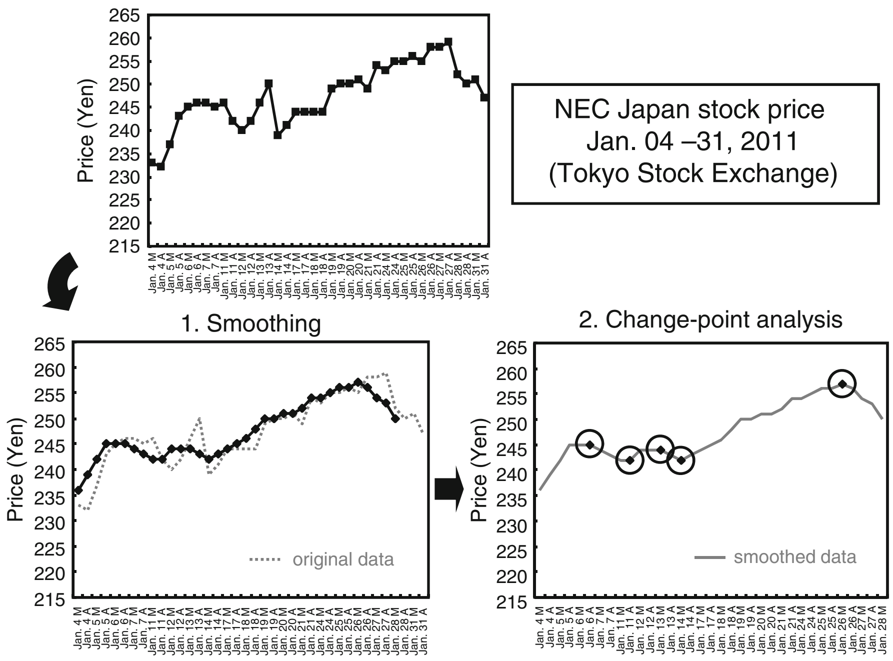


**Fig. 3** Motivating example—stock price analysis

Japan stock in the morning and afternoon sessions from January 4th (the beginning of this year's trading) to 31st in 2011; full real-time stock movements for each day were unavailable. As the authors of [21] have shown, however, market prices exhibit a fractal structure (i.e., tend to exhibit the same behavior) regardless of the measurement time-scale. This suggests that our analysis, while non-real-time, is sufficiently significant for our purposes.

When using data such as that shown in the figure, a smoothing operation, such as finite-impulse-response (FIR) filtering, is often first conducted in order to eliminate fluctuations in stock prices. After that, a change-point analysis, such as data mining, is often performed to the smoothed line in order to extract useful information in an algorithmic way. The extracted information can be used to conduct an advanced analysis. In this example, we have used a moving average of four stock prices as a smoothing operation and have used detection of local maximums and minimums as a change-point analysis. This change-point analysis requires a regular expression with Kleene closure. In addition, both operations clearly require user-defined aggregation functions. This means that we need to use procedural languages, such as C or Java, for application to CEP systems.

## 3 C-Based CEP Language

This section describes our CEP framework, which is based on the use of a C-based event language. The idea behind our language is the use of regular expressions. Then, each element of a regular expression is a function written in C. Concepts behind the **PATTERN** clause of SASE+ [1, 8] and the **MATCH-RECOGNIZE** clause of the current ANSI draft [19] served as references in the development of this idea. As a result, our C-based CEP framework supports the two most difficult CEP functions: regular expressions with Kleene closure and (user-defined) aggregation functions. In addition, in order to maintain compatibility with software-based CEP systems, our CEP framework allows SQL interfaces to be implemented on the event language because its descriptive ability has high affinity with SQL-based frameworks.

### *3.1 Language Overview*

Figure 4 presents an overview of our C-based event language. The language has three user-defined data structures: **evin_t**, **evout_t**, and **evarg_t**. While structure **evin_t** contains tuples of an input event, structure **evout_t** contains tuples of an event that is output when a specified regular expression matches a sequence of events. Structure **evarg_t** contains data shared among functions. Since arrays can be used in structure **evarg_t**, it can use array elements to store values contained in individual streams, for the purpose of stream partitioning.

**(1) `EVENT_RULE` macro:**

```
EVENT_RULE (<rname>, <rule>, <initial>, <final>)
```

| | |
|---|---|
| `<rname>` | **The name of this event rule** |
| `<rule>` | **Event regular expression** |
| `<initial>` | **Initial statement** |
| `<final>` | **Final statement** |

**(2) Regular expression definition:**

| | |
|---|---|
| `<fname>` | **Function; the return value is *boolean*** |
| `(r)` | **Grouping; bypass default binding** |
| $r_1$ $r_2$ | **Sequence; $r_1$ followed by $r_2$** |
| $r_1$ \| $r_2$ | **Choice; either $r_1$ or $r_2$** |
| `r+` | **Closure; one or more repetitions of r** |

**(3) Function definition:**

```
bool <fname> ( evin_t ev, evarg_t *arg ){}
```

| | |
|---|---|
| `<fname>` | **Function name** |
| `ev:` | **Input event** |
| `arg:` | **Data shared among functions** |

**Fig. 4** C-based event language

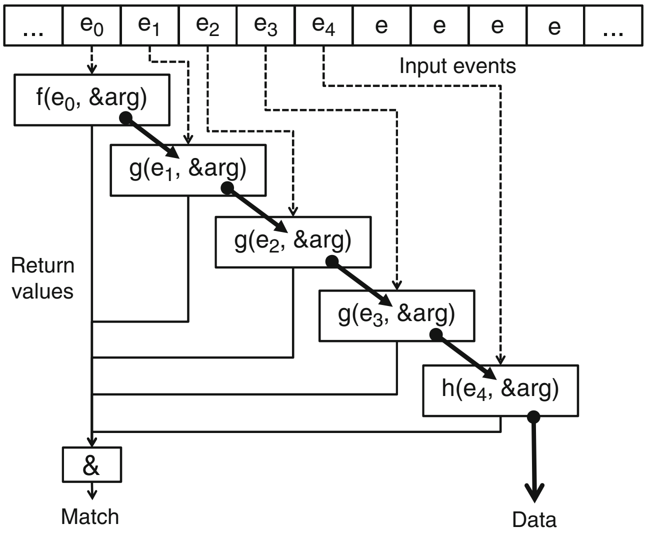


**Fig. 5** Operation overview of our CEP

A user simply writes an **EVENT_RULE** macro that has four arguments: < **rname** >, < **rule** >, < **initial** > and < **final** >. < **rname** > indicates the name of the event rule. < **initial** > is an initial statement that sets initial values in an argument used in the first function used (a variable **evarg** of structure **evarg_t**). < **final** > is a final statement that outputs the results obtained in the last function used (a variable **evarg** of structure **evarg_t**) as an output event (a variable **evout** of **evout_t**). < **rule** > describes a regular expression to be used for event processing (i.e., the **PATTERN** clause). Like traditional regular expression syntaxes, ours has four basic rules: grouping (), sequence $\mathbf{r}_1\mathbf{r}_2$, choice $\mathbf{r}_1|\mathbf{r}_2$, and (Kleene) closure $\mathbf{r}+$. Here, zero ore more repetitions of **r**, known as $\mathbf{r}^*$, can be expressed with a combination of a choice and a closure (e.g., $\mathbf{ab}^* = \mathbf{a}|\mathbf{ab}+$).

The elements of regular expressions are C-based functions, referred to as < **fname** >, whose return values are Boolean (i.e., predicates for the **WHERE** and **HAVING** clauses). Each function has two arguments: an input event (**ev**) and a pointer to shared data (**arg**). It uses the two to return true when a specified condition is a match. An input event is given to all functions at the same time. When all return values of functions invoked along a regular expression are true, the entire regular expression is a match. The most important point here is that changed **arg** values will propagate along a specified regular expression, with each function (starting with the second function) in the chain receiving values that have been modified by the previous function (i.e., for the **HAVING** clause and aggregation functions).

Figure 5 shows an example operation of a regular expression whose elements are three functions: **f()**, **g()**, and **h()**.In this example, the regular expression is defined as **f()g()+h()**. As shown in the figure, input events are sequentially given to each function. Here, if all return values of functions (e.g., **f()g()g() g()h()**) are true, the

```
typedef struct { uint32_t price; uint32_t time } evin_t;
typedef evin_t  evout_t;
typedef evout_t evarg_t;

EVENT_RULE("Smoothing", "F0 F1 F1 F2",
           "evarg.price = evarg.time = 0;",
           "evout.price = evarg.price;
            evout.time  = evarg.time;");
bool                           bool
F0(evin_t ev, evarg_t *arg)    F2(evin_t ev, evarg_t *arg)
{                              {
  arg->price = ev.price;         arg->price =
  arg->time  = ev.time;          (arg->price + ev.price)/4;
  return 1                       return 1
}                              }

bool
F1(evin_t ev, evarg_t *arg)
{
  arg->price += ev.price;
  return 1
}
```

**Fig. 6** Example source code in smoothing

regular expression is a match. Then, the final data (**arg**) are obtained from values modified along the function sequence **f()g()g()g()h()**. Although, for simplicity, we explain here only single-argument functions, multiple-arguments functions may be used in the same way.

Figure 6 shows an example source code in a smoothing operation. In this example, input event **ev**, output event **evout**, and argument **arg** each contain two data items: **time** and **price**. Here, we use a moving average of four stock prices as a smoothing operation. Function **F0** stores the time and price of the current event in argument **arg**, function **F1** adds the price of the current event to that of the previous event, and function **F2** calculates a moving average, dividing the sum of the prices of the current event and the previous event by four. Here, function **F1** is used twice.

Figure 7 shows an example source code in a change-point analysis operation. In this example, input event **ev** and output event **evout** each contain two data items: **time** and **price**. Argument **arg** contains three data items: **trend** (for the direction of the smoothed polygonal line), **last_time**, and **last_price** (i.e., time and price data for the previous event). Here, we use detection of local maximums and minimums for our change-point analysis. Function **C0** stores the current event to argument **arg** when the price of the event is equal or greater than that of the previous event. Function **C1** does the same operation when the price of the current event is equal or less than that of the previous event. The closure of either function **C0** or **C1** will result in repeated invoking of corresponding functions. Function **C2** either outputs a local maximum, when the price of a current event is less than that of the previous event after function **C0** has been invoked, or outputs a local minimum, when the price of a current event is greater than that of the previous event after function **C1** has been invoked.

```
typedef struct { uint32_t price; uint32_t time } evin_t;
typedef evin_t  evout_t;
typedef struct {uint32_t last_price; uint32_t last_time;
  int trend; } evarg_t;

EVENT_RULE("ChangePoint", "(C0|C1)+ C2",
            "evarg.last_price = evarg.last_time = 0;
             evarg.trend = TREND_NOP",
            "evout.price = evarg.last_price;
             evout.time  = evarg.last_time;");

bool
C0(evin_t ev, evarg_t *arg)
{
  boot ret = 0;
  if ( arg->last_price < ev.price ||
     ( arg->trend == TREND_UP && arg->last_price == ev.price )){
    arg->trend = TREND_UP;
    ret = 1;
  }
  arg->last_price = ev.price;
  arg->last_time  = ev.time;
  return ret;
}

bool
C1(evin_t ev, evarg_t *arg)
{
  boot ret = 0;
  if ( arg->last_price > ev.price ||
     ( arg->trend == TREND_DOWN && arg->last_price == ev.price )){
    arg->trend = TREND_DOWN;
    ret = 1;
  }
  arg->last_price = ev.price;
  arg->last_time  = ev.time;
  return ret;
}

bool
C2(evin_t ev, evarg_t *arg)
{
  if ( arg->trend == TREND_UP   && arg->last_price > ev.price ) {
    return 1;
  } else if
     ( arg->trend == TREND_DOWN && arg->last_price < ev.price ) {
    return 1;
  }
  return 0;
}
```

**Fig. 7** Example source code in change-point analysis

## *3.2 Logic Construction*

Our novel method makes it possible for the event processing logic described in our event language to be systematically constructed with four rules (see Fig. 8). *The idea behind it is the logic synthesis of regular expressions with data paths.* In other words, while our method constructs a non-deterministic finite automaton (NFA) structure for return values of defined functions by using methods based on those in [20] (in addition to [22–29]), it correctly connects arguments among defined functions on the

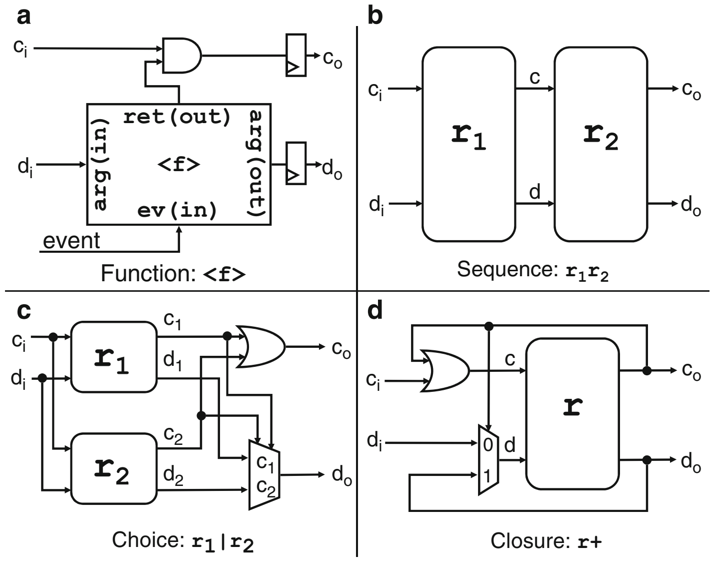


**Fig. 8** Logic construction method

NFA structure in a pipelined way. Grouping requires no construction rules since it changes only the operator bindings. It should be noted that it is difficult to implement high-throughput regular expressions used for events with a simple C library (i.e., without our logic construction method) because of its algorithm complexity.

Function $< \mathbf{f} >$ (see Fig. 8a) is simply replaced by a synthesizable function used in a behavioral description language. A logical AND operation is performed on the return value of function $< \mathbf{f} >$ and the value $c_i$ of the previous control path, yielding, via a flip-flop, the value $c_o$ for the next control path. Function $< \mathbf{f} >$ inputs an event and the value $d_i$ of the previous data path, and outputs, via a flip-flop, calculated data as the value $d_o$ for the next data path. For simplicity, we here explain the construction rule for only one-cycle operations. We have supported pipelined multi-cycle operations with two extensions: valid signals which indicate that control and data path signals are valid at a cycle, and event queues, each of which is associated with a function in order to both store input events and provide an input event to the function when a valid signal associated with the previous control and data path signals is asserted.

Sequence $\mathbf{r}_1\mathbf{r}_2$ (see Fig. 8b) simply connects (1) the values $c_i$ and $d_i$ of the previous control and data paths to the corresponding input values of rule $\mathbf{r}_1$, (2) two output values c and d of rule $\mathbf{r}_1$ to the corresponding input values of rule $\mathbf{r}_2$, and (3) the two output values of rule $\mathbf{r}_2$ to the values $c_o$ and $d_o$ for the next control and data paths. This construction rule is equivalent to an assignment statement in C.

Choice $\mathbf{r}_1|\mathbf{r}_2$ (see Fig. 8c) connects the values $c_i$ and $d_i$ of the previous control and data paths to the corresponding input values in rules $\mathbf{r}_1$ and $\mathbf{r}_2$. A logical OR

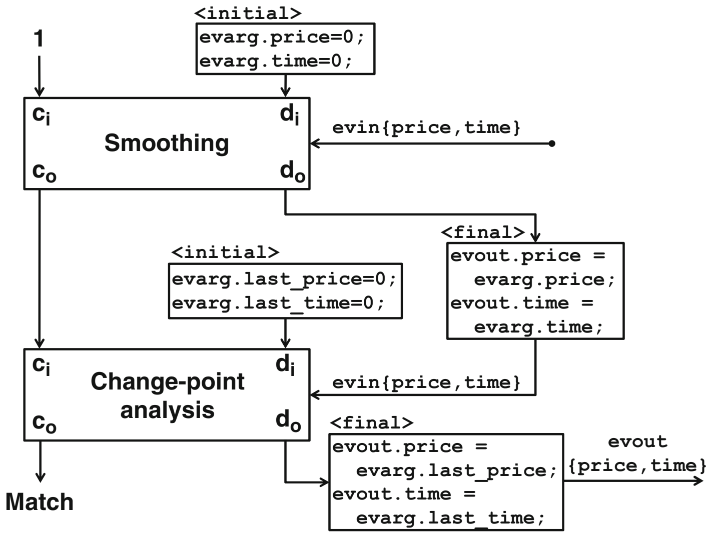


**Fig. 9** Logic overview of smoothing and change-point analysis

operation on the value $c_1$ of the control path of rule $\mathbf{r}_1$ and the value $c_2$ of the control path of rule $\mathbf{r}_2$ is performed, yielding the value $c_o$ for the next control path. The value $d_o$ for the next data path is selected from either $d_1$ or $d_2$ on the basis of values $c_1$ or $c_2$ of the control paths of rules $\mathbf{r}_1$ and $\mathbf{r}_2$. Although the number of selection options is considerable, our current option is as follows: if $c_2$ is true, $d_o$ is $d_2$. Otherwise, $d_o$ is $d_1$. This construction rule is equivalent to an **if** statement in C.

Closure **r+** (see Fig. 8d) connects two output values of rule **r** to the values $c_o$ and $d_o$ for the next control and data paths. Although the number of repetition options is considerable, our repetition option employs the longest-match rule: if $c_o$ is true, the input values c and d of rule **r** are connected to the values $c_o$ and $d_o$ for the next control and data paths. Otherwise, they are connected to the values $c_i$ and $d_i$ of the previous control and data paths. This construction rule is equivalent to a **while** statement in C.

Figure 9 shows the system overview of smoothing and change-point analysis operations in accord with the **EVENT_RULE** macros shown in Figs. 7 and 8. In the smoothing operation, the input value $c_i$ of the control path is 1. Then, the output value $c_o$ of the control path is directly connected to the input value $c_i$ of the control path in the change-point analysis operation. The input values $d_i$ of the data paths in both operations are obtained from the initial statements < **initial** > described in Figs. 7 and 8. The output value $d_o$ of the data path in the smooth operation is given to the change-point analysis operation as an input event obtained through the final statement < **final** > described in Fig. 6. Finally, in the change-point analysis operation, the output value $c_o$ of the control path indicates a match, and the output value $d_o$ of the data path is a local maximum/minimum as an output event obtained

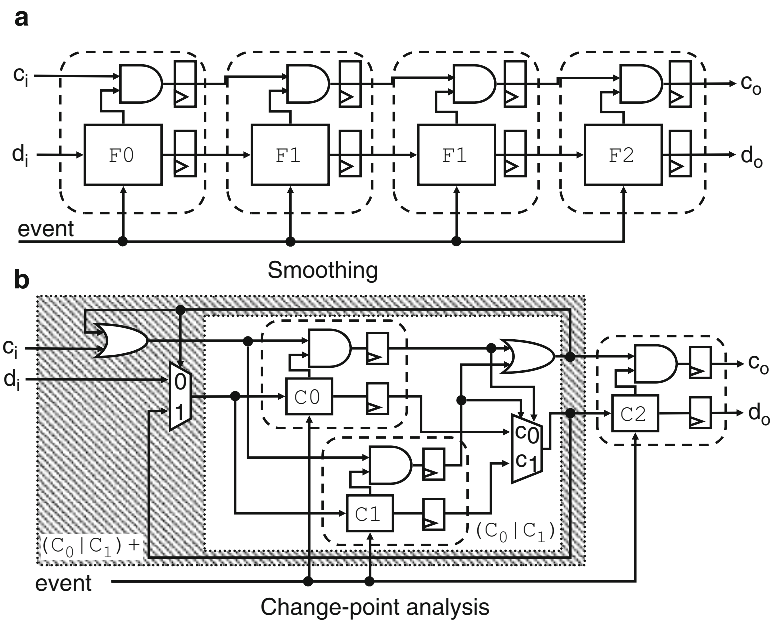


**Fig. 10** Logic examples for smoothing and change-point analysis

through the final statement < **final** > described in Fig. 7. In this way, two operations are subsequently invoked.

Figure 10 illustrates the synthesized logic of smoothing and change-point analysis operations in accord with the source codes shown in Figs. 6 and 7. In the smoothing operation (see Fig. 10a), functions **F0**, **F1** and **F2** are simply connected in series, using the sequence construction rule. In the change-point analysis operation (see Fig. 10b), functions **C0**, **C1** and **C2** are connected with the sequence, choice, and closure construction rules. In both examples, the initial value $c_i$ of the control path is always true, and the initial value $d_i$ of the data path is assigned by initial statement < **initial** >. Moreover, when $c_o$ is true, the final value $d_o$ of the data path is assigned to an output event **evout** by final statement < **final** >. In this way, our new construction method allows regular expressions written in our C-based event language for efficiently mapping to hardware.

## 3.3 Design Flow

In our framework, C functions themselves are directly converted by high-level synthesis industry tools. Recent advance in high-level synthesis puts synthesizable C codes into a practical use. In order to synthesize C functions, we have used NEC CyberWorkBench [17, 18] that supports both extended American national standards

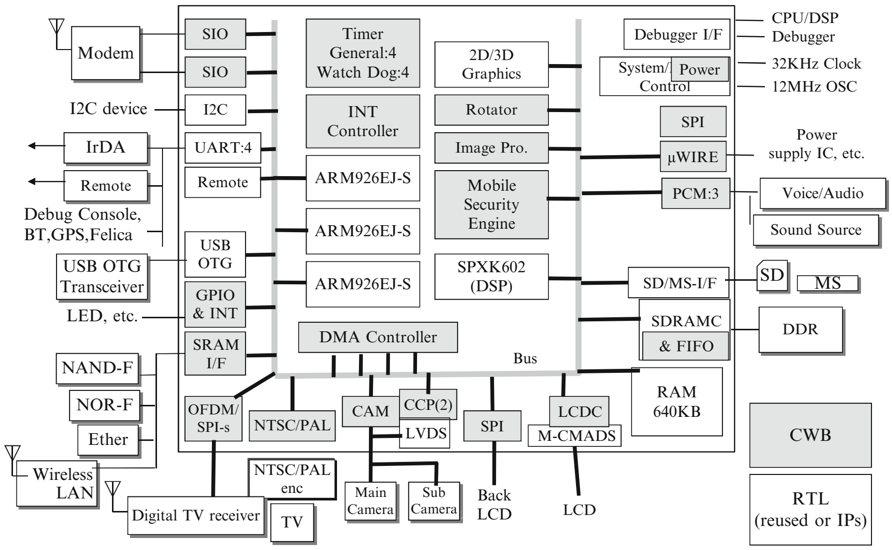
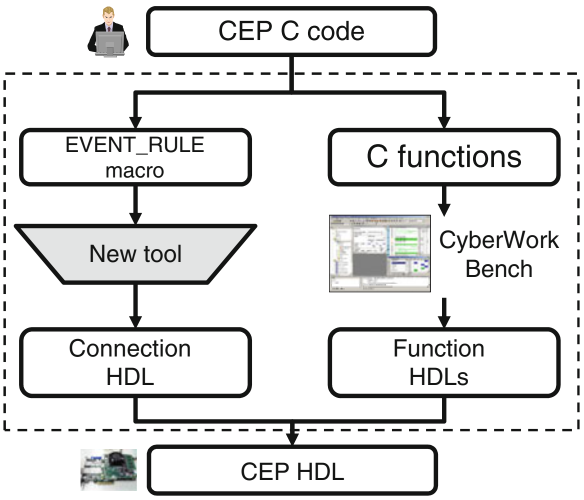


**Fig. 11** Design example of NEC CyberWorkBench

institute (ANSI) C and SystemC. High-level synthesis tools have been practically used for many commercial products. Figure 11 shows NEC 3G mobile phone application chip [30], called MP211, as a design example of NEC CyberWorkBench. In the figure, the gray boxes are designed with NEC CyberWorkBench.

CyberWorkBench supports almost all control structures (e.g., bounded-/unbounded-loops, conditionals, break/continue, and functions), and many data types (e.g., multi-dimensional arrays, pointers, struct, classes, templates, static variables, and typedefs). In addition, CyberWorkBench has three types of extensions to describe realistic hardware description. The first type of extensions includes hardware-specific descriptions, such as input/output port declarations, bit-widths of variables, fixed point arithmetic, synchronization with a special wait function, clocking with clock-boundary symbols, and interrupt/trap operations with always construct meaning statements. The second type of extensions includes hardware-oriented operations, such as bit slice, reduction and/or, and structural components (e.g., multiplexers). The third type of extensions includes optimization descriptions (pragmas), such as register/wire assignments for variables, memory/register file assignments for arrays, and data-transfer type assignments (e.g., wire, registers, latches, or tri-state gates), synchronous/asynchronous sets of initial register values, specifications of gated clocked registers, data initiation interval for pipelined loops, partial-loop-unrolling numbers, and manual binding indicators. Behavioral IP libraries, called Cyberware, such as encryption, decryption, and floating point arithmetic are also included. CyberWorkBench, however, has some restrictions on the ANSI-C language. Dynamic behaviors, such as dynamic allocation, and recursion, are currently unsupported.

**Fig. 12** Overview of our C-based CEP design flow



We have designed a new tool which incorporates our logic construction method (see Sect. 3.2) in order to convert **EVENT_RULE** macros to HDL codes. The new tool will be integrated into a high-level synthesis tool, such as CyberWorkBench. Figure 12 summarizes the overview of our C-based CEP design flow. A customer first writes a CEP query in our C-based event language. As described in Sect. 3.1, this query includes both an **EVENT_RULE** macro and C functions. A high-level synthesis tool enables C functions themselves, such as functions **F0, F1, F2, C0, C1,** and **C2** in Figs. 7 and 8, to be directly translated into HDL codes. On the other hand, the new tool converts the **EVENT_RULE** macro to HDL codes. Finally, the customer obtains the whole CEP HDL codes, combining the HDL codes of C functions and the one of an **EVENT_RULE** macro (i.e., connection along a regular expression).

## 4 Hardware-Accelerated CEP System

This section describes our hardware-accelerated CEP system. In the system, we use FPGA-based NICs in order to execute CEP queries generated by our new design methodology (see Sect. 3) on them.

### *4.1 System Overview*

Figure 13 presents an overview of our hardware-accelerated CEP system, which processes events in two steps: (1) pre-processing a large number of sequential events on an FPGA-based NIC, and (2) main-processing a small amount of useful information

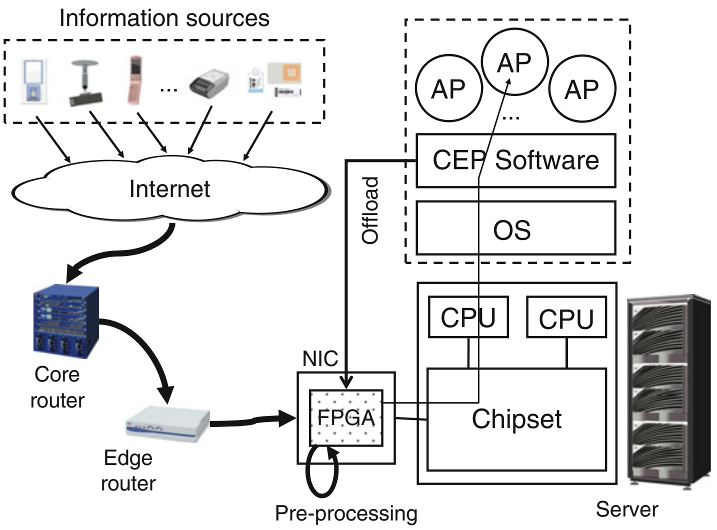


**Fig. 13** Overview of hardware-accelerated CEP system; AP stands for application

on CEP software. A server executes real-time applications with conventional CEP software. This software configures an FPGA-based NIC of the server in accord with application demands in order to receive a small amount of useful information. When a large number of events generated by a wide variety of information sources reach the server via core and edge routers, the NIC conducts configured event processing (i.e., executes queries) on them. If a matched condition is satisfied, the NIC sends an output event to the CEP software. The CEP software then performs main-processing on the basis of the filtered events. In this way, our CEP system makes possible both a high level of real-time application throughput and a large number of real-time applications to be executed on it since pre-processing on the NIC hardware makes it possible to avoid heavy CPU loads.

## *4.2 FPGA-Based NIC*

Figure 14 presents a design overview of our event processing adapter in an FPGA-based NIC. This NIC allows packets to be processed via Atlantic interface [31], a high-performance standard interface suitable for packet processing. The interface provides (1) (for packet sending) five control signals, 128 (16 byte) data signals, and four byte-enable signals, and (2) (for packet receiving) six control signals, 128 (16 byte) data signals, and four byte-enable signals. This means that a packet will be fragmented into data chunks of 16 bytes, and the adapter will receive and/or send up to 16 bytes every cycle.

The current design of our event processing adapter consists of five modules: the first, second, and third header checkers, event processing logic, and an event sender.

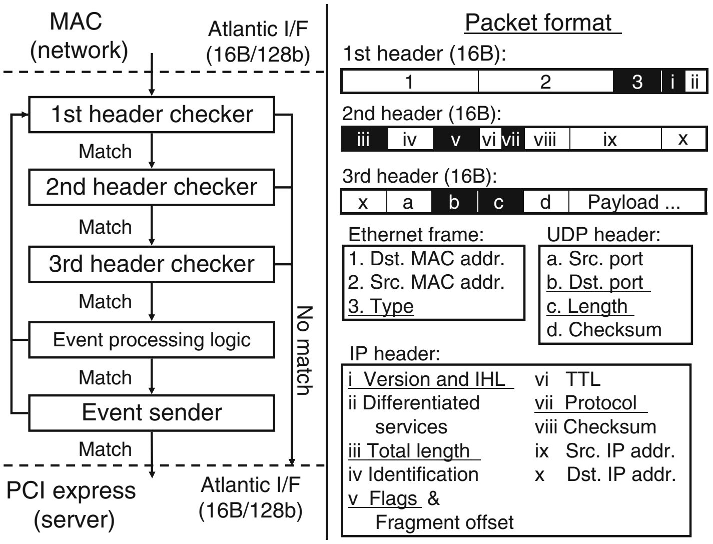


**Fig. 14** Inside our FPGA-based NIC

For simplicity, each UDP packet in our CEP system includes only a single event (i.e., there is no packet fragmentation). This means that the sum of an input event size and the IP/UDP header sizes needs to be less than the maximum transmission unit (MTU) size.

The three header checkers check corresponding header fields of a received packet (e.g., the black boxes in Fig. 14) via Atlantic interface. The first header checker checks whether the *type* field of an Ethernet frame is 0x0800 (IPv4) and whether the *version* and *internet header length (IHL)* fields of an IP header are, respectively, 0x4 and 0x5 (IPv4 and 20-byte header). Next, the second header checker checks at the next cycle whether the *total length* and *protocol* fields of the IP header are, respectively, the sum of IP/UDP header sizes (28 bytes) and an input event size, and 0x11 (UDP). This checker may confirm whether the *flags* field of the IP header indicates "*Don't fragment*". The third header checker then checks at the next cycle whether the *destination port* and *length* fields of a UDP packet are, respectively, the port used by our CEP system and the sum of the UDP header size (8 bytes) and an input event size. When any of the three checkers detects an unmatched field, the packet is simply forwarded to a server.

The event processing logic constructed from our event language inputs the entire payload of a UDP packet as one event. It may connect multiple operations in series or in parallel (e.g., serial connection of smoothing and change-point analysis operations in our motivating example). If the logic detects that a specified regular expression has been matched, an event sender sends a server a UDP packet that includes an output event, calculating a new IP checksum for the packet. In this way, our event processing adapter effectively handles events wrapped in UDP packets.

## 5 Evaluation

Figure 15 shows our target 20 Gbps FPGA-based NIC; Table 1 summarizes its specifications. To the best of our knowledge, this is the first report of an over-10 Gbps evaluation environment. We use an in-house compiler that converts codes written in our event language to synthesizable C codes, which we then compile to HDL codes with NEC CyberWorkBench [17, 18].

### *5.1 Implementation*

We have implemented both smoothing and change-point analysis operations with our event processing adapter on our target NIC. For verification, we have used the NEC Japan stock information (see Sect. 2) as events. The date of each event is replaced by a sequence number in order to continuously input events at the cycle

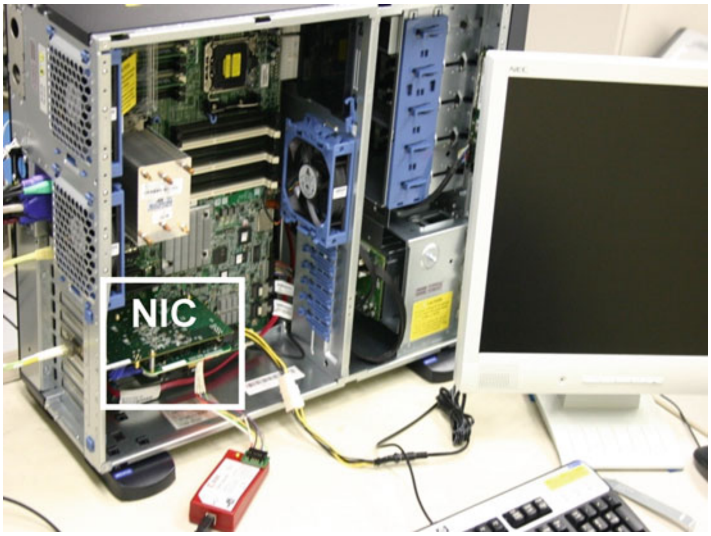


**Fig. 15** Target 20 Gbps FPGA-based NIC

**Table 1** Target NIC specifications

| Item | Features |
|---|---|
| FPGA | Xilinx XC5VLX330T-2 (156 MHz) |
| I/O | XFP 10 Gbps $\times$ 2, PCI Express Gen.1 $\times$ 8 |
| CAD | NEC CyberWorkBench 5.11/Xilinx ISE 11.4 |

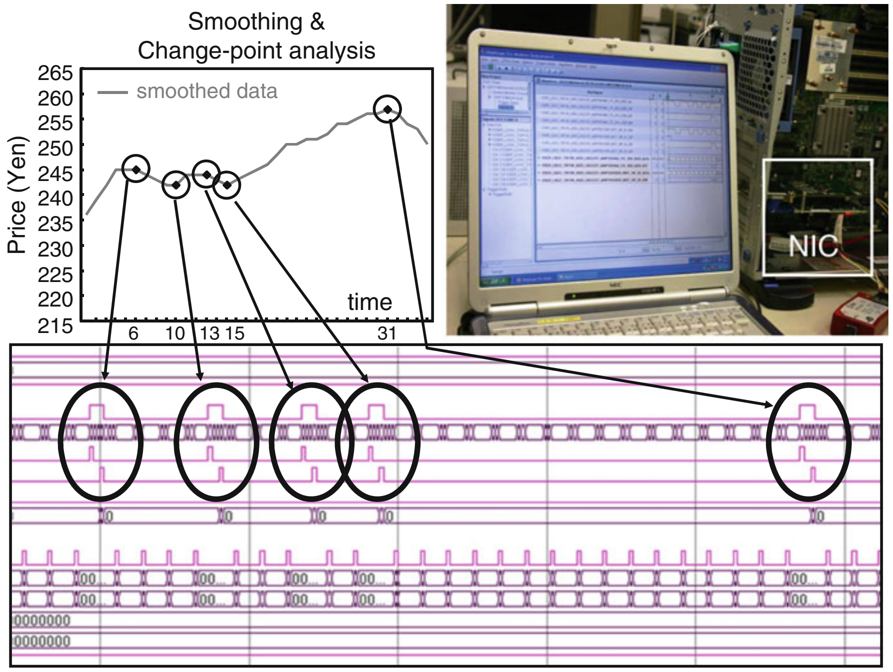


**Fig. 16** Implementation of our motivating example

**Table 2** Logic usage in our motivating example

| Item | Available | Increase |
|---|---|---|
| Number of slice registers | 207,360 | +2,492 (1.2 %) |
| Number of slice LUTs | 207,360 | +4,290 (2.1 %) |
| Number of occupied slices | 51,840 | +1,378 (2.7 %) |

level. As shown in Fig. 16, the logic successfully detects five change points, and it achieves 20 Gbps peak event processing performance; the clock frequency is 156 MHz and the data width is 128 bits (i.e., $156\,\text{MHz} \times 128\,\text{bit} = 19{,}968\,\text{Mbps}$). It should be noted that we have properly received five packets corresponding to the change points on a server although a waveform is shown in the figure.

## 5.2 *Logic Usage*

We have evaluated how much the above implementation increases slice logic utilization of our NIC FPGA. Since the number of occupied slices increases only 2.7 % (see Table 2), the entire logic, including our event processing adapter, can be efficiently implemented.

**Table 3** Performance speed-ups over software

| Item | Software (ns/event) | Our work (ns/event) | Speed-up |
|---|---|---|---|
| Motivating example | 78.9 | 6.4 | 12.3× |
| VWAP | 46.2 | 6.4 | 7.2× |

## *5.3 Performance Speed-Up*

We have evaluated how much our FPGA framework makes it possible to achieve better performance than does CPU software in our motivating example. In order to measure the CPU software performance, we have newly implemented a C source code which directly executes the operations of our motivating example without any CEP systems. This slightly gives an advantage to CPU software because the code eliminates CEP-based performance overheads. In addition, for reference, we have evaluated a simple, famous trading benchmark, referred to as volume-weighted average price (VWAP), which means the ratio of the price traded to the total volume traded over a particular time [32]. This benchmark simply requires an aggregation function without any regular expressions.

Table 3 shows performance comparison results. Here, we have used an Intel Xeon CPU running at 3.3 GHz with a 12 MB cache, and the CPU software performance is an average of ten measurements due to its variability. In the table, compared with CPU software, our work achieves both 12.3 times better performance with respect to our motivating example and 7.2 times better one with respect to the VWAP benchmark. These speed-ups come from the facts that our C-based event language enables the two applications to be efficiently described and our logic construction method allows both regular expressions with Kleene closure and aggregation functions to be directly mapped to hardware in a pipelined way. In addition, the reason why our motivating example significantly shows a better speed-up than does the VWAP benchmark is because our motivating example requires the execution of both regular expressions with Kleene closure and aggregation functions. Thus, the results imply that our work is potentially suitable for achieving higher performance than does CPU software with respect to many CEP application domains that require both regular expressions with Kleene closure and aggregation functions.

## 6 Related Work

Our research differs in a number of respects from the current body of research on CEP. Our event language is designed to achieve both high-throughput regular expressions with Kleene closure and support of various aggregation functions in order to obtain useful information from event sequences along a specified regular expression. Table 4 summarizes related work discussed in Sect. 1.

**Table 4** Related work

| Item | Software CEP [12] | Hardware CEP | |
|---|---|---|---|
| | | Previous work [15] | Our work |
| Language | SQL | SQL | C |
| Regular expression | Yes | Yes | Yes |
| Aggregation | Yes | No | Yes |
| Peak throughput | 0.12 Gbps | 1 Gbps | 20 Gbps |

Much work has been conducted on high-speed logic construction from regular expressions [20,22–29]. While Kennedy et al. [27] have achieved over-40 Gbps deep packet inspection, they support no aggregation functions required for CEP. Sadoghi et al. [33] have proposed an interesting financial trading platform. While they implement an event matching mechanism, they support no aggregation functions. Jiang and Gokhale [34] have presented a machine learning platform for up to 80 Gbps network traffic. They, however, only classify data without any event matching mechanisms. Thus, existing work supports either control flow operations based on regular expressions or data flow operations which use normal functions.

We focus on efficiently constructing single-stream CEP queries on FPGAs, as shown in the above sections. In order to support multi-stream CEP queries on FPGAs, our event language needs to incorporate multi-stream functions; such as *partitioning* (i.e., group by), which classifies a stream contents in terms of fields, and *join*, which merges multiple streams into a single stream by using a common value, operators in SQL; in it. For this purpose, Woods et al. [15] have proposed a *stream partitioning* technique that enables query circuits to be mostly shared. In addition, Teubner and Mueller [35] have presented a *handshake join* technique that leverages hardware parallelism. Since the techniques have high degree of compatibility with SQL, the SQL interfaces implemented on our C-based event language will simply support similar operators. Alternatively, we will implement such interesting ideas as C libraries. Moreover, our language differs substantially from such C-based stream languages as Brook [36] and StreamIt [37] in having its control flow based on regular expressions.

## 7 Conclusion

The requirements for fast CEP will necessitate hardware acceleration which uses reconfigurable devices. Unlike conventional SQL-based approaches, our approach features logic automation constructed with a new C-based event language that supports regular expressions on the basis of C functions, so that a wide variety of event-processing applications can be efficiently mapped to FPGAs. In a financial trading application, we have, in fact, achieved 12.3 times better event processing performance on an FPGA-based NIC than does CPU software. In future work, we intend both to support SQL interfaces on our C-based event language and to employ partial reconfiguration for run-time function replacement.

**Acknowledgments** We thank O. Arakawa, T. Iihoshi, K. Ichino, K. Imamura, O. Itoku, S. Kamiya, S. Karino, S. Morioka, A. Motoki, S. Nakadai, Y. Nakamura, N. Nishi, M. Nishihara, M. Petersen, H. Tagato, M. Tani, A. Tsuji, K. Wakabayashi, and N. Yamagaki for their great contributions.

## References

1. SASE+ http://avid.cs.umass.edu/sase/index.php?page=navigation_menus
2. Abadi DJ, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: a new model and architecture for data stream management. Int J Very Large Data Bases 12(2):120–139
3. Anicic D, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang J-H, Linder W, Maskey AS, Rasin A, Ryvkina E, Tatbul N, Xing Y, Zdonik S (2005) The design of the borelias stream processing engine. In: Biennial conference on innovative data systems research. Very Large Data Base Endowment Inc. Franklin County, Ohio, USA, pp 277–289
4. Anicic D, Fodor P, Rudolph S, Stuehmer R, Stojanovic N, Studer R (2010) A rule-based language for complex event processing and reasoning. In: International conference on web reasoning and rule systems. Springer, Berlin, (Lecture Notes in Computer Science series), pp 42–57
5. Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah M (2003) TelegraphCQ: continuous dataflow processing for an uncertain world. In: Biennial conference on innovative data systems research. Very Large Data Base Endowment Inc. Franklin County, Ohio, USA, pp 269–280
6. Demers A, Gehrke J, Panda B, Riedewald M, Sharma V, White W (2007) Cayuga: a general purpose event monitoring system. In: Biennial conference on innovative data systems research. Very Large Data Base Endowment Inc. Franklin County, Ohio, USA, pp 412–422
7. Gedik B, Andrade H, Wu K-L, Yu PS, Doo MC (2008) SPADE: the system s declarative stream processing engine. In: ACM international conference on management of data. ACM, New York, USA, pp 1123–1134
8. Gyllstrom D, Agrawal J, Diao Y, Immerman N (2008) On supporting Kleene closure over event streams. In: International conference on data engineering. IEEE Computer Society, Washington, DC, USA, pp 1391–1393
9. Kraemer J, Seeger B (2004) PIPES-a public infrastructure for processing and exploring streams. In: ACM international conference on management of data. ACM, New York, USA, pp 925–926
10. Naughton J, Chen J, Kang J, Prakash N, Shanmugasundaram J, Ramamurthy R, Chen R, DeWitt D, Galanis L, Luo Q, Tian F, Zhang C, Jackson B, Gupta A, Maier D, Tufte K (2001) The Niagara internet query system. IEEE Data Eng Bulletin 24(1):27–33
11. The STREAM Group (2003) STREAM: the stanford stream data manager. IEEE Data Eng Bulletin 26(1):19–26
12. Mendes MR, Bizarro P, Marques P (2009) A performance study of event processing systems. Performance Evaluation and Benchmarking, vol 5895, pp 221–236
13. OPRA Updated traffic projections 2011 & 2012. http://www.opradata.com/specs/upd_traffic_proj_11_12.pdf
14. Mueller R, Teubner J, Alonso G (2009) Streams over wires – a query compiler for FPGAs. Int Conf Very Large Data Bases 2(1):229–240
15. Woods L, Teubner J, Alonso G (2010) Complex event detection at wire speed with FPGAs. Int Conf Very Large Data Bases 3(1–2):660–669
16. Inoue H, Takenaka T, Motomura M (2011) 20Gbps C-based complex event processing. IEEE international conference on field programmable logic and applications. IEEE Computer Society, Washington, DC, USA, pp 97–102

17. NEC CyberWorkBench. http://www.nec.com/global/prod/cwb/
18. Wakabayashi K, Schafer BC (2008) All-in-C: behavioral synthesis and verification with CyberWorkBench. High-Level synthesis. Springer, Berlin, pp 113–127
19. Zemke F, Witkowski A, Cherniak M (2007) Pattern matching in sequences of rows. ANSI Standard Proposal
20. Sidhu R, Prasanna V (2001) Fast regular expression matching using FPGAs. In: IEEE symposium on field-programmable custom computing machines. IEEE Computer Society, Washington, DC, USA, pp 227–238
21. Mandelbrot RB, Hudson RL (2006) The (mis)behavior of markets – a fractal view of risk, ruin, and reward. Basic Books
22. Baker ZK, Prasanna VK (2004) A methodology for the synthesis of efficient intrusion detectoin systems on FPGAs. In: IEEE symposium on field programmable custom computing machine. IEEE Computer Society, Washington, DC, USA, pp 135–144
23. Bruschi F, Paolieri M, Rana V (2010) A reconfigurable system based on a parallel and pipelined solution for regular expression matching. In: IEEE international conference on field-programmable logic and applications. IEEE Computer Society, Washington, DC, USA, pp 44–49
24. Cho YH, Navab S, WHM-Smith (2002) Specialized hardware for deep network packet filtering. In: International conference on field programmable logic and applications. Springer-Verlag, London, UK, pp 452–461
25. Clark CR, Schimmel DE (2003) Efficient reconfigurable logic circuits for matching complex network intrustion detection patterns. In: International conference on field programmable logic and applications. Springer, Berlin, (Lecture Notes in Computer Science series), pp 956–959
26. Hutchings BL, Franklin R, Carver D (2002) Assisting network intrusion detection with reconfigurable hardware. In: IEEE symposium on field-programmable custom computing machine. IEEE Computer Society, Washington, DC, USA, pp 111–120
27. Kennedy A, Wang X, Liu Z, Liu B (2010) Ultra-high throughput string matching for deep packet inspection. In: ACM/IEEE design, automation & test in Europe. European Design and Automation Association 3001 Leuven, Belgium, pp 399–404
28. Lin C-H, Huang C-T, Jiang C-P, Chang S-C (2007) Optimization of pattern matching circuits for regular expression on FPGA. IEEE Trans Very Large Scale Integration Syst 15(12): 1303–1310
29. Yamagaki N, Sidhu R, Kamiya S (2008) High-speed regular expressin matching engine using multi-character NFA.In: IEEE international conference on field programmable logic and applications. IEEE Computer Society, Washington, DC, USA, pp 131–136
30. Torii S, Suzuki S, Tomonaga H, Tokue T, Sakai J, Suzuki N, Murakami K, Hiraga T, Shigemoto K, Tatebe Y, Obuchi E, Kayama N, Edahiro M, Kusano T, Nishi N (2005) A 600MIPS 120mW 70uA leakage triple-CPU mobile application processor chip. In: IEEE International Solid-State Circuits Conference. IEEE Piscataway, NJ, USA, pp 136–137
31. Altera (2002) Atlantic interface specification ver 3.0
32. VWAP http://en.wikipedia.org/wiki/VWAP
33. Sadoghi M, Labrecque M, Singh H, Shum W, Jacobsen H-A (2010) Efficient event processing through reconfigurable hardware for algorithmic trading. Int Conf Very Large Data Bases 3(1–2):1525–1528
34. Jiang W, Gokhale M (2010) Real-time classification of multimedia traffic using FPGA. In: IEEE international conference on field programmable logic and applications. IEEE Computer Society, Washington, DC, USA pp 56–63
35. Teubner J, Mueller R (2011) How soccer players would do stream joins. In: ACM International conference on management of data. ACM, New York, USA, pp 625–6s
36. Brook spec. v0.2 (2003) http://merrimac.stanford.edu/brook/brookspec-v0.2.pdf
37. StreamIt language specification version 2.1 (2006) http://groups.csail.mit.edu/cag/streamit/papers/streamit-lang-spec.pdf

# Model-based Performance Evaluation of Dynamic Partial Reconfigurable Datapaths for FPGA-based Systems

**Rehan Ahmed and Peter Hallschmid**

## 1 Introduction

One approach to reducing the unit cost of FPGAs is to time-multiplex functionality, thus reducing the required size of the FPGA. This is accomplished via dynamic partial reconfiguration (DPR) in which partial configuration bitstreams are loaded into FPGA configuration memory at run-time. In addition to saving area, this technique can also be used to dynamically optimize the system for different phases of execution to improve speed and power efficiency.

A dynamic reconfiguration is achieved by transferring partial configuration bitstreams from external memory to FPGA configuration memory via the *DPR datapath*. The performance of this datapath can have a significant impact on overall system performance and should be considered early in the design cycle. This is especially true for systems in which partial reconfigurations occur on the critical path and for fine-grained architectures in which programming bitstreams are longer. Unfortunately, predicting reconfiguration performance early is difficult, in part, due to non-deterministic factors such as the sharing of the bus structures with traffic not related to the reconfiguration process. This type of traffic, which we call *non-PR traffic*, is typically due to memory accesses initiated by on-chip processors or other peripherals attached to the bus. This traffic is correlated with the unknown run-time behavior of the software which we assume to be a stochastic process.

In this chapter, we describe a method of modeling the reconfiguration process using well-established tools from queueing theory. By modeling the reconfiguration datapath using queueing theory, performance measures can be estimated early in the design cycle for a wide variety of architectures with non-deterministic elements. This approach has many advantages over current approaches in which hardware measurements are made after the system has been built. Reconfiguration

R. Ahmed (✉) • P. Hallschmid
School of Engineering, The University of British Columbia, Canada
e-mail: rehan.ahmed@ubc.ca; peter.hallschmid@ubc.ca

performance is heavily dependent on the detailed characteristics of the datapath and on the particular workload imposed on the system during measurement and thus can only be used to make projections for systems similar to that used for initial measurements. A flexible modeling method that works for a wide range of DPR architectures allows us to explore a large design space early in the design flow ultimately leading to better implementation.

A set of guidelines is proposed for mapping any reconfiguration datapath to a multi-class queueing network such that the various phases of the reconfiguration traffic (PR traffic) and non-PR traffic can be modeled simultaneously using multiple classes of traffic. Thus, each class can be assigned different queueing and routing properties thus allowing for the correct modeling of shared bus resources. Once defined, this model can then be used to generate performance estimates such as the reconfiguration throughput, resource utilization, and memory requirements. This can be accomplished by solving the network using either an analytical or simulation-based approach, each of which has its advantages and disadvantages.

Performance estimates generated from the model can be used in the design process in several ways: bottlenecks can be identified, performance trends can be generated to relate system performance to hardware parameters, and the effects of non-PR traffic on PR traffic and vice-versa can be quantified. The advantage of using queueing networks over other modeling techniques is that queueing theory has a wealth of analytical and simulation-based approaches for solving a wide variety of modeling features.

The remainder of the chapter is organized as follows: Sects. 2 and 3 discuss relevant reconfiguration metrics and techniques used to make performance measurements and predictions, respectively. Section 4 describes how queueing theory can be used to model reconfiguration datapaths and provides a scheme for mapping them to queueing networks. Finally, Sect. 5 provides a case study of a two-phase DPR system with a custom controller in the presence of non deterministic non-PR traffic.

## 2 DPR Performance Metrics

Industrially available FPGAs such as Xilinx's Virtex-II, -IV, and -V devices support DPR. In addition to the programmable array, Xilinx-based systems often include either Microblaze soft processor cores or PowerPC (PPC) hard cores both of which can either be connected directly to the processor local bus (PLB) or to the slower on-chip peripheral bus (OPB) via a bridge. Modules implemented in the programmable array can be attached to the processor and can then serve as peripherals. Both the PLB and the OPB are shared resources that can be used for PR and non-PR traffic depending on the configuration of the datapath.

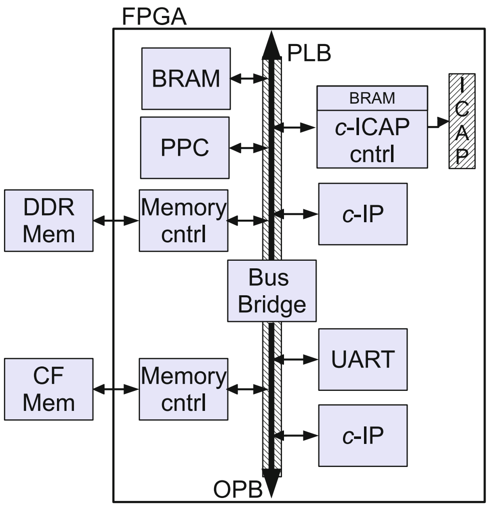


**Fig. 1** A typical Xilinx-based reconfigurable system

The configuration memory of Xilinx-based FPGAs such as that shown in Fig. 1 can be accessed via the internal configuration access port (ICAP). The size of an ICAP can range from 8-bits in width for the Virtex-II Pro to 32 bits in the Virtex-IV and Virtex-V families. Xilinx provides soft IP cores and software libraries to facilitate the loading of bitstream data into configuration memory via the ICAP. In this configuration, the processor acts as the reconfiguration controller. For a more optimal solution, a custom IP can be created to interface with the ICAP and to act as a bus master such that it loads the bitstream via DMA [4]. An example system with a custom ICAP controller interfaced with the PLB bus is shown in Fig. 1. Results have shown that a custom-based solution provides a 58$\times$ increase in reconfiguration speed [4].

The most important performance metric for a reconfiguration datapath is the *reconfiguration time* which is generally considered to be the time needed to transfer a partial bitstream from off-chip memory to the internal FPGA configuration memory. In the early stages of system development, it is not only important to predict the reconfiguration time of the system but also its variability, especially for safety-critical systems. The primary source of variability comes from non-deterministic components involved in the reconfiguration process. Examples of non-deterministic components include the ICAP port and buses shared with non-deterministic non-PR traffic.

In addition to reconfiguration time, another important metric includes the *reconfiguration throughput* of a particular stage, or phase, of the datapath. The inverse of this is the estimated time spent in a phase—the sum of all phases corresponds to the time spent per byte to transfer a bitstream from external memory to configuration memory. *Memory utilization* is also important because it directly corresponds to the expected size requirements of memory. Further, utilization of various hardware components can be useful for identifying the bottlenecks of the system.

There are several factors that influence the aforementioned metrics in a hardware implementation. These include the speed and type of external memory, the speed of the memory controller, the type of ICAP controller and the way it is interfaced with the bus. Depending on these choices, reconfiguration time can vary significantly across implementations.

## 3 The Landscape of DPR Performance Estimation

Several methods have been developed for evaluating and estimating DPR performance metrics. Papadimitrou et al. [10] provide an extensive survey of recent works that measure reconfiguration time for "real-world" platforms. This survey compares the reconfiguration time of several platforms as a function of the type of bitstream storage (i.e., BRAM, SRAM, DDR, or DDR2 to name a few), the ICAP width and operating frequency, the bitstream size, and the type of controller, whether it be vendor provided or custom built. The expected reconfiguration time is calculated based on the time spent in the different phases of the reconfiguration process. The limitations of this approach were explored by Gries et al. [6] and Galdino et al. [5] who showed that such predictions can be in error by one to two orders of magnitude. Perhaps the most important disadvantage to these approaches is the fact that these systems must be built first before they can be measured. This is counter to the motivation of this work which is to predict performance trends early in the design cycle and without the need to implement the system.

Claus et al. [4] provide a set of equations to calculate the expected reconfiguration time and throughput. These equations are based on a value called the *busy-factor* of the ICAP, which is the percentage time that the ICAP is busy and not able to receive new bitstream data. Unfortunately, calculating the busy factor requires that the system be built first. Further, this approach only works well for datapaths in which the ICAP is the bottleneck. An additional limitation of the approach is that it does not account for the effects of non-PR traffic on PR performance.

The reconfiguration cost model proposed by Papadimitriou et al. [10] is the first cost model based on a theoretical analysis of the different phases, of dynamic PR. The PR process is divided into phases and the total reconfiguration time is calculated by adding together the time spent in each phase. Because software time stamps are used to make measurements, additional contributions to the measurement beyond that of the bus transfer are not included such as the time spent by the PR controller to initiate the transfer and the impact of non-PR traffic. Because these additional contributions can be significant and are implementation specific, phase measurements are only suitable for use with systems that are incrementally different from the measured system. In this chapter we construct our model using hardware specifications rather than from measured values from software stamps. Further, we model non-PR traffic as a separate factor from the phases of PR traffic.

Hsiung et al. [8] developed a SystemC model of a simple system consisting of a processor, bus, memory, and reconfigurable region and used it to evaluate system performance and to identify bottlenecks. This work was restricted by the limited availability of SystemC functions to model the PR process and considered only an oversimplified form of the PR datapath.

The modeling approach proposed in this chapter accounts for the stochastic behavior of non-deterministic components in the system such as the ICAP port and buses shared with non-PR traffic. In addition, it models the PR datapath at a detailed level and accounts for factors such as burst size, memory speed, and the degree of pipelining. Most significantly, our model does not require that systems be built to evaluate their performance.

## 4 Modeling DPR with Queueing Networks

In the following sections, we describe the approach we take to model the reconfiguration datapath using queueing networks. First, we list the datapath features we model thus defining the inclusivity of the model. Second, we outline which principles from queueing theory are used in the proposed modeling approach. Third, we provide a set of guidelines for mapping datapaths to queueing networks.

### *4.1 DPR Features Modeled*

We consider a generalized datapath as shown in Fig. 2 in which a partial reconfiguration consists of the transfer of partial bitstreams from external memory to FPGA configuration memory through a series of zero or more intermediate memories. Transfers between memories can occur over dedicated interconnect or over buses shared with other, non-PR, traffic. We refer to the transfer between any two intermediate memories as a *phase*.

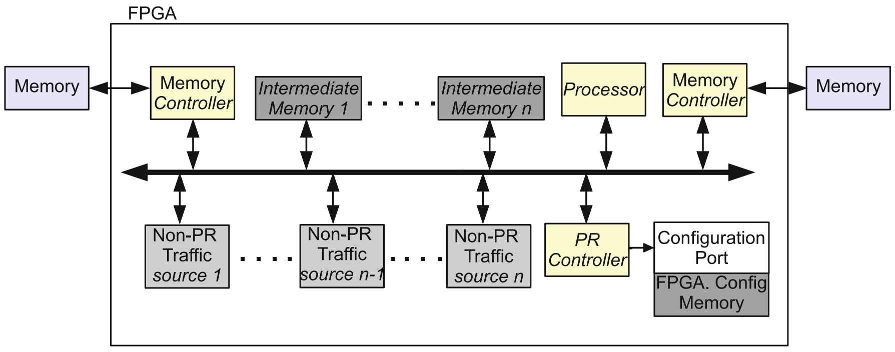


**Fig. 2** A generalized datapath for the dynamic partial reconfiguration of an FPGA-based system

Assuming this generalized datapath, a number of datapath components and features must be modeled. The following is a list of these components and features with a brief description:

## Architecture Components

- *Processor*: In a typical dynamically reconfigurable system, software running on a processor core initiates the reconfiguration process. The processor may be involved in one more phases of the reconfiguration by controlling the transfer of bitstream data on a word-by-word basic or via a DMA transfer. If, at any point, control is handed over to a custom controller, this controller ends the reconfiguration by informing the processor with an interrupt. To model the effectiveness of the reconfiguration datapath, we need only consider the time at which the first packet of bitstream data enters the first phase until the last packet of bitstream data exits the final phase. We need not consider the time spent by the processor initiating the reconfiguration.
- *Bus*: The bus is a shared resource that is used to perform DPRs and by other peripherals for inter-chip communications. From the perspective of the reconfiguration process, the processor's use of the bus is a source of nondeterminism. Reconfiguration time is directly affected by bus access patterns initiated by the processor.
- *Memory*: Memories are broadly divided into off-chip and on-chip memories. PR traffic (partial bitstream) is typically stored in off-chip memory, the type of which varies from compact flash (CF) card to DDR memory. On-chip memory in the form of block RAM (BRAM) is used to cache PR traffic before it is transferred to configuration memory.
- *Configuration Port*: The configuration port acts as a gateway through which partial bitstreams are transferred to FPGA configuration memory. The width and operating frequency of the port vary from vendor to vendor and from device to device.

## Operating Features

- *Reconfiguration Phases*: The reconfiguration process typically has more than one phase. We define a phase as a transfer of bitstream data across a dedicated or shared routing resource ending at a memory. Phases can be sequential such that an entire block of data must complete a phase before the subsequent phase begins, or phases can be pipelined such that a phase can simultaneously operate on the same block of data as the previous phase. This is accomplished using a FIFO.
- *Size of Transfers*: PR traffic through the bus can either be transferred word-by-word or in bursts.
- *Bus Arbitration*: Bus arbitration should be modeled to enforce fairness in the use of a shared bus. Typically, a round-robin schedule is used.

**Non-determinism**

- *Non-PR Traffic*: There can be random events in the system such as traffic coming from different intellectual property (IP) blocks with nondeterministic start times and end times. This traffic competes for shared resources such as the bus with PR traffic.
- *Configuration Port*: The configuration port for Xilinx devices is known as ICAP port. The transfers across the ICAP can only be initiated when *ICAP busy* signal is not active. The behavior of this signal is stochastic in nature.

## 4.2 Application of Queueing Networks to DPR Modeling

We propose the use of queueing theory to model the generalized datapath described in the previous section. A brief introduction to queueing networks is provided in the appendix with several concepts discussed such as *balance equations*, *marginal probabilities*, *joint-probabilities*, and *Jackson networks*. Jackson networks provide a way to express the probability of an aggregate state of the network as the product of its single-node state probabilities. Another way to express this is to say that the *joint probability* $p_n$ is equal to the product of its marginal-probabilities $q_n$ over all nodes. This is also known as a product-form solution. In its product form, the joint-probability can be calculated easily and can then be used to calculate various performance estimates.

Unfortunately, Jackson networks are not suitable for modeling dynamic reconfiguration because all customer traffic (i.e., packets of PR and non-PR traffic) in such networks belong to a single class and therefore have the same routing, arrival-rate, and service-rate characteristics. BCMP networks are an extension of Jackson networks but with the additional feature that several classes of traffic can co-exist in the network each with their own statistical characteristics [1]. To correctly model the various phases and types of traffic in the system, BCMP networks are used.

For BCMP networks, it is required that all network nodes belong to one of the following four types:

1. *Type-1*: *First-come-first-serve (FCFS)* node. All classes must have the same service time.
2. *Type-2*: *Processor-sharing (PS)* node. Classes can have different services times but order is not maintained. Instead, customers are processed in time slices weighted by their total service time.
3. *Type-3*: *Infinite-server (IS)* node. An infinite number of servers allow all customers to be serviced immediately.
4. *Type-4*: *Last-come-first-serve (LCFS)* node with preemptive resume. Preempted customers are resumed without loss.

As will be discussed later in the chapter, this restriction is a problem only when we solve the network analytically.

Assuming there are $N$ nodes and $C$ classes in the queueing network, the joint probability for the BCMP queueing network is given as

$$p[S = (y_1, y_2, \dots, y_N)] = \frac{1}{G} d(S) \prod_{i=1}^{N} F_i(y_i) \tag{1}$$

where,

- $S = (y_1, y_2, \dots, y_N)$ is the aggregated state of an $N$ node network with $y_i$ denoting the state of the $i$th node. $y_i = (y_{i,1}, y_{i,2}, \dots, y_{i,C})$ is a vector of length $C$ with $y_{i,j}$ representing the number of class-$j$ customers at node $i$.
- $G$ is a normalization constant that insures the sum of probabilities is equal to 1.
- $d(S)$ is a function of all queue arrival times. All networks used in this chapter are closed chains and thus this function becomes 1.
- $F_i(y_i)$ is a function to each node type.

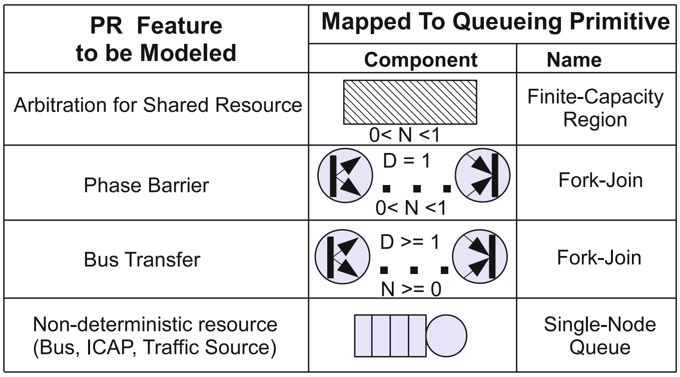
A more detailed description of Eq. 1 can be found in [1].

In order to model different types of traffic in the DPR datapath, as explained in Sect. 4.1, the BCMP network type-1 node has to support different queue service time per customer class. This feature is needed to model the various types of traffic passing through a shared resource, where each arriving customer has different service times depending on its class. Further, to accurately model bus level transfers and different phases of the system, a fork–join pair is required. The purpose of fork nodes is to divide incoming queueing customers into one or more sub-customers. All sub-customers generated by a fork must be routed downstream into a corresponding join node. Therefore, all sub-customers must wait until all "sibling" sub-customers have arrived, after which they recombine into the original customer.

To correctly model the datapath, two of the above-mentioned features are required which are incompatible with the BCMP theorem. Thus, they no longer have a product-form expression for the joint-probability and are thus not easily solved analytically. The first incompatibility is that fork–join nodes are used to model the word-by-word transfers of data across the bus. The second is in the use of noncompliant type-1 queueing nodes for which each arriving customer has different service times depending on its class.

One solution for solving a non-product-form network is to use approximation methods. The disadvantage of this is that it cannot be used to model multi-class networks and it is very difficult to find approximations methods that work with both fork–join nodes and non-compliant type-1 nodes simultaneously. A second approach is to reduce the accuracy of the model through the introduction of assumptions such that the network becomes simpler and therefore BCMP compliant which can be solved analytically. The third approach is to use a simulation-based approach and to collect performance statistics rather than to calculate them analytically.

The primary advantage of simulation-based solutions is that all queueing features are supported. This includes fork–join pairs, phase barriers, and FCRs. Additional benefits to a simulation-based approach include the availability of transient analysis



| PR Feature to be Modeled | Mapped To Queueing Primitive | |
|---|---|---|
| | Component | Name |
| Arbitration for Shared Resource | 0< N <1 | Finite-Capacity Region |
| Phase Barrier | D = 1 . . . 0< N <1 | Fork-Join |
| Bus Transfer | D >= 1 . . . N >= 0 | Fork-Join |
| Non-deterministic resource (Bus, ICAP, Traffic Source) | | Single-Node Queue |

**Fig. 3** Mapping of PR features to queueing primitives

and what-if analysis techniques. The disadvantage to simulation-based network solvers is poor run-time complexity; however, this is not an issue due to the relatively small sizes of the networks needed for modeling DPR datapaths. While simulations can provide a solution in cases where an analytical solution is not possible, there are situations in which an analytical solution is required. One such situation is when an analytical solution is required in order to be incorporated into optimization algorithms.

In this section we provide mapping guidelines for both simulation- and analytical-based approaches. The choice of modeling scheme depends on the complexity of the network and intended use of the result. Section 4.3 presents the mapping scheme for simulation-based solutions followed by the mapping scheme for analytical-based solutions in Sect. 4.4.

## *4.3 Mapping Scheme for Simulation-Based Solutions*

Regardless of whether the network is solved analytically or by simulation, a mapping scheme is required to map features between the PR datapath being modeled and the queueing network. This mapping is achieved through various queueing primitives, as shown in Fig. 3, which are used to model different datapath features. Buses and memories used for the reconfiguration process are modeled with single-node queues. The partial bitstream is represented by queueing customers that pass through a sequence of single-node queues that together represent the PR datapath. The partial bitstream is divided into smaller byte or word-sized blocks (i.e., sub-customers) depending on the nature of the transfer. Single-node queues are connected with directed edges that dictate the flow of bitstream data through hardware resources thus defining the *reconfiguration datapath*. Customers are categorized into classes to model different types of traffic whether they be different phases of the PR process or traffic not directly related to the PR process such as the fetching of instructions by the processor core (i.e., non-PR traffic). By using customer classes, we can assign different service rates and different routing policies for each type of traffic passing through shared resources.

### 4.3.1 Separation of Concerns

To capture the behavior of a PR datapath for analysis and the calculation of performance measures, aspects of the overall reconfigurable system not related to the reconfiguration process are not included in the queueing model. The first application of this principle relates to the time frame considered when modeling a PR. We model the beginning of a PR process at the point when bitstream data is first transferred out of external memory to the FPGA. All other activities driving this process such as the initiation of the transfer by the processor are not considered. We model the end of the PR process when the last byte of the partial bitstream has been transferred to configuration memory. The queueing network models only the transfer of bitstream data between these start and end points. *Closed networks* are used as a convenient way to model the assumption that new transfers start only when old transfers are finished.

All aspects of the architecture not directly relevant to the PR process are removed from the model. The only exception to this is the inclusion of all non-PR traffic that shares bus resources with PR traffic. The set of generalized rules for mapping any reconfigurable datapath to queueing network is described below.

### 4.3.2 Modeling Non-deterministic Resources

Components of the PR process that are nondeterministic in nature are modeled as *single-node queues* as shown in the mapping table provided in Fig. 3. Depending on the nature of the hardware components of the datapath being modeled, this can include shared buses, the configuration port (i.e., ICAP port), the source nodes for PR and non-PR traffic, memories, and custom IP cores. In this chapter, we assume that the arrival of PR and non-PR traffic is nondeterministic in addition to the ICAP port and the arbitration of the shared bus. The ICAP port is modeled as a queue with exponential service distribution and with an average service time corresponding to the measured speed of the ICAP as reported in [3]. With respect to the shared bus, the service time is different for each class of customer (i.e., each type of PR and/or non-PR traffic) and is dependent on the number of words per transfer and the speeds of the devices involved in the transfer. Arrivals to the bus are dictated by source queues used for each type of traffic; in the case of PR traffic, this is provided by the queue representing external memory.

Although other distributions are possible, all queues are modeled in this chapter using an exponential service time distribution which is the most commonly used in queueing theory. It corresponds to a process by which the arrival times (service times) are uniformly random and the number of arrivals that occur per interval of equal width is identically distributed.

#### 4.3.3 Modeling Bus Traffic

The transfer of a block of the configuration bitstream from one memory to another corresponds to a *PR phase*. As shown in Fig. 3, bus transfers are modeled using fork–join pairs with a forking degree of $D \geq 1$ and a capacity $N \geq 0$. $D$ represents the degree to which a block is divided for the transfer. If, for example, a 2048-byte block is to be transferred over a bus in 32-bit words, a 512-way fork is needed. After passing through the bus, the 32-bit words are reassembled to form the 2048-byte block via a corresponding join node. By breaking blocks into words as such, we can accurately model the interlaced flow of traffic through the bus.

To model bus arbitration, a *finite-capacity region (FCR)* is used as listed in Fig. 3. The number of customers, $N$, allowed into an FCR is limited to a maximum, $1 \leq N \leq N_{\max}$. When a queue representing a shared bus is placed in the FCR, customers of different classes are allowed into the queue by a round-robin arbitration. This is required to prevent a large number of customers spawned from a fork from filling the queue thus preventing customers of a different class from having interleaved access to the bus.

For block transfers, it may be required that a new PR phase not start until the entire block has completed the previous phase. Join nodes inherently solve this problem by enforcing a synchronization mechanism in which all sub-customers generated from a single fork must arrive at the join node before the join is completed. If this behavior is not needed, then the join node must be placed after the ICAP node when the complete transfer is finished.

Phases may be composed of more than one "sub-phase". For some datapaths, it may be required that the number of blocks that can be "processed" by a series of phases is limited. To model this, the *phase barrier* shown in Fig. 3 is used which is a fork–join pair with a forking degree of $D = 1$ and a finite capacity of $0 \leq N \leq 1$.

#### 4.3.4 Modeling Pipelining

Phase pipelining is modeled by populating the queueing network with more than one customer such that each customer represents a portion of the bitstream. The number of customers in the system represents the number of bitstream blocks that can be processed by different phases of the pipeline at the same time. The maximum number of customers in the system is limited by the maximum number of phases in the pipeline.

### *4.4 Mapping Scheme for Analytical-Based Solutions*

The philosophy behind the analytical solution method is to represent the system using mathematical equation or set of equations from which certain measures can be deduced. The solution of these equations can be a *closed-form solution* or can be obtained through appropriate algorithms from numerical analysis techniques.

The closed-form solution is bounded by basic-operations which make the solution computationally less complex and hence faster to calculate. In many situations where it's not possible to obtain a closed-form solution formula, the solution is often approximated using numerical methods which usually have higher computational complexity.

In the work discussed in this chapter, we focus on analytical models with closed-form solutions. To achieve this, we model the reconfigurable system using a queueing network which can be solved using the BCMP theorem, as explained in the appendix, thus yielding a closed-form solution.

In Sect. 4.3, we introduced a generalized mapping scheme for simulation-based solutions. In doing so, we introduced the use of several modeling features such as fork–join pairs in order to accurately model bus traffic. Some of these queueing features are not compatible with product-form theorems such as BCMP, and consequently approximation methods must be used. If, on the other hand, a closed-form solution is required then the only option is to simplify the network by removing features incompatible with BCMP. Because the network serves as a model of the datapath, a network with fewer modeling features can only be used as a "force-fit" if we assume a simplified view of datapath behavior. As a result, the model loses the ability to capture the real-world behavior of the system, and hence, a difference might exist between the simulation-based and analytical-based results.

To fit the proposed modeling scheme to BCMP, we make the following simplifications:

1. We eliminate the use of fork–join nodes. Thus, we no longer model word-level bus interactions by splitting partial bitstreams into words. Instead, we model the transfer of large blocks across the bus by increasing their queue service time. This can potentially skew the results when one type of traffic uses burst mode transfers. Another implication of removing fork–join pairs is that the interaction between sequential phases cannot be modeled well. In addition to fork–join pairs being useful to break reconfiguration bitstreams into smaller parts, it is also useful for synchronizing packets. All sub-packets created by a fork get blocked at the corresponding join until all sibling sub-packets arrive; this provides a useful way of differentiating between two sequential phases such that a set of packets are transferred to memory in the first phase and, upon completion, the subsequent phase begins.
2. We eliminate the use of finite-capacity constructs which are used to limit the number of queueing customers allowed within a set of one or more queues. These regions are used for our simulation-based approach in two ways. First, we use finite-capacity fork–join pairs to limit the number of packets that can enter a sequence of phases and to then synchronize them. Second, we limit the number of packets allowed to queue at a bus thus enforcing bus arbitration.
3. For BCMP type-1 nodes, all classes sharing a queue must have the same service time. The implication is that we will not be able to assign different service times for different traffic types on a shared resource such as a bus. In order to assign different service times to PR and non-PR traffic at a bus node, we assume that bus nodes use a processor sharing service strategy (i.e, BCMP type-2 node).

## 5 Modeling DPR System: A Case Study

In the following section, we discuss the example reconfigurable datapath shown in Fig. 4a. The architecture used in this example is typically used to lower reconfiguration overhead through the use of a custom reconfiguration port controller implemented using standard FPGA resources. Additionally, the two data-fetching phases of the datapath are pipelined, thus further improving throughput. Below we explain the experimental setup and results achieved through actual measurements and using model-based estimates.

### *5.1 Experimental Setup*

In the example system in Fig. 4a, the partial bitstream is transferred from external DDR memory to configuration memory in two distinct but overlapped phases. The only involvement of the PPC processor core during these phases is to initiate the transfer, while the rest of the transfer is handled by the custom ICAP port controller. We call this system a "two-phase datapath with custom ICAP port controller."

The two phases of the reconfiguration process for this platform are as follows:

- *First phase*: A 128-byte block is transferred from DDR memory to BRAM via the PLB in burst mode.
- *Second phase*: Data is transferred byte-by-byte from BRAM to configuration memory via the ICAP port in 8-bit packets.

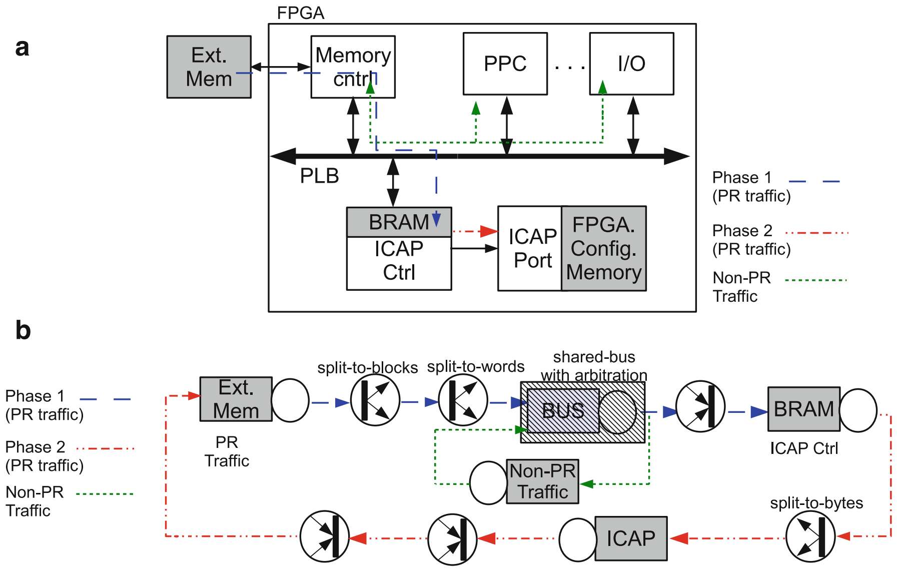


**Fig. 4** (**a**) Example PR system and the corresponding (**b**) queueing network model

Phases-1 and -2 are pipelined such that when the bitstream is being written into the FPGA configuration memory, new bitstream data is fetched from external memory and placed after the previously read data in BRAM. This process is controlled by a custom ICAP controller which transfers partial bitstream data from external DDR memory to the FPGAs configuration memory using DMA transfers. The system is implemented on a Xilinx Virtex-II Pro device thus allowing us to validate the estimated results with measured results.

## 5.2 *Mapping to a Queueing Network*

In order to obtain estimates of the various performance metrics for the example system in Fig. 4a, we map its datapath to a corresponding queueing network model as prescribed by the guidelines provided in Sect. 4.3. Figure 4b is an illustration of the corresponding queueing network model.

There are five queues in the queueing network representing the shared PLB, the ICAP port, BRAM, and source nodes for both PR and non-PR traffic. The source node for PR traffic represents DDR memory in hardware, and its service rate models the initiation of transfers as dictated by the PR controller. The source node for non-PR traffic models sources for all traffic not related to PR such as memory reads and writes involving the PPC, PPC memory, and other IPs attached to the bus. Both phases are bounded by memory transfer primitives (i.e., fork–join nodes) to convert 128-byte blocks to words and words to bytes, respectively.

Customers are divided into three classes, namely, *phase-1* customers, *phase-2* customers and *non-PR traffic* customers. Customers belonging to class phase-1 and phase-2 represent the PR traffic. Phase-1 customers originate from external memory node and reach the BRAM via the bus maintaining the same class. After BRAM, phase-1 customers switch to another class indicating they are now in phase-2 after which they are transferred to the ICAP. The non-PR traffic customers originate from the non-PR source node and transferred to the bus. The bus is shared between PR and non-PR traffic in an interlaced fashion thus modeling the impact of non-PR traffic on PR traffic.

Table 1 provides a mapping between hardware specifications of the example system to calculated queue service times for the network model. The service time of external memory is set to zero as the station acts as a source that immediately provides bitstream blocks once the previous block has exited the system. Given that the operating frequency of the PLB bus is 100 Mhz and it takes 30 bus clock cycles to read 16 (64-bit) words from DDR memory, the service time for PR traffic at the bus is 0.3 $\mu$ s. We assume that non-PR traffic for this system is generated by the PPC and takes one bus cycle per transfer. We assume that the PPC operates at 300 Mhz, has 30 % memory operations, and has a 10 % miss rate for both instruction and data caches. Thus, the service rate for the non-PR queue is 0.0256 $\mu$ s. The ICAP is 8 bits

**Table 1** Different traffic types present at different hardware (H/W) nodes and the corresponding queue service times

| Traffic type at node | H/W node specs. | Service time (u sec) |
|---|---|---|
| PR (phase-1) at Bus | bus freq/width: 100 Mhz/64 bits<br>Mem cycles: 30 Bus cycles/128 bytes | 0.3 |
| Non-PR (phase-2) at Bus | Bus freq/width: 100 Mhz/64 bits<br>Processing: 1 bus cycle/8 bytes | 0.01 |
| PR (phase-1) at ICAP | ICAP freq/width: 100 Mhz/8 bits<br>No of cycles/byte: 1.099 | 0.01099 |
| Non-PR (Phase-2) at PPC | PPC freq: 300 Mhz<br>30 % memory operations 0.1 miss rate | 0.0256 |

in width and operates at a frequency of 100 Mhz. Due to the stochastic nature of the ICAP-BUSY signal, it is assumed that a single byte takes 1.099 cycles, which is a less-than-ideal transfer rate. Thus, the service time for the ICAP queue is 91.6 µs.

## 5.3 *Model-Based Simulation Results*

Simulations were conducted using a modified version of the JMT network simulator [2]. In particular, JMT was modified to include multi-class switching. Simulations were conducted on a 2.2 GHz Intel Core2 Duo T7500 with a typical run-time of 2 min.

In the following sections, the ICAP width and external memory speed were varied thus representing a set of candidate architectures. In both cases, utilization and throughput results are provided.

### 5.3.1 Effects of Varying ICAP Width

Results were generated for the expected utilization and throughput of queues in the network and are shown in Fig. 5. In this plot, the *x*-axis represents ICAP width in bits, the right *y*-axis represents the utilization, and the left *y*-axis represents the throughput in bytes/s of the various queues in the network model.

As the size of the ICAP port is increased from 2 bits to 128 bits, the utilization of the ICAP due to PR traffic drops from 99.8 % to 28.1 %. As the width of the ICAP increases, it can transfer more data per unit time thus reducing its utilization. The reverse trend could be observed for the utilization of the shared bus due to PR traffic which increased from 5.3 % to 95.1 % over the same range. This demonstrates that improvements in ICAP speed places increased pressure on the bus thus making the

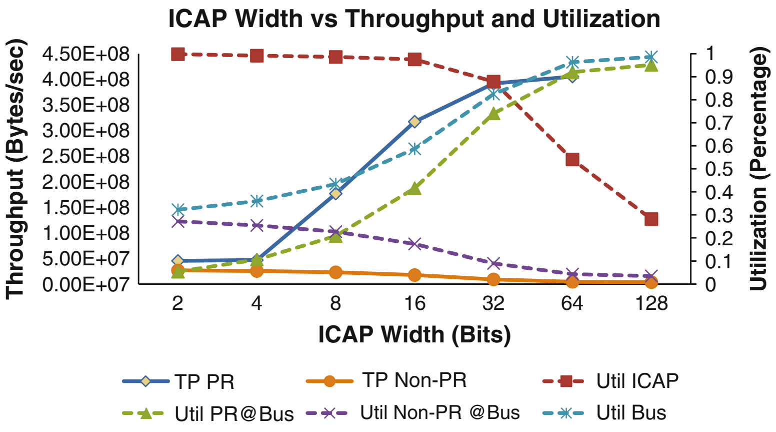


**Fig. 5** The effect of *ICAP width* on utilization and throughput

bus the new bottleneck thus affecting overall throughput. As shown in Fig. 5, overall PR throughput increasingly improved with increased ICAP width until 32 bits after which it began to saturate due to the high utilization of the bus.

For small ICAP widths, utilization of the bus was roughly equal for both PR and non-PR traffic. As PR traffic was increased due to larger ICAP widths, PR requests for the bus increased thus reducing the availability of the bus for non-PR traffic. As a consequence, the utilization of the bus due to non-PR traffic reduced from 27.2 % to 3.5 %, and its throughput correspondingly suffered. These results clearly demonstrate the relationship between PR and non-PR traffic in a system with shared resources and supports the need to consider non-PR traffic early when designing such a system.

### 5.3.2 Effects of Varying External Memory Speed

Results were generated for the expected utilization and throughput of the ICAP and the bus as a function of external memory speed as shown in Fig. 6. In this plot, the $x$-axis represents the number of external memory access cycles, the right $y$-axis represents queue utilization, and the left $y$-axis represents throughput in bytes/s.

Memory speed was varied by changing the number of bus cycles needed to perform a memory read over the PLB. The default number of bus cycles was 30 which corresponds to DDR memory. Experiments were conducted over the range of 10 to 70 cycles representing a broad range of memory speeds for an ICAP width of 32 bits. As the number of bus cycles was increased, thus representing slower memories, the utilization of the bus due to PR traffic increased from 39.5 % to 87.8 %. At the same time, the utilization of the bus due to non-PR traffic was reduced. Longer PR transfers due to slower memory resulted in higher utilization of the bus due to PR traffic, and thus, non-PR had less utilization of the bus. Longer

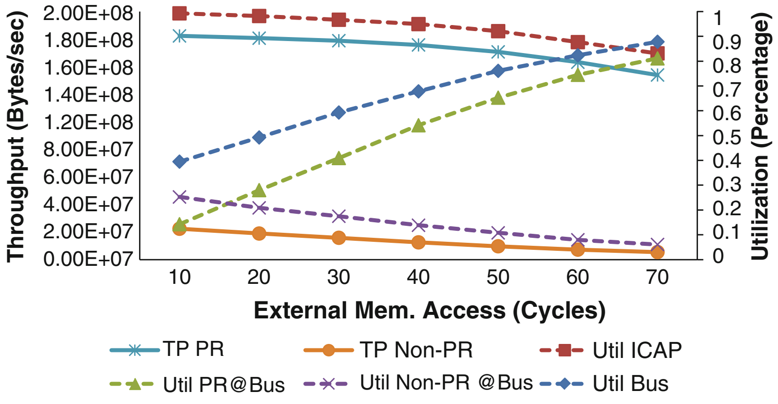


**Fig. 6** The effect of *external memory speed* on utilization and throughput

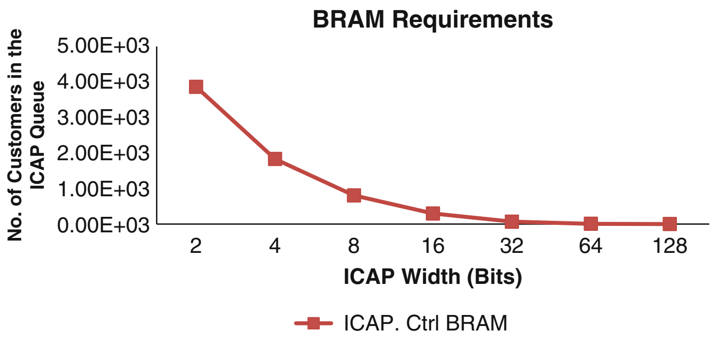


**Fig. 7** The effect of *ICAP width* on the expected BRAM size requirements

bus transfers also resulted in fewer transfers of PR traffic across the ICAP, and thus, its utilization went down. Overall, the throughput of both PR and non-PR were negatively affected by slower external memory.

### 5.3.3 Expected BRAM Requirements

The average number of customers (bytes) for the ICAP queue is shown in Fig. 7 which provides insight into the BRAM needs of the system. As the size of the ICAP was varied from 2 bits to 128 bits, the average number of customers followed an exponential decay from 3860.67 bytes to 1.65 bytes. These results predict that a BRAM size of 512 bytes would be more than sufficient for the 8-bit port. Results of this type demonstrate how the proposed model could be used to determine memory requirements of a system before it is built.

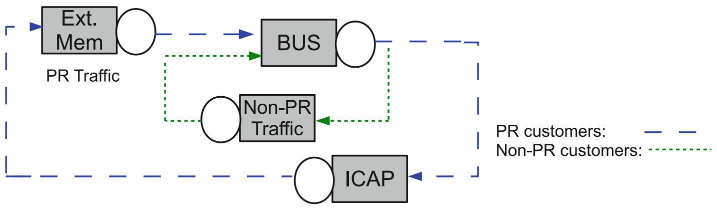


**Fig. 8** Queueing network model of the example datapath

## *5.4 Model-Based Analytical Results*

In this section, we present results for an analytical solution to the example datapath. Because this platform is relatively simple, it is well suited to be "force-fit" to a BMCP network. This is true because it has only two simple phases for PR traffic. The difference in transfer time of the two phases can be approximated using queue service times.

To illustrate the analytical approach, we map a simplified version of the datapath shown in Fig. 4a, following the mapping guidelines provided in Sect. 4.4. The resultant BCMP queueing network is shown in Fig. 8. This model consists of four nodes and two job classes. One job class is used for PR traffic and the second for non-PR traffic. All queues use FCFS queue service strategies.

This network of queues can be solved analytically using the BCMP theorem, which states that the probability of a particular steady network state is given by Eq. 1, which represents the closed-form solution. Upon solving, different performance measures such as throughput and utilization of different nodes in the queueing network can be deduced.

For the network shown in Fig. 8 which is a closed network, the term $d(S)$ in Eq. 1 equates to one and $G$ represents the normalization constant. The calculation of this normalization constant, $G$, requires that all states of the network be considered. A more tractable approach makes use of an iterative algorithm that does not need to consider the states of the queueing network when computing the normalization constant. A commonly used algorithm of this type is the *mean value analysis (MVA)* which gives the mean values of the performance measures directly without computing the normalization constant [11].

### 5.4.1 Effects of ICAP Width

We apply the MVA algorithm along with Eq. 1 to determine PR and non-PR traffic throughput and utilization. Figure 9 shows the obtained analytical results compared with the simulation-based solutions for varying ICAP-width from 8 bits to 128 bits. The solid lines represent the ICAP utilization while the dashed lines represent the throughput for the (A)nalytical and (S)imulation-based results. It is evident

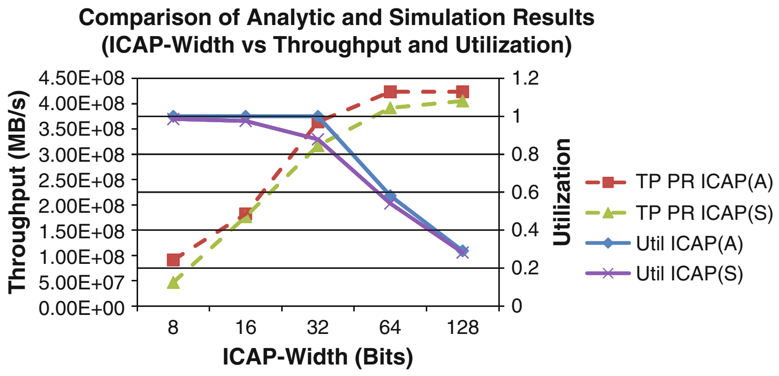


**Fig. 9** A comparison of analytical and simulation-based utilization and throughput results as a function of *ICAP width*

**Table 2** Comparison of measured, estimated, and analytical results for an ICAP width of 8 bits

| | Measured | Estimated | Analytical |
|---|---|---|---|
| TP-PR (MB/s) | 90 | 89.7 | 91 |
| RT (ms) | 1.66 | 1.67 | 1.64 |
| (BS Size:150 KB) | | | |

from the graph that the results found through an analytical approach closely match that generated using a simulation-based approach. Thus, the assumptions made for the analytical mapping scheme for our example system did not adversely affect estimation accuracy.

### 5.4.2 Model Validation

In Table 2, physical measurements were compared against those estimated using both the simulation-based and analytical-based approaches for a bitstream of size 150 KB. Results showed that both approaches were able to predict throughput with 0.3 % error. This result helps to validate that the proposed model provides a reasonable estimate of PR throughput compared to hardware. A more detailed calibration of the model based on hardware measurements for a wider variety of platforms will be the subject of future work.

## 6 Summary

In this chapter, we described a method for modeling the DPR of FPGA-based systems using multiclass queueing networks. We provided a generalized scheme for mapping hardware components of the reconfiguration datapath to queueing primitives. We provided an example system which we modeled and solved using simulation- and analytical-based approaches. Results were provided for system throughput and utilization as a function of ICAP width and external memory speed. These results were used to identify bottlenecks and optimize the system for both reconfiguration time and cost in the presence of non-PR traffic.

The proposed modeling approach is a promising tool for the design and evaluation of dynamically reconfigurable systems. It helps system designers to make informed decisions early in the design process thus avoiding the time and costs associated with building candidate systems. Further, the ease at which candidate architectures can be evaluated allows for a broader exploration of the design space and results in a faster and lower-risk design process.

## Appendix

Queues are formed whenever there is competition for limited resources. Often this happens in real-world systems when there is more demand for service than there is capacity for service. Due to this discrepancy, the customers for a service entering the system form queues. Queueing theory provides a set a tools for analyzing such systems to predict system performance under varying circumstances. In its simplest form, a queueing model consists of a single service facility containing both a queue and a server. Customers arriving at the input of the service facility can be served immediately or must wait in the queue until the server is free. After being served, customers leave the service facility. This type of arrangement is also known as a "single-node" system, as shown in Fig. 10a. The arrival rate, $\lambda$, and service rate, $\mu$, of the queue are nondeterministic and are therefore modeled as stochastic processes.

Queueing theory includes mechanisms for calculating several useful measures including the average wait time in the queue ($W_q$) and the system ($W$), the expected number of customers in the queue ($L_q$) and the system ($L$), and the response time of a queue, its utilization, and system throughput. Suppose that the number of customers in the system at time $t$ is $N(t)$, then the probability of having $n$ customers in the system is $p_n(t) = Pr\{N(t) = n\}$, and $p_n = Pr\{N = n\}$ in steady state. For a system

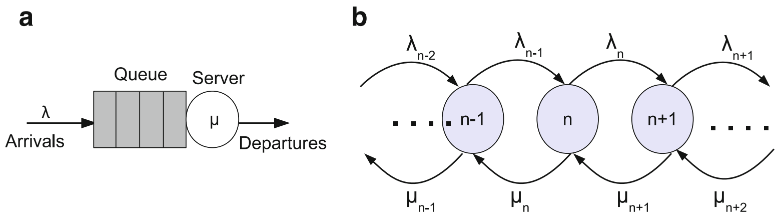


**Fig. 10** (**a**) Single-node queueing system, (**b**) birth-death process

with up to $c$ queues, the expected value of the mean number of customers in the system and in the queue is

$$L = E[N] = \sum_{n=0}^{\infty} np_n, \tag{2}$$

and

$$L = E[N_q] = \sum_{n=c+1}^{\infty} (n-c)\,p_n, \tag{3}$$

respectively. Little's formulas [7] relates these estimates for both the queue and the system using

$$L = \lambda W \tag{4}$$

and

$$L_q = \lambda W_q. \tag{5}$$

A system can be represented by a set of states with state transitions for which states are defined by the number of customers in the system. The system can occupy only one state at any given time and will evolve by moving from state to state with some probability. The set of all states defines the *state space* of the system. If the future evolution of the system depends only its current state and not on its past history, then the system is said to be *memoryless*. Such a system is said to have the *markov property* and can be represented by a *Markov chain*. The state space corresponding to the single-node system in Fig. 10a can be represented by the Markov chain shown in Fig. 2b.

The Markov chain shown in Fig. 10b is a *birth-death* process for which an arrival at a node increases the number in the queue by one and a departure decreases it by one. A first-come-first-serve queue dictated by a birth-death process with a Poisson arrival rate and an exponential service-time distribution is called an *M/M/1 queue*.

Once the state space model of a system has been specified, a set of *flow balance equations* can be determined based on the balance of flow of incoming and outgoing customers for each state of the system. The flow balance equation corresponding to the state space shown in Fig. 10b is as follows:

$$(\lambda_n + \mu_n)\rho_n = \lambda_{n-1}\mu_{n-1} + \mu_{n+1}\rho_{n+1}, (n \geq 1) \tag{6}$$

and

$$\lambda_0 p_0 = \mu_0 p_1. \tag{7}$$

where $\lambda_n$ and $\rho_n$ are the arrival and departure rates in state $n$, respectively.

The flow-balance equations can be used to solve for the probability of the system being in a state, $p_n$. $p_n$ can then be used to calculate Eqs. 2 and 3.

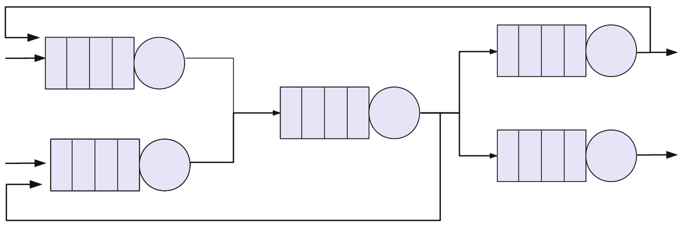


**Fig. 11** An example of a queueing network

In order to capture the behavior of the real-world problems, queueing models can consist of multiple queueing nodes connected in a networked fashion, as shown in Fig. 11. This arrangement of nodes, where customers require service at more than one service station, is referred to as a *queueing network* or a *multiple-node* system. A queueing network is called *open* when customers can enter the network from the outside and can similarly exit the network. Customers can arrive from outside the network to any node and depart from the network from any node. In contrast, a queueing network is said to be *closed* when customers can neither enter nor leave the network. The number of customers in a closed network is constant. There are many other variations of networks, the details of which are not discussed here in the interest of space.

An important measure of queueing networks is the joint probability $p_{n_1,n_2,\ldots,n_k}$, which is the probability that a network of $k$ nodes is in the state $\{n_1, n_2, \ldots, n_k\}$. As the complexity of a network increases, the ease at which this probability can be solved grows correspondingly. Jackson showed that the joint probability can be expressed as the product of its single-node station marginal probabilities [9]:

$$p_{n_1,n_2,\ldots,n_k} = \frac{1}{G}\rho_1^{n_1}\rho_2^{n_2}\cdots\rho_k^{n_k}, \tag{8}$$

where

$$G = \sum_{n_1+n_2+\cdots+n_k=N} \rho_1^{n_1}\rho_2^{n_2}\cdots\rho_k^{n_k}, \tag{9}$$

is a normalizing constant equal to 1 representing the sum of all possible joint probabilities. From this, equations similar to Eqs. 2 and 3 can be used to calculate performance numbers for the entire system.

One limitation of Jackson networks is that all customers belong to the single class and therefore have the same routing, arrival-rate, and service-rate characteristics. This is often a limiting factor when modeling real-life systems for which customers have different behaviors. BCMP networks are an extension of Jackson networks such that several classes of customer can exist in the system each with their own statistical characteristics [1]. As discussed in Sect. 4, BCMP networks are used in the proposed modeling approach to model the partial reconfiguration process for FPGAs.

## References

1. Baskett F, Chandy KM, Muntz RR, Palacios FG (1975) Open, closed, and mixed networks of queues with different classes of customers. J ACM 22:248–260. URL DOI http://doi.acm.org/10.1145/321879.321887. URL http://doi.acm.org/10.1145/321879.321887
2. Bertoli M, Casale G, Serazzi G (2009) Jmt: performance engineering tools for system modeling. SIGMETRICS Perform Eval Rev 36(4):10–15. DOI http://doi.acm.org/10.1145/1530873.1530877
3. Claus C, Ahmed R, Altenried F, Stechele W (2010) Towards rapid dynamic partial reconfiguration in video-based driver assistance systems. In: ARC, pp 55–67
4. Claus C, Zhang B, Stechele W, Braun L, Hübner M, Becker J (2008) A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. In: FPL, pp 535–538
5. Galindo J, Peskin E, Larson B, Roylance G (2008) Leveraging firmware in multichip systems to maximize fpga resources: an application of self-partial reconfiguration. In: Proceedings of the 2008 international conference on reconfigurable computing and FPGAs, pp 139–144. IEEE Computer Society, Washington, DC, USA DOI 10.1109/ReConFig.2008.81. URL http://portal.acm.org/citation.cfm?id=1494647.1495194
6. Griese B, Vonnahme E, Porrmann M, Rückert U (2004) Hardware support for dynamic reconfiguration in reconfigurable soc architectures. In: FPL, pp 842–846
7. Gross D, Harris CM (1985) Fundamentals of queueing theory, 2nd edn. Wiley, New York
8. Hsiung PA, Lin CS, Liao CF (2008) Perfecto: a systemc-based design-space exploration framework for dynamically reconfigurable architectures. ACM Trans Reconfigurable Technol Syst 1:17:1–17:30
9. Jackson JR (1957) Networks of waiting lines. Oper Res 5(4):518–521. URL http://www.jstor.org/stable/167249
10. Papadimitriou K, Dollas A, Hauck S (2011) Performance of partial reconfiguration in FPGA systems: A survey and a cost model. ACM Trans Reconfigurable Technol Syst 4:36:1–36:24. New York, NY, USA. URL http://doi.acm.org/10.1145/2068716.2068722
11. Pattipati KR, Kostreva MM, Teele JL (1990) Approximate mean value analysis algorithms for queuing networks: existence, uniqueness, and convergence results. J ACM 37:643–673. DOI http://doi.acm.org/10.1145/79147.214074. URL http://doi.acm.org/10.1145/79147.214074

# Switch Design for Soft Interconnection Networks

**Giorgos Dimitrakopoulos, Christoforos Kachris, and Emmanouil Kalligeros**

## 1 Introduction

Many-core chip multiprocessors integrate many processing cores that need a modular communication infrastructure in order to show their full potential. Scalable interconnection networks that use a network of switches connected with point-to-point links can parallelize the communication between these modules and improve performance significantly [7]. Such on-chip interconnection networks are already a mainstream technology for ASICs, while they gain significant importance in FPGA-based systems-on-chip (SOCs) [1]. The first on-chip interconnection networks mimicked the designs that were architected for large, high-performance multiprocessors. However, as interconnects migrate to the on-chip environment, constraints and trade-offs shift, and they should be appropriately adapted to the characteristics of the implementation fabric [2].

An FPGA can host two forms of interconnection networks: the soft (or overlay) interconnection networks that are statically mapped on the configurable logic of the FPGA using LUTs, registers, and RAMs, as any other ordinary circuit [5], and the dynamically reconfigurable interconnection networks that exploit the reconfigurable nature of the FPGA and allow the design of customized alternatives

G. Dimitrakopoulos (✉)
Electrical and Computer Engineering Department, Democritus University of Thrace (DUTH), Kimmeria Campus B(1.11), Xanthi, GR 67100, Greece
e-mail: dimitrak@ee.duth.gr

C. Kachris
Athens Information Technology (AIT), Athens, Greece
e-mail: kachris@ait.edu.gr

E. Kalligeros
Information and Communication Systems Engineering Department, University of the Aegean, Samos, Greece
e-mail: kalliger@aegean.gr

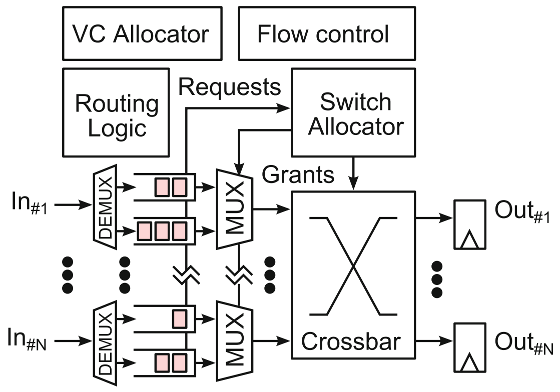


**Fig. 1** Basic building blocks of a switch

that can be adapted at runtime using both the available logic blocks and even the reconfiguration network itself [25]. In the first case, the FPGA fabric acts as an ASIC replacement that hosts a complex multiprocessor system on chip. The system consists of general-purpose soft processors, application-specific accelerators, as well as memory and I/O controllers communicating via the soft interconnection network that transfers the software-generated messages. In the second case, the system is customized for a specific set of tasks, and any application change requires its dynamic reconfiguration at the logic level [27]. The performance of the system depends on how often a reconfiguration is required and how much gain can be earned by the customization of both the processing elements and the interconnection network.

Soft interconnection networks are more generic and can support various switching technologies ranging from statically scheduled circuit-switched operation, with predetermined and prescheduled routes that avoid contention, to fully dynamic packet switching that favors statistical multiplexing on the network's links [15]. In this chapter, we focus on the dynamic approach, assuming wormhole or virtual-cut-through networks with single- or multiple-word packets. When such networks are mapped on the FPGA, a critical factor to overall system's efficiency is (a) the selection of the appropriate network topology that would reduce the communication distance and utilize efficiently the on-chip wiring resources and (b) the selection of the appropriate switch architecture that fits better to the LUT-based nature of the FPGA and offers area-delay efficient designs [16, 24]. Both factors are closely interrelated, since the reduction of the communication distance between any two nodes increases the radix (number of input and output ports) of the switches and makes their design more difficult. Concentration or the addition of express channels further increases the radix of the corresponding switches [8].

The switches of the network follow roughly the architecture depicted in Fig. 1. Incoming packets are stored in input buffers and possibly in output buffers after crossing the crossbar. Routing logic unwraps incoming packets' header and determines their output destination. The inputs that are allowed to send their data over the crossbar are determined by the switch allocator. The switch allocator accepts the requests from each input and decides which one to grant in order to produce a valid connection pattern for the crossbar. In cases that we want to differentiate between separate traffic classes, i.e., request/reply packets, and to offer deadlock-free adaptive routing, we can allow the sharing of network's channels by virtual channels (VCs) [6]. The assignment of a valid VC to a packet, before it leaves the switch, is a responsibility of the VC allocator.

While routing computation can be performed in parallel to the rest operations by employing lookahead routing [11], and VC allocation can operate in parallel to switch allocation using speculation [22], switch allocation and traversal remain closely interrelated, with switch allocation always preceding and guiding switch traversal. In fact, several designs already proved that switch allocation and traversal determine the critical path of the switch and limit any potential speed improvements [17]. So far, any innovation regarding the removal of this speed bottleneck relied mostly to architecture-level solutions that took for granted the characteristics of the allocators and the crossbar and, without any further modifications, tried to reorganize them in a more efficient way. Examples of this approach are the pipelined switch allocation and traversal units that increased the latency of the switches or prediction-based switches [18, 21]. Although such pure high-level design has produced highly efficient switches, the question on how better the switch would be if better building blocks were available remains to be investigated.

In this chapter, we try to answer this question for the case of the switch allocators and the crossbar that constitute a large part of the switch and determine the delay of the implementation. Our study will first present and compare the traditional implementations that are based on separate allocator and crossbar modules, and then will expand the design space by presenting new soft macros that can handle allocation and multiplexing concurrently. With the new macros, switch allocation and switch traversal can be performed simultaneously in the same cycle, while still offering energy-delay efficient implementations.

The rest of the chapter is organized as follows: Introductory material regarding the switch allocator alternatives at the architectural and the logic level is given in Sect. 2. Then, a review of the state-of-the-art separate arbiters and multiplexers that are used to build the switch allocator and the crossbar is presented in Sect. 3. Section 4 introduces the new merged arbiter and multiplexer module, while its efficiency relative to state-of-the-art is investigated experimentally in Sect. 5. Finally, conclusions are drawn in the last section.

## 2 Switch Allocation and Traversal

Each input of the switch can hold packets (or flits for wormhole switching) for multiple outputs. Therefore, it can request one or more outputs every cycle. For a VC-less switch that has a single FIFO queue per input, only one request per input is possible. In that case, as shown in Fig. 2, the switch allocator is constructed using a

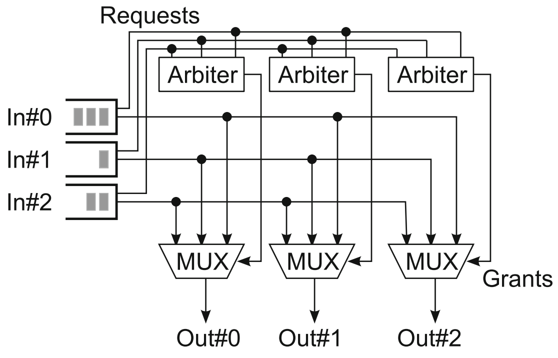


**Fig. 2** Single arbiter per output for switches with one FIFO per input

single arbiter per output of the switch, which decides independently which input to serve. The grant signals of each arbiter drive the corresponding output multiplexer, and they are given back to the inputs to acknowledge the achieved connection.

In the case of switches with VCs, the input buffers are organized in multiple-independent queues, one for each VC. Each input can send multiple requests per clock cycle. This feature complicates significantly the design of the switch allocator relative to a VC-less switch. In that case, switch allocation is organized in two phases since both per-input and per-output arbitrations are needed.[1] Even though the per-input and per-output arbiters operate independently, their eventual outcomes in switch allocation are very much dependent, each one affecting the aggregate matching quality of the switch [3, 20].

The two possible switch allocators for an $N$-input switch with $V$ virtual channels are shown in Fig. 3. In this figure, the output port requested by each VC is denoted by an $N$-bit wide one-hot coded bit vector. In the first case (Fig. 3a), each input is allowed to send to the outputs only one request. To decide which request to send, each input arbitrates locally among the requests of each VC. On the contrary, in the case of output-first allocation, all VCs are free to forward their requests to the output arbiters (Fig. 3b). In this way, it is possible that two or more VCs of the same input will receive a grant from different outputs. However, only one of them is allowed to pass its data to the crossbar. Therefore, a local arbitration needs to take place again that will resolve the conflict.

The grant signals produced by the input arbiters of an input-first switch allocator can drive the input local multiplexers in parallel to output arbitration. Therefore, when switch allocation and crossbar traversal are performed in the same cycle, this feature of input-first allocation allows some overlap in time between arbitration and

[1]An alternative to separable allocation is a centralized allocator like the wavefront allocator [26]. The main drawback of this design is the delay that grows linearly with the number of requests, while the cyclic combinational paths that are inherent to its structure cannot be handled by static timing analysis tools. The latter constraint can be removed by doubling the already aggravated delay [13].

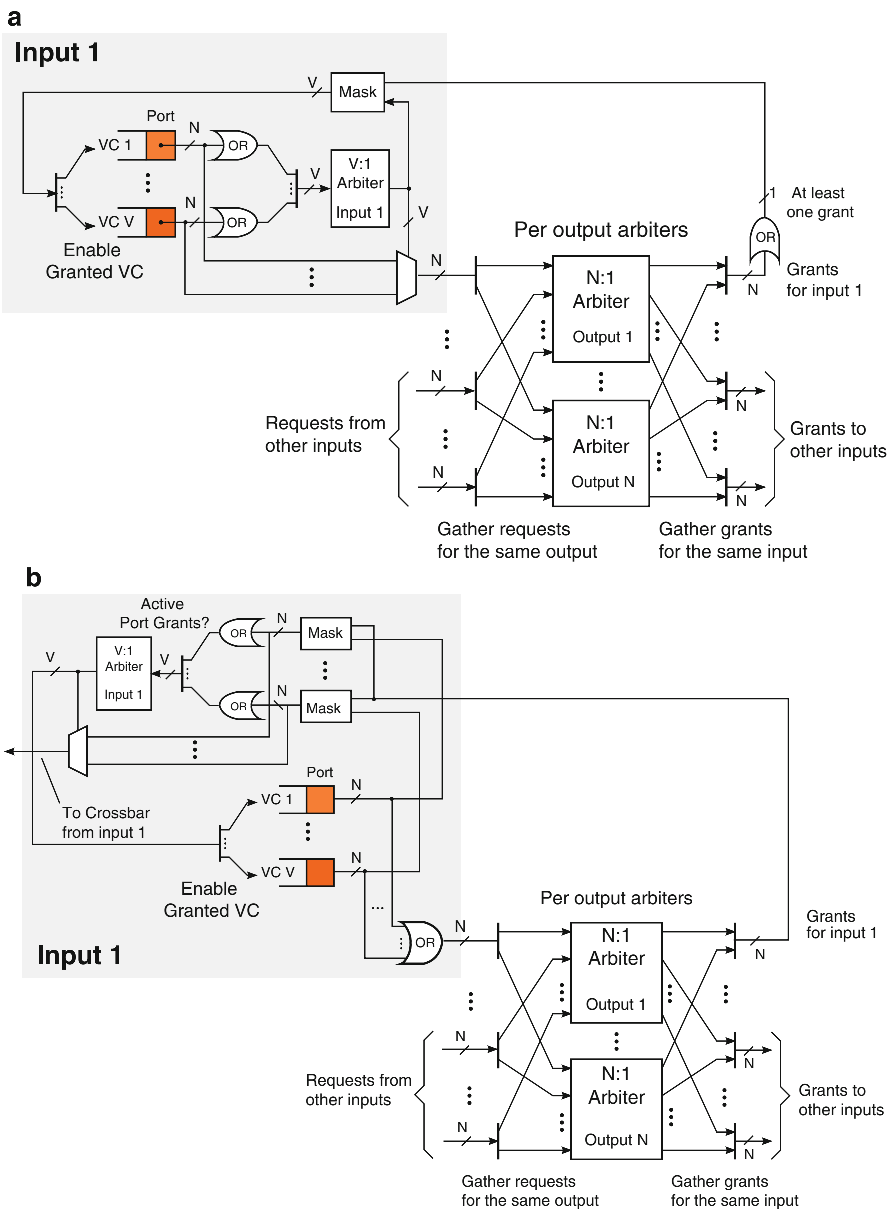


**Fig. 3** Separable switch allocation for VC-based switch: (**a**) input-first allocation, (**b**) output-first allocation

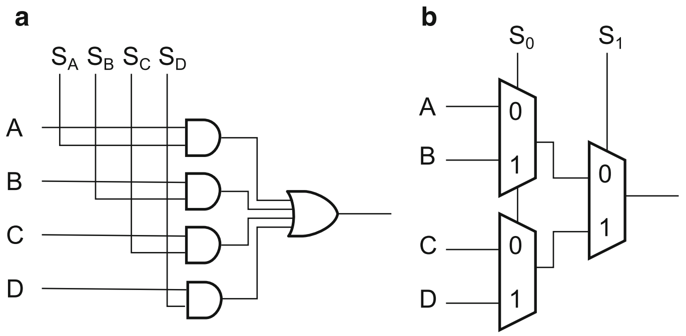


**Fig. 4** Multiplexer implementations: **(a)** AND-OR structure and **(b)** tree of smaller multiplexers

multiplexing that reduces the delay of the combined operation. Such an overlap is not possible to output-first allocation, where both stages of arbitration should be first completed before driving the multiplexers.

In every case, the kernel of switch allocation and traversal involves arbiter and multiplexer pairs that need to be carefully co-optimized in order to achieve an overall efficient implementation. For example, the encoding selected for the grant signals directly affects the design of the multiplexers. In the first case, shown in Fig. 4a, the grant decision is encoded in one-hot form, where only a single bit is set, and the multiplexer is implemented using an AND-OR structure. On the contrary, in Fig. 4b, the multiplexer is implemented as a tree of smaller multiplexers. In this case, the select lines that configure the paths of each level of the tree are encoded using weighted binary representation.

The LUT mapping of either form of multiplexers is well explored, and several optimizations have been presented in open literature. Briefly, the optimizations presented so far for the implementation of wide multiplexers on an FPGA fabric involve either the best possible packing of the multiplexer inputs and select signals in LUTs [4, 19] or the engagement of the multiplexers participating in the dedicated carry logic [28], as well as the mapping of the multiplexers on the embedded multipliers of the FPGA [14].

Even if the design choices for the multiplexer are practically limited to the alternatives shown in Fig. 4, the design space for the arbiter is larger. The arbiter, apart from resolving any conflicting requests for the same resource, it should guarantee that this resource is allocated fairly to the contenders, granting first the input with the highest priority. Therefore, for a fair allocation of the resources, we should be able to change dynamically the priority of the inputs [12]. A generic dynamic priority arbiter (DPA), as shown in Fig. 5, consists of two parts: the arbitration logic that decides which request to grant based on the current state of the priorities and the priority update logic that decides, according to the current grant vector, which inputs to promote. The priority state associated with each input may be one or more bits, depending on the complexity of the priority selection policy.

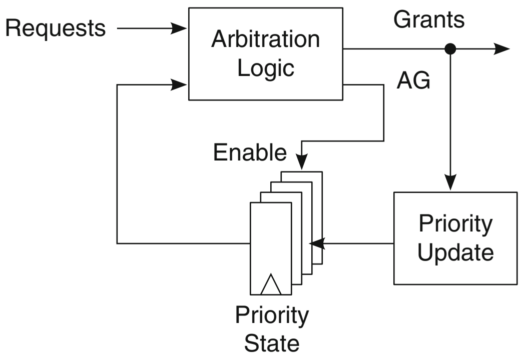


**Fig. 5** Dynamic priority arbiter

For example, a single priority bit per input suffices for round-robin policy, while for more complex weight-based policies such as first come first served (FCFS), multibit priority quantities are needed [23].

Therefore, the combined mapping of the arbiter–multiplexer pair to the programmable logic and the interconnect of the FPGA needs further exploration for unveiling the area-delay characteristics of traditional arbiter and multiplexer structures borrowed from the ASIC domain and for quantifying the potential benefits of new proposals.

## 3 Separate Arbiter and Multiplexer Design Choices

The simplest form of arbitration, called fixed-priority arbitration or priority encoding, assumes that the priorities of the inputs are statically allocated and only the relative order of the inputs' connections determines the outcome of the arbiter. In this case, the request of position 0 (rightmost) has the highest priority and the request of position $N-1$ the lowest. For example, when an 8-port fixed-priority arbiter receives the request vector $(R_7 \dots R_0) = 01100100$, it would grant input 2 since it has the rightmost active request. Two versions of an 8-port priority encoder driving a multiplexer are shown in Fig. 6. The first one involves a slow ripple-carry alternative, while the second is based on a fast parallel prefix structure.

In the case of fixed priorities, the combined operation of arbitration and multiplexing can be performed using only multiplexers. Such a structure is shown in Fig. 7. The fixed-priority order of assignment is implicitly implemented by the linear connection of the multiplexers, and thus the use of an arbiter is avoided. Despite its simplicity, the structure of Fig. 7 is only rarely used, mostly due to its increased delay.

Fixed priority is not an efficient policy, and hence it is not used in practice. On the contrary, round-robin arbitration is the most commonly used technique. Round-robin arbitration logic scans the input requests in a cyclic manner, beginning from the position that has the highest priority, and grants the first active request.

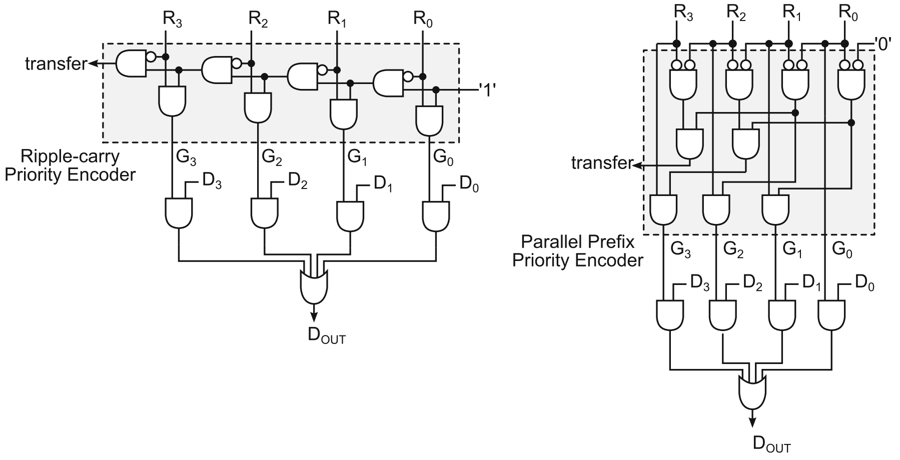
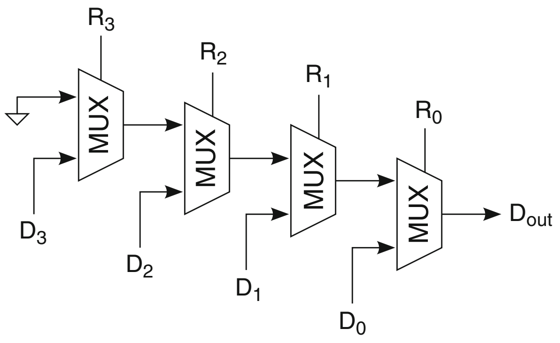

R3
R2
R1
R0
transfer
'1'
Ripple-carry Priority Encoder
G3
G2
G1
G0
D3
D2
D1
D0
DOUT
'0'
Parallel Prefix Priority Encoder

**Fig. 6** Fixed-priority arbiter driving an AND-OR multiplexer

**Fig. 7** A linear multiplexer implicitly implementing fixed priority



For the next arbitration cycle, the priority vector points to the position next to the granted input. In this way, the granted input receives the lowest priority in the next arbitration cycle. An example of the operation of a round-robin arbiter for four consecutive cycles is shown in Fig. 8 (the boxes labeled with a letter correspond to the active requests).

In the following, we will present three alternatives for the design of round-robin arbiters that are based on multiple priority encoders and on a customized carry-lookahead (CLA) structure. We focus on the implementation of the arbitration logic that scans the input requests in a cyclic manner. The design of the update logic is only briefly described since it consists of very simple modules that do not cause any timing violations. Besides, in a single cycle switch allocation and traversal, the delay of the pointer update logic is hidden, since it operates in parallel to the crossbar multiplexers.

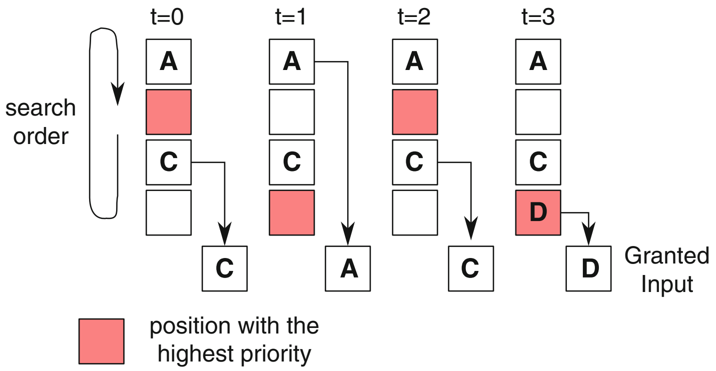
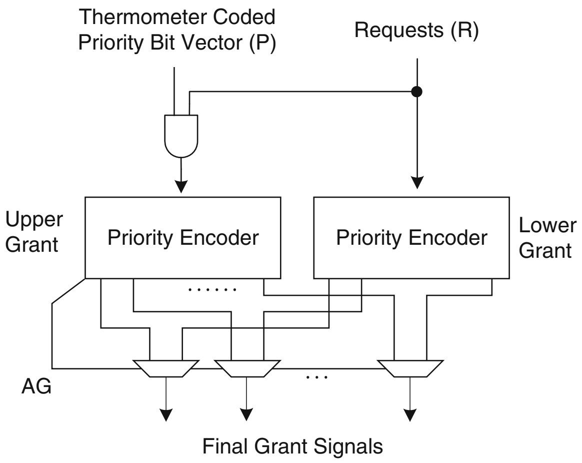


**Fig. 8** Steps of round-robin arbitration on consecutive cycles

**Fig. 9** PE-based round-robin arbiter



## *3.1 Priority Encoding-Based Round-Robin Arbiters*

The design of a priority encoding (PE)-based round-robin arbiter [12] that searches for the winning request in a circular manner beginning from the position with the highest priority involves two priority encoders, as shown in Fig. 9. In order to declare which input has the highest priority, the priority vector $P$ of the arbiter is thermometer-coded. For example, when $P = 11111000$ for an 8-input arbiter, position 3 has the highest priority. The upper PE of Fig. 9 is used to search for a winning request from the highest-priority position indexed by vector $P$ (hereafter, we will refer to this position as $P$pos), up to position $N - 1$. It does not cycle back to input 0, even if it could not find a request among the inputs $P$pos$\ldots N - 1$. In order to restrain the upper PE to search only in positions $P$pos$\ldots N - 1$, the requests it receives are masked with the thermometer-coded priority vector $P$. On the contrary, the lower PE is driven by the original request lines and searches for a winning request among all positions.

The two arbitration phases work in parallel, and only one of them has computed the correct grant vector. The selection between the two outputs is performed by employing a simple rule. If there are no requests in the range $P\text{pos}\ldots N-1$, the correct output is the same as the output of a lower PE. If there is a request in the range $P\text{pos}\ldots N-1$, then the correct output is given by the output of the upper PE. Differentiating between the two cases is performed by using the AG signal of the upper PE (AG is asserted if any input has been granted). In the following, we will refer to this architecture as the dual-path PE arbiter.

In the dual-path PE arbiter, the grant vectors follow the one-hot encoding, while the priority vector is thermometer-coded. Therefore, in order to implement correctly the round-robin pointer update policy, the grant signals should be transformed to their equivalent thermometer code. This transformation is performed inside the pointer update logic of the arbiter.

## *3.2 LZC-Based Round-Robin Arbiters*

Priority encoding identifies the position of the rightmost 1 on the request vector and keeps it alive at the output. At the same time, the remaining requests are killed, and the grant vector contains a single 1. Similarly, the process of leading-zero counting (or detection) counts the number of zeros that appear before the leftmost 1 of a word. If a transposed request vector is given to the leading-zero counter (LZC), then priority encoding and leading-zero counting are equivalent, since they both try to encode the position of the rightmost 1 in a digital word. The difference between the two methods is the encoding used to denote the selected position. In the case of priority encoding the grant vector is in one-hot form, while in the case of leading-zero counting, the output vector follows the weighted binary representation.

A round-robin arbiter that is based on LZCs can be designed by following again the dual-path approach presented in Fig. 9. The priority encoders are replaced by the corresponding LZCs that receive the requests transposed. In this case, the grant vector is composed of $\log_2 N$ bits that encode the position of the winning request, and it is connected directly to a tree of multiplexers that switch to the output the winning data, as shown in Fig. 10 (note that AZ is the All Zero signal of an LZC and is essentially the complement of the AG signal of a typical arbiter).

The most efficient LZC is presented in [10], where, for the first time, compact closed-form relations have been presented for the bits of the LZC. The iterative leading-zero counting equations can be fed directly to a logic synthesis tool and derive efficient LUT mappings. The employed LZC works in $\log_2 N$ stages, equal to the bits required for the weighted binary representation of the winning position. At each stage, the LZC computes one bit of the output by deciding, via the same operator, if the number of the leading zeros of the requests is odd or even. The first stage involves all the requests, while the following stages assume a reduced request vector. At each stage, the reduced request vector is produced by combining with an

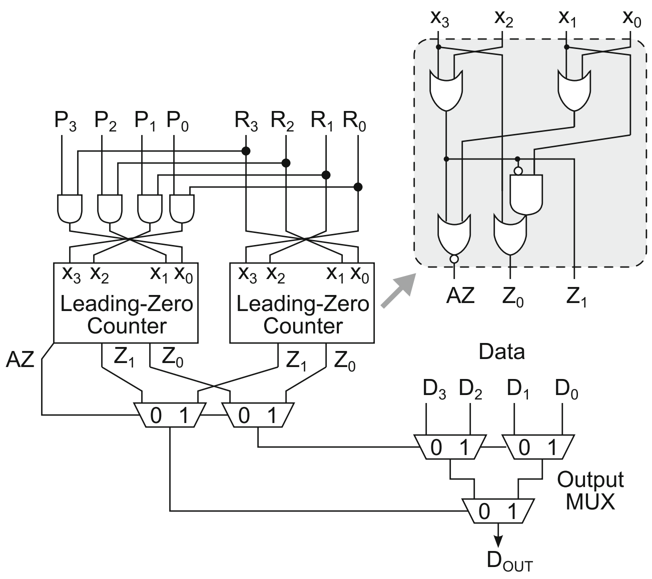


**Fig. 10** An LZC-based round-robin arbiter driving a multiplexer

OR relation the nonconsecutive pairs of bits of the previous request vector. This OR reduction is equivalent to a binary tree of OR gates.

## 3.3 *Carry-Lookahead-Based Round-Robin Arbiters*

Alternatively, a round-robin arbiter can be built using CLA structures. In this case, the highest priority is declared using a priority vector $P$ that is encoded in one-hot form. The main characteristic of the CLA-based arbiters is that they do not require multiple copies of the same circuit, since they inherently handle the circular nature of priority propagation [9]. In this case, the priority transfer to the $i$th position is modeled recursively via a priority transfer signal named $X_i$. The $i$th request gets the highest priority, i.e., $X_i = 1$, when either bit $P_i$ of the priority vector is set or when the previous position $i-1$ does not have an active request ($R_{i-1} = 0$). Transforming this rule to a boolean relation we get that

$$X_i = P_i + \overline{R}_{i-1} \cdot X_{i-1} \tag{1}$$

When $X_i = 1$ it means that the $i$th request has the highest priority to win a grant. Therefore, the grant signal $G_i$ is asserted when both $X_i$ and $R_i$ are equal to 1:

$$G_i = R_i \cdot X_i \tag{2}$$

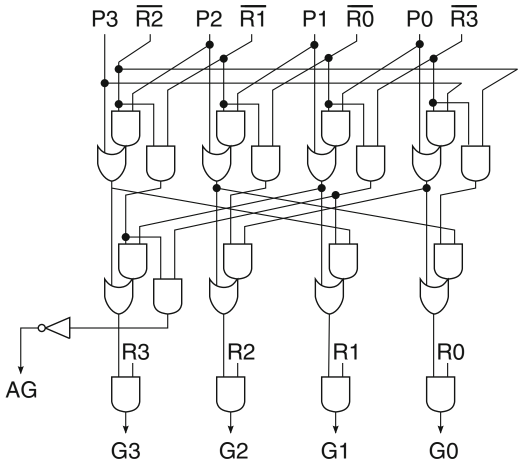


**Fig. 11** The logic-level structure of a CLA-based round-robin arbiter

The search for the winning position should be performed in a circular manner after all positions are examined. Therefore, in order to guarantee the cyclic transfer of the priority, signal $X_{N-1}$ out of the most significant position should be fed back as a carry input to position 0, i.e., $X_{-1} = X_{N-1}$. Of course, we cannot connect $X_{N-1}$ directly to position 0 since this creates a combinational loop. In [9], an alternative fast circuit has been proposed that avoids the combinational loop and computes all $X$ bits in parallel using the butterfly-like CLA structure shown in Fig. 11. Similar circuits can be derived after mapping on the FPGA the simplified and fully unrolled equations that describe the computation of the priority transfer signal $X_i$:

$$X_i = P_i + \sum_{j=0}^{n-2} \left( \prod_{k=j+1}^{n-1} \overline{R}_{|k+1|_N} \right) P_{|i+j+1|_N}, \tag{3}$$

where $|y|_N = y \mod N$. Finally, since both the grant vector and the priority vector are encoded in one-hot form, no extra translation circuit is required in the pointer update unit of this round-robin arbiter.

## 4 Merged Arbiter and Multiplexer

In this section, we present new soft macros that can handle concurrently arbitration and multiplexing. In this way, switch allocation and traversal can be performed efficiently in the same cycle, while still offering energy-delay efficient implementations. The design of these new efficient macros is based on an algorithmic transformation of round-robin arbitration to an equivalent sorting-like problem.

Similar to the PE-based round-robin arbiter of Sect. 3.1, the merged round robin arbiter and multiplexer utilizes an $N$-bit priority vector $P$ that follows the thermometer code. As shown in the example of Fig. 12, the priority vector splits

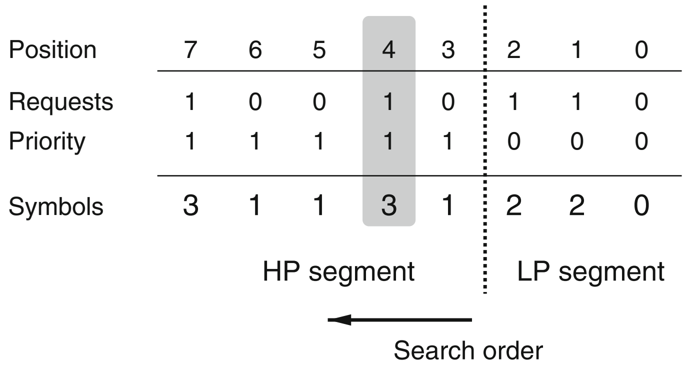


**Fig. 12** The separation of input requests to HP and LP segments according to the contents of the priority vector and the transformation of the priority and the request vectors to arithmetic symbols

the input requests in two segments. The high-priority (HP) segment consists of the requests that belong to high-priority positions where $P_i = 1$, while the requests, which are placed in positions with $P_i = 0$, belong to the low-priority (LP) segment. The operation of the arbiter is to give a grant to the first (rightmost) active request of the HP segment and, if not finding any, to give a grant to the first (rightmost) active request of the LP segment. According to the already known solutions, this operation involves, either implicitly or explicitly, a cyclic search of the requests, starting from the HP segment and continuing to the LP segment.

Either at the HP or the LP segment, the pairs of bits $(R_i, P_i)$ can assume any value. We are interested in giving an arithmetic meaning to these pairs. Therefore, we treat the bits $R_iP_i$ as a 2-bit unsigned quantity with a value equal to $2R_i + P_i$. For example, in the case of an 8-input arbiter, the arithmetic symbols we get for a randomly selected request and priority vector are shown in Fig. 12. From the four possible arithmetic symbols, i.e., 3, 2, 1, 0, the symbols that represent an active request are either 3 (from the HP segment) or 2 (from the LP segment). On the contrary, the symbols 1 and 0 denote an inactive request that belongs to the HP and the LP segment, respectively.

According to the described arbitration policy and the example priority vector of Fig. 12, the arbiter should start looking for an active request from position 3, moving upwards to positions 4, 5, 6, 7, and then to 0, 1, 2 until it finds the first active request. The request that should be granted lies in position 4, which is the first (rightmost) request of the HP segment. Since this request belongs to the HP segment, its corresponding arithmetic symbol is equal to 3. Therefore, granting the first (rightmost) request of the HP segment is equivalent to giving a grant to the first maximum symbol that we find when searching from right to left. This general principle also holds for the case that the HP segment does not contain any active request. Then, all arithmetic symbols of the HP segment would be equal to 1, and any active request of the LP segment would be mapped to a larger number (arithmetic symbol 2).

Therefore, by treating the request and the priority bits as arithmetic symbols, we can transform the round-robin cyclic search to the equivalent operation of selecting the maximum arithmetic symbol that lies in the rightmost position. Searching for the maximum symbol and reporting at the output only its first (rightmost) appearance, implicitly implements the cyclic transfer of the priority from the HP to the LP segment, without requiring any true cycle in the circuit. In principle, any maximum selector does not contain any cycle paths and is built using a tree or a linear comparison structure. The proposed arbiter is built using a set of small comparison nodes. Each node receives two arithmetic symbols, one coming from the left and one from the right side. The maximum of the two symbols under comparison appears at the output of each node. Also, each node generates one additional control flag that denotes if the left or the right symbol has won, i.e., it was the largest. In case of a tie, when equal symbols are compared, this flag always points to the right. In this way, the first (rightmost) symbol is propagated to the output as dictated by the operation of the arbiter.

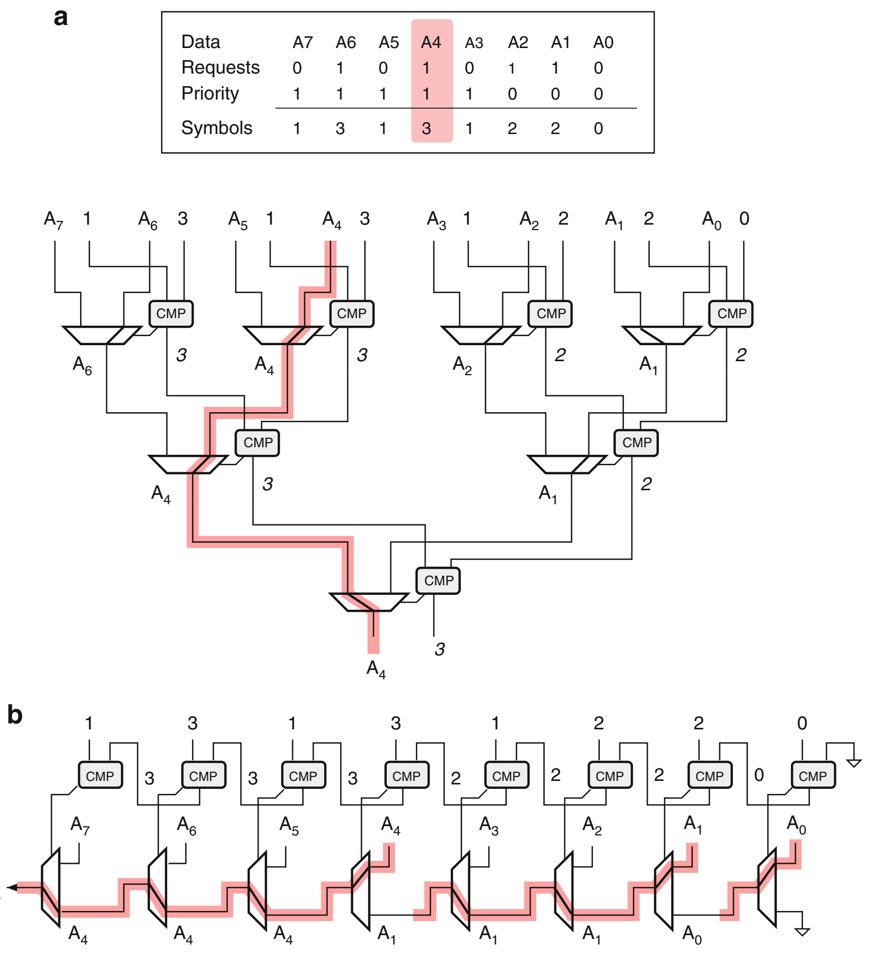
In every case, the winning path is clearly defined by the direction flags produced by the comparison nodes. Thus, if we use these flags to switch the data words that are associated with the corresponding arithmetic symbols, we can route at the output the data word that is associated with the winning request. This combined operation can be implemented by adding a 2-to-1 multiplexer next to each comparison node and connecting the direction flag to the select line of the multiplexer. The structure of both a binary tree and a linear merged arbiter multiplexer with 8 inputs, along with a running example of their operation, is shown in Fig. 13. Following the example, we observe that the first, in a round-robin order, data word $A_4$ is correctly routed at the output.

Although the tree-structured merged arbiter multiplexer has smaller delay than the linear-structured one, the latter can take advantage of the dedicated mux-carry logic of the FPGA.

## *4.1 Computation of the Grant Signals*

The merged arbiter multiplexer, besides transferring at the output the data word of the granted input, should also return in a useful format the position of the winning request (or equivalently the grant index). The proposed maximum-selection tree, shown in Fig. 13a, that replaces the traditional round-robin arbiter can be enhanced to facilitate the simultaneous generation of the corresponding grant signals via the flag bits of the CMP nodes.

At first, we deal with the case that the grants are encoded in weighted binary representation. In this case, we can observe that, by construction, the weighted binary encoding of the winning request is formed by putting together the flag bits of the CMP nodes that lie in the path from the winning input to the root of the tree (see Fig. 14a). Consequently, the generation of the grant signals in weighted binary representation is done by combining at each level of tree the winning flag bits of



| Data | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|---|
| Requests | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| Priority | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Symbols | 1 | 3 | 1 | 3 | 1 | 2 | 2 | 0 |

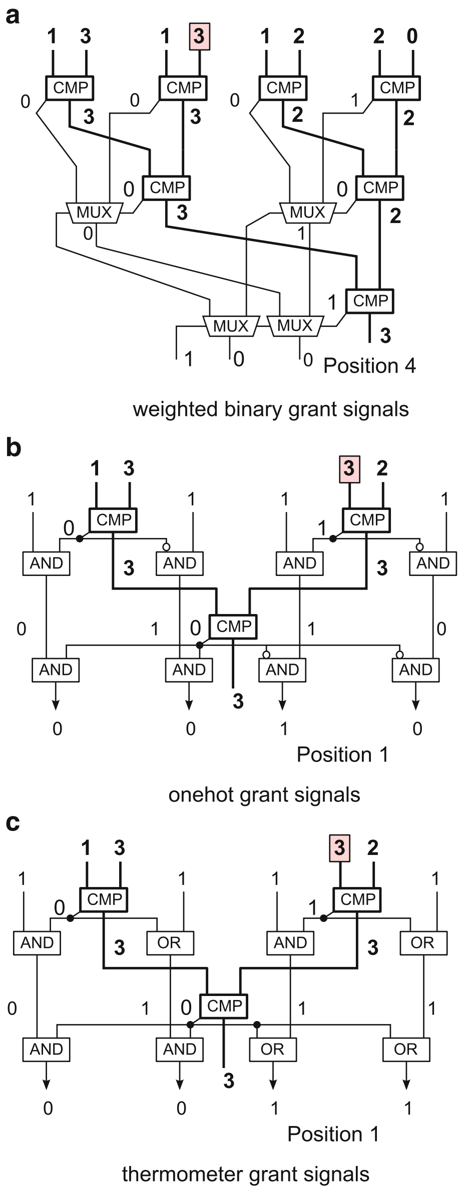
a
$A_7$ 1 $A_6$ 3 $A_5$ 1 $A_4$ 3 $A_3$ 1 $A_2$ 2 $A_1$ 2 $A_0$ 0
CMP
$A_6$ 3 $A_4$ 3 $A_2$ 2 $A_1$ 2
$A_4$ 3 $A_1$ 2
$A_4$ 3
b
1 3 1 3 1 2 2 0
3 3 3 2 2 2 0
$A_7$ $A_6$ $A_5$ $A_4$ $A_3$ $A_2$ $A_1$ $A_0$
A 4
$A_4$ $A_4$ $A_4$ $A_1$ $A_1$ $A_1$ $A_0$

**Fig. 13** The merged arbiter multiplexer: (**a**) Tree and (**b**) linear comparison structure

the previous levels with the flags of the current level. This is achieved by means of some additional multiplexers, as shown in Fig. 14a, for the case of a tree-based merged arbiter multiplexer.

For the one-hot encoding, we need a different implementation. Initially, i.e., at the inputs of the one-hot grant generation circuit, we assume that every position has a grant signal equal to 1. At the following levels, some of these grant signals are transformed to 0s if their associated symbols are not the winning ones at the corresponding CMP nodes. Thus, at the output, only a single 1 will remain and the rest would be nullified. The circuit that generates the corresponding grant signals in one-hot form, for 4 input symbols, is shown in Fig. 14b. Keeping and nullifying the grant signals is performed by the AND gates that mask, at each level of the

**Fig. 14** The grant generation circuits that run in parallel to the CMP nodes for a tree organization of the merged arbiter multiplexer



tree, the intermediate grant vector of the previous level with the associated direction flags. The inversions are needed to keep alive the grant signals that correspond to a winning symbol of the right subtrees. Observe that, if we replace the invert-AND gates of Fig. 14b with OR gates, the outcome would be a thermometer-coded grant vector instead of the one-hot code. The resulting circuit is shown in Fig. 14c.

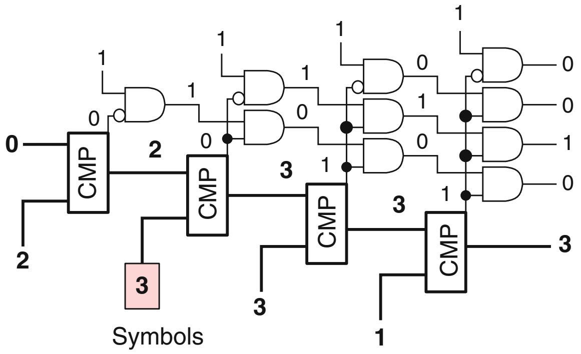


**Fig. 15** The one-hot grant generation circuit for the linear organization of the merged arbiter multiplexer

In this way, with minimum cost, we are able to fully cover all possible useful grant encodings, thus alleviating the need for additional translation circuits.

When the linear comparison structure is selected for the organization of the merged arbiter multiplexer, we can design the grant generation circuits following a similar procedure. A one-hot grant generation circuit for the case of a 4-input merged arbiter multiplexer is shown in Fig. 15. The AND gates at each comparison stage are driven by the direction flags of the CMP nodes and a constant 1 that allows us to simplify the invert-AND gates to inverters. Again, if the invert-AND gates are replaced by OR gates, a thermometer code word can be derived for the grant signals.

As for the weighted binary grant generation circuit, we can use the same linear structure of Fig. 13b, replacing the data words that drive the multiplexers with the various position indices in weighted binary format (i.e., in Fig. 13b, $A_0$ is replaced by 000, $A_1$ by 001, etc.). This means that, in this case, two multiplexers are needed at each stage of the linear structure, one for switching the data and another for switching the weighted binary indices.

When there is no active request, the arbiter should deassert the AG signal. This case is identified by observing the symbol at the output of the comparison structure. When it is equal to either 0 or 1, it means that no active request exists in either priority segment.

### 4.2 Switches with Merged Arbiter-Multiplexer Structures

The design of switches that use the proposed round-robin merged arbiter multiplexers (MARX) is straightforward. Figure 16a depicts the design of a VC-less switch using the proposed macros. This is the simplest case, where the arbiter-multiplexer pairs that existed at each output are directly replaced by the proposed units. As in any switch, the data placed on the input registers or the head of the input queues should not be changed or dequeued until the corresponding input is granted access to the requested output port.

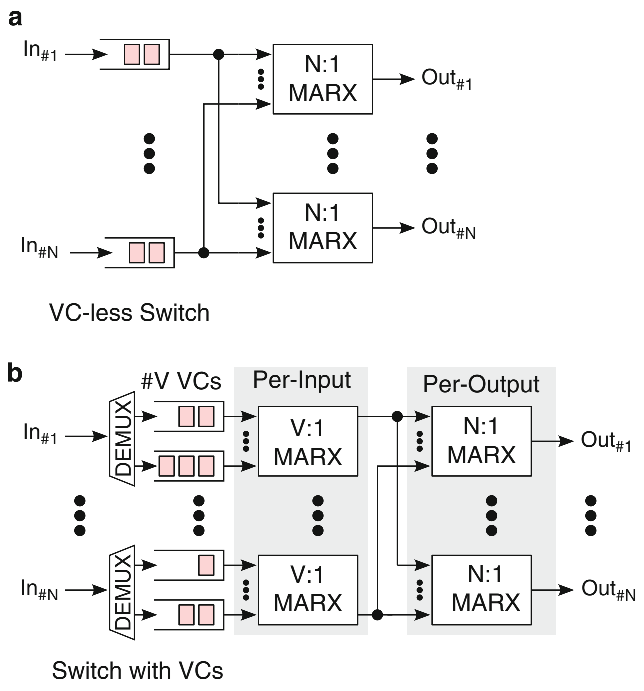
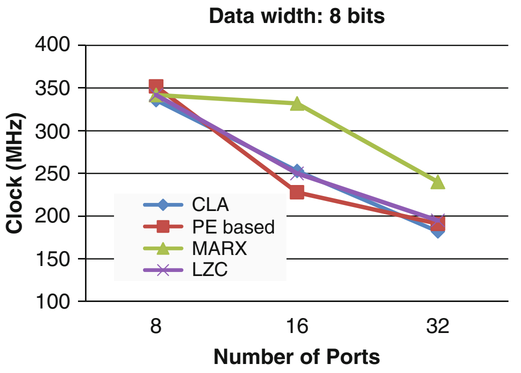


**Fig. 16** Switches built with MARX units: **(a)** VC-less wormhole switch and **(b)** VC-based switch

In the case of switches with VCs, the design is more complicated due to the per-input and per-output stages of arbitration and multiplexing. The proposed macros fit better in the case of input-first allocation. This organization is shown in Fig. 16b. The per-input MARX units select locally an eligible VC among the V available and carry along its corresponding flit. The VCs selected from each input compete for gaining access to their requested outputs via the per-output MARX units that simultaneously resolve any conflicts and give at the output the flit of the winning VC.

## 5 Experimental Results

In this section, we explore the implementation characteristics of the presented designs. The analysis that follows aims to identify the fastest and/or the most area-efficient alternative by varying the number of ports of the arbiter and multiplexer and the data width of each port. For attaining our comparison data, we first generated the equivalent VHDL descriptions of all designs under comparison. After extensive simulations that verified the correctness of each description, each design was synthesized and mapped to a Virtex-5 XC5VLX330 FPGA chip. For the synthesis, mapping, and placement and routing of the designs, we used the ISE 12.2 toolset



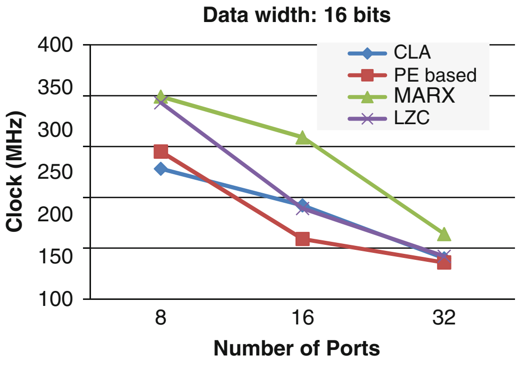
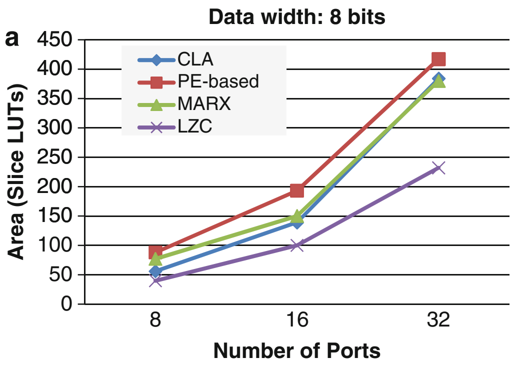


**Fig. 17** The delay of arbiters and multiplexers varying the number of ports for 8 and 16 bits port width

of Xilinx. Please note that the reported results involve only the optimizations performed by the CAD tools alone, without any manual intervention that would further optimize the circuits under comparison. In this way, the presented results can be reproduced by every designer by just following the same automated design flow.

At first, we compare the presented design alternatives in terms of delay. Delay is critically affected by the number of ports that the circuit is designed to serve, as well as the width of the corresponding data words that increases the loading of the multiplexers' control signals. The best delays achieved for each circuit after varying the number of inputs and keeping constant the data width to 8 and 16 bits are shown in Fig. 17. From the presented results that were measured after place and route, we can draw several conclusions. The merged arbiter multiplexer (we refer to the tree-structured implementation) is, in all cases, the fastest, and the delay savings are more than 20 % on average. This trend is followed irrespective of the number of bits used per multiplexer port. The observed delay convergence, when the number of ports increases, is attributed to the aggravated effect of routing interconnect delay that constitutes more than 80 % of the total path delay. This is a sign that such wide multiplexers of single-stage switching systems should be avoided, and



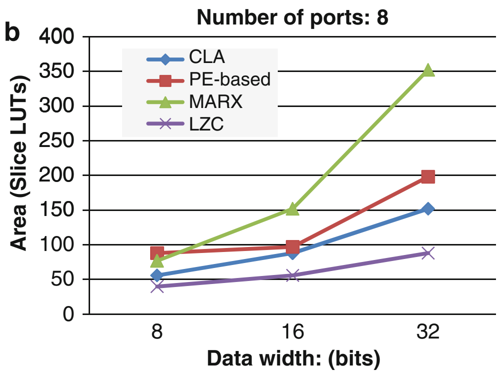


**Fig. 18** The area of the arbiters and multiplexers when varying (**a**) the number of ports at a constant port width of 8 bits and (**b**) the width of each port of an 8-input circuit

the communication among multiple modules should be organized as a network of switches. Observe though that, even for such extreme multiplexer widths, the delay advantage of MARX is considerable.

The area of the examined designs is reported in Fig. 18. Specifically, Fig. 18a reports the area occupied by the compared designs assuming 8 bits per port and varying the number of ports. On the other hand, Fig. 18b shows the area of all the designs for an 8-input arbiter and multiplexer when varying the width of each port. The most clear conclusion derived from both figures is that the LZC-based arbiter and multiplexer is the most area-efficient solution requiring roughly 30 % less area on average. On the contrary, the fastest design, i.e., the merged arbiter and multiplexer, although it behaves similarly to the other designs for small port widths, requires significantly more area when the bits per port are increased to 16 and 32 bits. This behavior though enables the designer to explore the area-delay trade-off; the MARX allows for very fast implementations with the overhead of extra area for increased data widths, whereas the LZC-based design offers low implementation cost with fairly small delays.

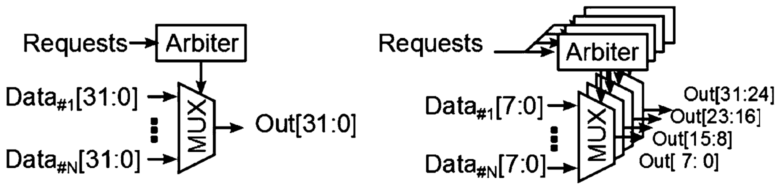


**Fig. 19** The application of bit slicing to arbiter and multiplexer pairs

## *5.1 Bit Slicing*

The multiplexer can be sliced to smaller multiplexers with the same number of input ports but of smaller data width. This operation is equivalent to spreading parts of input words to multiple smaller multiplexers, where each multiplexer is driven by a dedicated arbiter. As shown in Fig. 19, the control logic of the independent multiplexers remains unified, and each submultiplexer receives the same grant decisions. This happens since all arbiters work in sync, receiving the same requests and updating, in the same way, the priority of each position. Bit slicing partially alleviates the high-fanout problem of the grant signals and may offer higher-speed solutions. In reality, the fanout taken from the grant signals is given to the request lines that now need to be broadcasted to more arbiters.

In the following, we investigate the delay benefits of bit slicing and try to identify which slicing factor is the best for the designs under comparison. We applied bit slicing on an 8-input and a 16-input arbiter and multiplexer carrying data of 32 bits. We used all power-of-two slicing factors SF between the two extremes: SF = 1 that corresponds to no slicing and SF = 32 that corresponds to full slicing, where each bit has its own arbiter. In the general case, bit slicing by a factor SF means that we implement SF multiplexer and arbiter pairs, with each multiplexer carrying 32/SF bits. The obtained results that clearly depict an optimum slicing factor for each design are shown in Fig. 20.

## 6 Summary and Conclusions

In this chapter we presented and compared various alternatives for the design of an arbiter and a multiplexer in an FPGA. The design space includes traditional separate arbiter and multiplexer pairs, as well as recently introduced merged arbiter and multiplexer macros that handle arbitration and multiplexing concurrently. Although the mapping of multiplexers in LUT logic has received a lot of attention in the

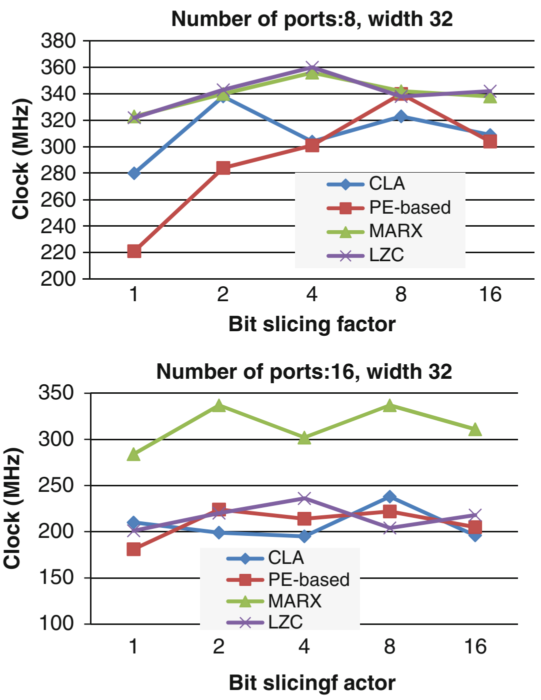


**Fig. 20** The delay of all 8-input and 16-input circuits for all power-of-two bit slicing factors

previous years, the combined implementation of an arbiter and a multiplexer was partially unexplored. This work covers this gap and extends, at the same time, the design space with new efficient solutions that simplify the design of high-radix soft switches.

## References

1. Altera (2011) Applying the benefits of network on a chip architecture to FPGA system design. Tech. Rep., Jan 2011
2. Azimi M, Dai D, Mejia A, Park D, Saharoy R, Vaidya AS (2009) Flexible and adaptive on-chip interconnect for terascale architectures. Intel Tech J 13(4):62–77
3. Becker DU, Dally WJ (2009) Allocator implementations for network-on-chip routers. In: Proceedings of the ACM/IEEEE international supercomputing conference, 2009
4. Bhatti NK, Shingal N (2009) LUT based multiplexers. US Patent 7,486,110, Feb 2009
5. Brebner G, Levi D (2003) Networking on chip with platform FPGAs. In: Proceedings of the IEEE international conference on field-programmable technology, Dec 2003, pp 13–20
6. Dally WJ (1990) Virtual-channel flow control. In: Proceedings of the 17th annual international symposium computer architecture (ISCA), May 1990, pp 60–68
7. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Proceedings of the 38th design automation conference (DAC), June 2001, pp 684–689
8. Dally WJ, Towles B (2004) Principles and practices of interconnection networks. Morgan Kauffman, San Francisco

9. Dimitrakopoulos G, Chrysos N, Galanopoulos C (2008) Fast arbiters for on-chip network switches. In: IEEE international conference on computer design (ICCD), 2008, pp 664–670
10. Dimitrakopoulos G, Galanopoulos K, Mavrokefalidis C, Nikolos D (2008) Low-power leading-zero counting and anticipation logic for high-speed floating point units. IEEE Transactions on very large scale integration (VLSI) Systems (16)7:837–850
11. Galles M (1997) Spider: a high-speed network interconnect. IEEE Micro 17(1):34–39
12. Gupta P, McKeown N (1999) Designing and implementing a fast crossbar scheduler. IEEE Micro 19(1):20–28
13. Hurt J, May A, Zhu X, Lin B (1999) Design and implementation of high-speed symmetric crossbar schedulers. In: IEEE international conference on communications (ICC), June 1999, pp 253–258
14. Jamieson P, Rose J (2005) Mapping multiplexers onto hard multipliers in FPGAs. In: Proceedings of IEEE NewCAS conference, June 2005, pp 323–326
15. Kapre N, Mehta N, Delorimier M, Rubin R, Barnor H, Wilson MJ, Wrighton M, Dehon A (2006) Packet-switched vs. time-multiplexed FPGA overlay networks. In: Proceedings of the IEEE symposium on field-programmable custom computing machines, 2006, pp 205–215
16. Lee J, Shannon L (2010) Predicting the performance of application specific NoCs implemented on FPGAs. In: Proceedings of the 18th annual ACM/SIGDA international symposium on field programmable gate arrays - FPGA 10. ACM Press, New York, 2010, pp 23–32
17. Lu Y, McCanny J, Sezer S (2011) Generic low-latency NoC router architecture for FPGA computing systems, fpl, In: 21st international conference on field programmable logic and applications, 2011, pp 82–89
18. Matsutani H, Koibuchi M, Amano H, Yoshinaga T (2009) Prediction router: yet another low latency on-chip router architecture. In: Proceedings of the 15th IEEE international symposium on high-performance computer architecture (HPCA), Feb. 2009, pp 367–378
19. Metzgen P, Nancekievill D (2005) Multiplexer restructuring for FPGA implementation cost reduction. In: Proceedings of the 42nd design automation conference, 2005, pp 421–426
20. Mukherjee SS, Silla F, Bannon P, Emer JS, Lang S, Webb D (2002) A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In: Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X), 2002, pp 223–234
21. Mullins RD, West AF, Moore SW (2004) Low-latency virtual-channel routers for on-chip networks. In: Proceedings of the 31st annual international symposium on computer architecture (ISCA), 2004, pp 188–197
22. Peh L.-S, Dally WJ (2001) A delay model and speculative architecture for pipelined routers. In: Proceedings of the 7th international symposium on high-performance computer architecture (HPCA-7), 2001
23. Pirvu M, Bhuyan L, Ni N (1999) The impact of link arbitration on switch performance. In: Proceedings of the 5th high-performance computer architecture (HPCA), 1999, pp 228–235
24. Saldana M, Shannon L, Craig J, Chow P (2007) Routability of network topologies in FPGAs. IEEE Transactions on very large scale integration (VLSI) Systems 15(8):948–951
25. Shelburne M, Patterson C, Athanas P, Jones M, Martin B, Fong R (2008) MetaWire: using FPGA configuration circuitry to emulate a network-on-chip. In: Proceedings of the 2008 international conference on field programmable logic and applications, Sept 2008, pp 257–262
26. Tamir Y, Chi H.-C (1993) Symmetric crossbar arbiters for VLSI communication switches. IEEE Trans Parallel Distr Syst 4(1):13–27
27. Vassiliadis S, Sourdis I (2007) FLUX interconnection networks on demand. J Syst Architect 53(10):777–793
28. Wittig RD, Mohan S (2003) Method for implementing large multiplexers with FPGA lookup tables. US Patent 6,505,337 B1, Jan 2003

# Embedded Systems Start-Up Under Timing Constraints on Modern FPGAs

**Joachim Meyer, Juanjo Noguera, Michael Hübner, Rodney Stewart, and Jürgen Becker**

## 1 Introduction

The continuous improvements in field-programmable gate arrays (FPGA) have made possible their increasing deployment in modern embedded systems [9]. The trend is to combine microprocessors with reconfigurable hardware, either on one printed circuit board basis or even on a chip basis in Zynq devices [21]. This enables a designer to partition an application into control flow dominant parts, suitable for a microprocessor and data flow dominant parts, suitable for a dedicated hardware implementation. This *hardware/software co-design* is an enormous help to meet performance, cost, and reliability goals.

Whenever there is no hard-coded microprocessor available inside an FPGA, but an additional microprocessor is too expensive, you can use soft-core microprocessors like the MicroBlaze [20]. Even with small FPGAs, it is possible to integrate in the same device hardware accelerators with such microprocessors, running full operating systems (e.g., Linux). On the other hand, many embedded systems nowadays have to meet extremely tight *start-up timing specifications*, that is, time the electronic system has to be operative after power-up. Examples of electronic systems with a start-up timing specification are PCI Express systems or CAN-based electronic control units (ECU) in automotive applications. In both examples, the electronic system has to be up and running within 100 ms after system power-up.

J. Meyer (✉) • M. Hübner • J. Becker
Karlsruhe Institute of Technology, Engesserstraße 5, 76131 Karlsruhe, Germany
e-mail: Joachim.Meyer@KIT.edu; Michael.Huebner@KIT.edu; Becker@KIT.edu

J. Noguera
Xilinx Inc., Dublin, Ireland
e-mail: Juanjo.Noguera@Xilinx.com

R. Stewart
Xilinx Inc., Killarney, Ireland
e-mail: Rodney.Stewart@Xilinx.com

**Table 1** Maximal bitstream sizes of different Xilinx FPGA families

| FPGA family | Release year[a] | Biggest device | Maximum bitstream size |
|---|---|---|---|
| Virtex II Pro (X) | 2002 | XC2VP100 | 4,0 MB |
| Virtex 4 | 2004 | XC4VLX200 | 6.1 MB |
| Virtex 5 | 2006 | XC5VLX330T | 9.8 MB |
| Virtex 6 | 2009 | XC6VLX760 | 22 MB |
| Virtex 7 | 2011 | XC7V2000T | 53 MB |

[a]Initial release of documentation

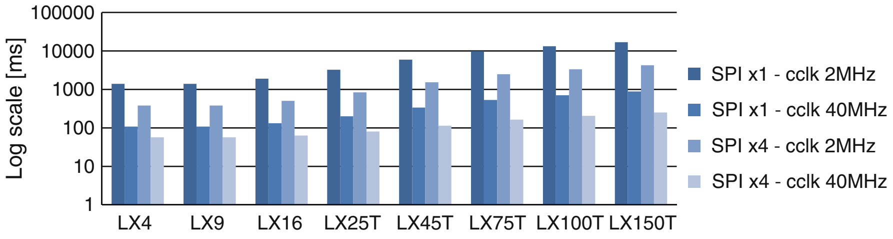


**Fig. 1** Calculated worst case configuration times for Spartan-6 devices

Otherwise, in the case of PCI Express, the system will not be recognized by the root complex [10], or the system might miss important communication messages in the case of CAN-based automotive ECUs.

The start-up process of an embedded system on a FPGA can be divided in two steps: (1) an FPGA configuration and (2) an embedded software boot process. Due to a continuously increasing amount of FPGA resources, the bitstream size of FPGAs and thus the configuration time grow as well; see Table 1. Therefore, even with medium-sized FPGAs, it is not possible to meet the start-up timing specification using low-cost configuration solutions. Figure 1 shows the calculated worst case configuration time for different Spartan-6 FPGAs using the low-cost SPI/Quad-SPI configuration interface. Even when using a fast configuration solution (i.e., Quad-SPI at 40 MHz clock), only the small FPGAs could meet the 100 ms start-up timing specification.

But optimizing the configuration process with concepts like [6] is just one half of the job. Especially for FPGA-based systems it is also very important to use an efficient software boot process in order to deal with the additional configuration time. Therefore, if an embedded system is implemented on a medium-/large-sized FPGA and it requires complex software (e.g., footprint size), it becomes extremely challenging to meet start-up timing requirements, like 100 ms, by using traditional system start-up concepts, methods and tool flows.

The following pages present novel concepts, methods, and tool flows for embedded systems start-up on modern FPGAs under tight timing constraints. A two-stage embedded system start-up concept is introduced. In this approach, the system starts up first with a minimal subsystem. While this approach enables a fast configuration of *timing-critical* hardware, it furthermore enables several significant optimizations in the concept of the software boot process, so the embedded system meets the hard deadline. After this initial subsystem is up and running, the *non-timing-critical* hardware and software components are loaded.

The content of this chapter can be summarized as follows:

- Techniques, methods, and tool flows that enable an FPGA-based embedded system to start up under hard timing constraints
- A software partitioning mechanism, enabling an efficient exploitation of different memory architectures on the FPGA-based system, in order to optimize the boot process
- Implementation of an application case study in the automotive domain implemented on leading-edge FPGAs (i.e., Spartan-6), which demonstrate the improvements over traditional methods and techniques

This chapter is structured in four additional sections. Next section provides an overview of the related work. Section 3 introduces the novel methods and techniques to reduce FPGA configuration time as well as software start-up time. Section 4 presents the results obtained when implementing an automotive case study on a Spartan-6 device. Finally, Sect. 5 gives the conclusion.

## 2 Related Work

Several research groups have proposed new approaches to analyze and reduce the start-up time of embedded systems based on flash memory. Since the bottleneck is usually given by the relative slow access to the flash memory, the common strategy here is to minimize or speed up any access to this memory during the start-up of the system.

A recent research activity dealing with boot time for flash-based systems is, for example, [14] which describes an approach to provide time-predictable system initialization for flash-memory embedded systems. In addition, a method to mount a file system from flash memory instantly is presented in [22].

For complex operating systems like Linux, there are many parameters you can optimize. The work introduced in [2] attempts to analyze the start-up time of the boot loader and Linux kernel and to compare the performance of several root file systems.

Another possibility to speed up the boot time is to split and run the software from different memory levels [1]. For example, reading the read-only data directly out of flash memory instead of copying it to a random access memory saves time in copying data by the boot loader.

However, none of the previously mentioned efforts considered start-up requirements for FPGA-based systems. For such systems, the configuration time of SRAM-based FPGAs is usually a major part of the system start-up, especially for modern FPGAs with millions of resources. While techniques like configuration caching [4, 8] and configuration prefetching [5] can be used to optimize the reconfiguration overhead of an FPGA only, a technique which is actually able to speed up the initial configuration and thus the start-up time of an FPGA is the compression of the configuration data [3, 7, 13]. In this approach several compression algorithms have been analyzed and successfully used in order to reduce the amount of configuration data which has to be transferred into the reconfiguration fabric of the FPGA.

Also, for FPGA-based embedded systems, hibernating concepts have been used to reduce the long initialization time of complex operating systems [11]. Although this is very helpful for complex operating systems, for small and simple operating systems, the major part of the start-up time is due to moving the software code from the slow nonvolatile storage to the fast but volatile random access memory, not to mention the additional FPGA configuration time.

In [16] dynamic partial reconfiguration together with bitstream compression was used in order to significantly reduce configuration time and to meet the timing deadline for PCIe. While for the targeted scenario the design meets the deadline, the approach still includes an initial configuration of all FPGA resources. Although using compression, this is a redundant overhead which grows with the size of the FPGA device.

The work proposed in this chapter concentrates on exploiting the FPGA flexibility (i.e., run-time programmability) to meet the hard deadline of an embedded system. The flexibility of FPGAs allows adapting the hardware to the requirement of the software in order to meet the deadline (e.g., implement the most appropriate memory hierarchy). This FPGA programmability offers new challenges and opportunities, not available in non-reconfigurable embedded systems, which are addressed in this paper.

## 3 Embedded Systems Start-Up Under Timing Constraints on FPGAs

The start-up process of an embedded system on a FPGA can be divided into two parts: (1) the hardware start-up time and (2) the embedded software start-up time. While for non-reconfigurable systems the hardware start-up time is rather negligible, for FPGAs this can easily become the major part due to the additional time required to configure the design on the FPGA.

This FPGA configuration time highly depends on two parameters, namely the FPGA size and therefore the size of the configuration bitstream and the type of the configuration interface which is used to load the configuration bitstream into the FPGA. Modern FPGAs support several types of configuration interfaces (e.g., low-cost serial interfaces or high-performance parallel interface). Those interfaces differ not only in bus width but also in the maximum configuration clock frequency.

Once the FPGA is configured, the next step is to start up the embedded software running on the processor implemented in the FPGA. The time which is needed in order to start the software is composed by the time to get the software in the right

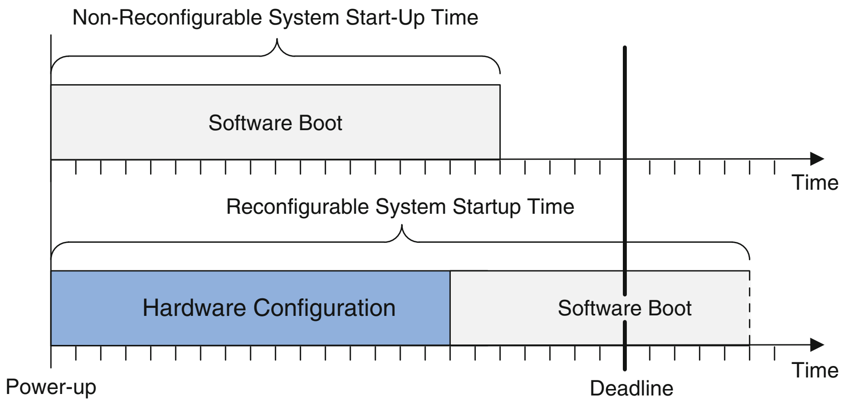

Non-Reconfigurable System Start-Up Time
Software Boot
Time
Reconfigurable System Startup Time
Hardware Configuration
Software Boot
Time
Power-up
Deadline

**Fig. 2** Due to the additional configuration time, reconfigurable systems often fail to boot their software in time to meet tight deadlines

status to be executed (e.g., move it to main memory) and the time the software needs to initialize the system. Depending on the complexity (i.e., size) of this embedded software, even if the hardware configuration meets the application deadline, the system start-up time might exceed it, as shown in Fig. 2.

## *3.1 Reducing FPGA Configuration Time*

Traditional configuration mechanisms for FPGAs initially configure the *complete* FPGA device at power-up. If fast FPGA configuration is needed, it is possible to use high-speed parallel interfaces for accessing the configuration data. Since those options are expensive, they are not acceptable in many price-sensitive embedded applications, like automotive. In these low-cost scenarios, an external SPI serial flash memory is used.

The FPGA configuration time highly depends on the amount of FPGA configuration data (i.e., configuration *bitstream* size). The larger the FPGA, the more time it will take to configure the device. To address this issue, a new configuration technique named *fast FPGA configuration* is proposed to reduce the initial FPGA configuration time. The key concept is to *initially configure only a small region of the FPGA at power-up*. That is, the FPGA device is not completely configured at power-up, which is the traditional case. This small FPGA region should include all timing-critical hardware components which should meet the hard start-up deadline. After this initial small configuration is finished, the design is fully operational and running on the FPGA. Afterwards, dynamic partial reconfiguration is used in order to configure the remaining region of the FPGA, which was not configured at power-up. These additional *non-timing-critical* hardware components provide additional functionality that will be required during application operation mode.

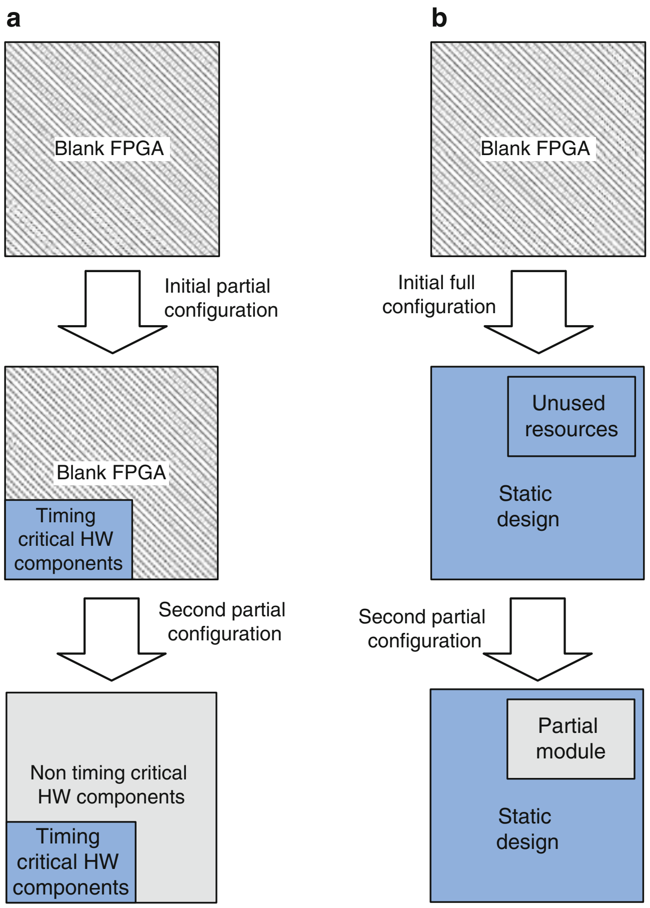

a
Blank FPGA
Initial partial configuration
Blank FPGA
Timing critical HW components
Second partial configuration
Non timing critical HW components
Timing critical HW components
b
Blank FPGA
Initial full configuration
Unused resources
Static design
Second partial configuration
Partial module
Static design

**Fig. 3** Comparison of the concept for *fast FPGA configuration* (**a**) and the traditional dynamic partial reconfiguration concept (**b**)

Although dynamic partial reconfiguration is exploited by the fast FPGA configuration, there are differences between the traditional usage of the dynamic partial reconfiguration; see Fig. 3b and the concept of the fast FPGA configuration; see Fig. 3a. While the concept of dynamic partial reconfiguration intends a full design to be used as initial configuration which can be modified during run time, the fast configuration technique already uses a partial bitstream in order to only configure those parts of the full FPGA design, which have a high priority to be up and running quickly.

Obviously, in order to get a small configuration time, it is key to have a small timing-critical design. Therefore, this initial system should only contain the hardware components which are absolutely necessary to run an embedded processor subsystem (i.e., meet the start-up deadline).

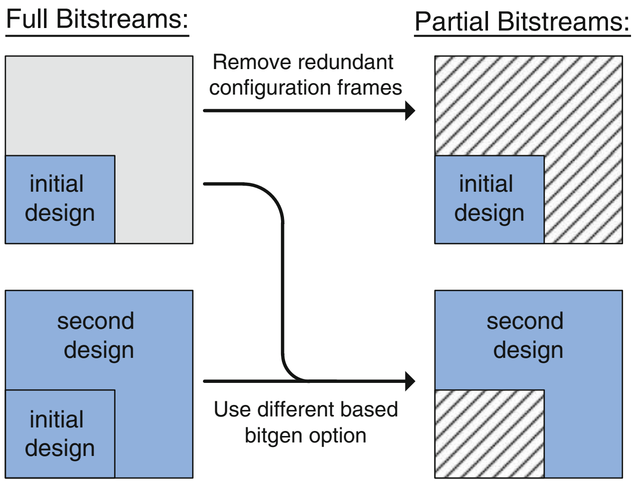


**Fig. 4** Basic approach to create the partial bitstreams for fast start-up

In order to implement the two-step configuration of this fast FPGA configuration technique the following steps have to be done:

- Partition the complete FPGA design into a timing-critical part and a non-timing-critical part.
- Create a special partial bitstream for the initial timing-critical configuration.
- Create a partial bitstream for dynamic partial reconfiguration.

How to partition a full design into timing-critical and non-timing-critical components depends on the specific application. While for most components it is obvious if you need them in your initial design or not, Sect. 3.2 will analyze different memory architecture options in order to help getting the best option for the fast configuration technique.

Since there is no support for Spartan-6 by any of the available partial reconfiguration tool flows, but the Spartan family is the preferred FPGA family from Xilinx for embedded systems due to the low costs and low energy consumption, the creation processes of the partial bitstream for the timing-critical partition as well as for the non-timing-critical partitions afford nonstandard procedures. The basic concept of the flow to create such bitstreams for Spartan 6 can be seen in Fig. 4. In order to get a partial initial bitfile which is holding the initial design configuration, we create a full bitstream of the initial design first. This full bitstream is edited on a binary level to remove the configuration data which is not required and to get the partial initial bitstream.

In order to create the partial bitstream for the dynamic partial reconfiguration of the second design, it is possible to use the BitGen option "-r" for difference-based partial reconfiguration which is still available for Spartan-6. Applied on a full design, using the full bitstream of the initial design as reference, this option produces a partial bitstream containing only the configuration data of the second design.

### 3.1.1 Generation of the Initial Partial Bitstream

As mentioned before, in order to get the initial partial bitstream, all redundant configuration data of a full bitstream has to be removed. This affords a deep knowledge of the configuration memory structure and the bitstream composition. The following low-level information about bitstream composition and configuration procedure is based on configuration user guides like [18] or [15].

The configuration of a Xilinx FPGA is organized in several configuration rows, each consisting of multiple columns of resource elements like, for example, the configuration logic blocks (CLBs). Such a configuration column can be broken down into several configuration frames which are the smallest addressable segments of the configuration memory space, and therefore an operation always affects a whole frame. A configuration frame can be thought of as a one-bit-wide column which spans a whole configuration column. Thus one frame holds only little configuration data of one specific resource element, but therefore it holds this information for all the resources in the corresponding configuration column.

In order to reduce the configuration bitstream size, the compress option of the Xilinx BitGen tool can be used. This option avoids writing similar frames multiple times into the FPGA. Instead, it writes this frame one time into the FDRI, and afterwards the combination of updating the frame address register (FAR) with one of the corresponding addresses for the frame and triggering a multiple frame write (MFW) follows. A MFW is a special configuration command which uses the current frame inside the FDRI to configure the configuration memory addressed by the current value of the FAR. After some no-operation commands, the procedure of updating the address and triggering an MFW is repeated until all addresses for the frame are written.

Using the compress option, it is possible to replace multiple similar frames of an ordinary bitstream with such a sequence of MFWs. An ordinary Spartan-6 frame usually contains 65 configuration words; an MFW sequence for one frame is about 4–5 configuration words. The efficiency of the compress option therefore obviously depends on the amount of similar frames in a design. For Xilinx FPGAs, the configuration data for unused resources consists of zeros only, such frames are called zero frames. Thus an FPGA design which only uses a small amount of logic of the FPGA contains a lot of such zero frames, and therefore using compress with such a design will decrease the configuration bitstream size significantly.

However, all the memory addresses, the MFW commands and the no-operation command words are still inside the bitstream. But for zero frames, this is redundant information because after the house cleaning process, all configuration memory should be initialized with zero anyway. While for an ordinary configuration bitstream, it is quiet complex to remove the configuration data of unused resources and add the necessary address updates by hand, this is much easier for a compressed bitstream. This is because the compressed bitstream structure already separates the zero frames by putting them into MFWs. Therefore, the zero frames can be removed easily from the bitstream by removing all MFWs of zero frames. A comparable approach was used in [12] to decrease the amount of nonvolatile memory for an initial configuration bitstream using Virtex 4.

The removal of zero frames of a compressed bitstream results in a valid partial bitstream which can be used for initial configuration. To get a full tool flow we automated these modifications in a small custom software which does not only remove the frames but also recalculates the cyclic redundancy check value of the bitstream.

### 3.1.2 Placement of Timing-Critical Hardware

In order to bring up all timing-critical modules as fast as possible, you have to choose the area in the FPGA for these modules with care. Of course all modules should be placed in one contiguous area as small as possible but still providing all necessary resources. Utilizing resources outside these areas is possible but will add additional frames to the initial bitstream, because even a single net through an unused region prevents several zero frames which could have been removed. Therefore, a region has to be found which holds as much special resources of your timing-critical modules as possible. It often makes sense to place the region near the ICAP primitive.

Furthermore it is useful to align the horizontal boundaries of the area for timing-critical components with the border of configuration frames like in Fig. 5 on the right-hand side. By this you support that frames are either used completely or not at all. This is helping to create as much zero frames as possible.

### 3.1.3 Dynamic Partial Reconfiguration for Spartan-6

While it is possible for Virtex architectures to use a standard partial reconfiguration tool flow in order to create the partial bitstream for the second configuration, Spartan-6 is not supported by Xilinx for partial reconfiguration. Nevertheless, with the right combination of standard implementation techniques and the BitGen option for difference-based partial reconfiguration, it is possible to create partial bitstreams which were successfully used for dynamic partial reconfiguration. As mentioned before and shown in Fig. 4, the difference-based BitGen option can be used to extract the difference of the full design and the initial design. Therefore, the key element of the flow is to create those two designs in a way which ensures the initial design part doesn’t change. This makes sure the partial bitstream for the second configuration only contains information of the second design part.

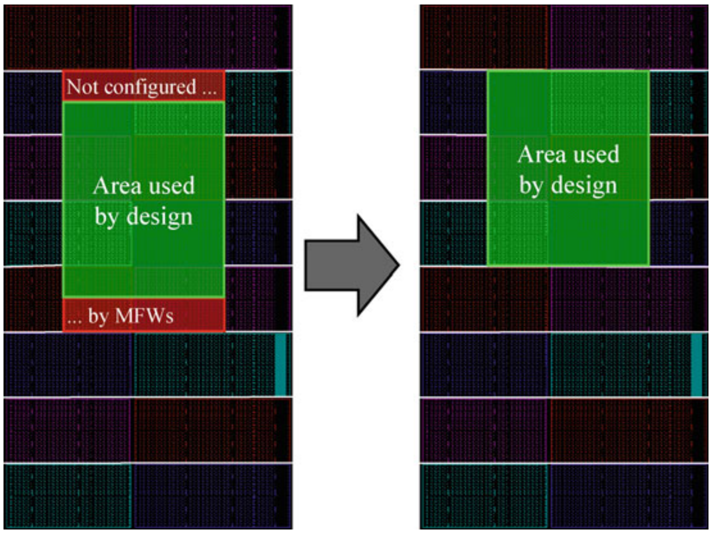


**Fig. 5** Basic approach to create the partial bitstreams for fast start-up

Keeping the initial design part from changing during the two implementations can be achieved by design preservation using partitions [17]. Those partitions create logical boundaries between hierarchical modules and thus make it possible to reuse the implementation information of partitions already implemented in a previous design. To preserve the complete routing of the initial design, all IO buffers which are driven by signals from this design part should be instantiated inside the corresponding hierarchical sub-module.

For nets which leave a logical module of the initial design part in order to build a connection to the second design part, the strategy is to route them through an interface logic which is placed outside of the area of the initial design part but belonging logically to the initial design part module and thus to the preserved partition. This can be used to make sure no frames in the area with the first design part are reconfigured when the dynamic partial reconfiguration adds the second design and the connection to the mentioned interface logic. This logic should also provide an enabled signal which makes it possible to disable the connection. This is used to avoid glitches, resulting from the configuration of the second design, to reach the first design part. In order to avoid the nets from the second partition to get routed through the area of the first design part, the "contained route" constraint should be used for the partition of the second design.

## *3.2 Memory Architectures for Fast Start-Up*

FPGAs do offer design-time and run-time programmability, which we exploit in order to meet the application's start-up requirements. The timing-critical hardware

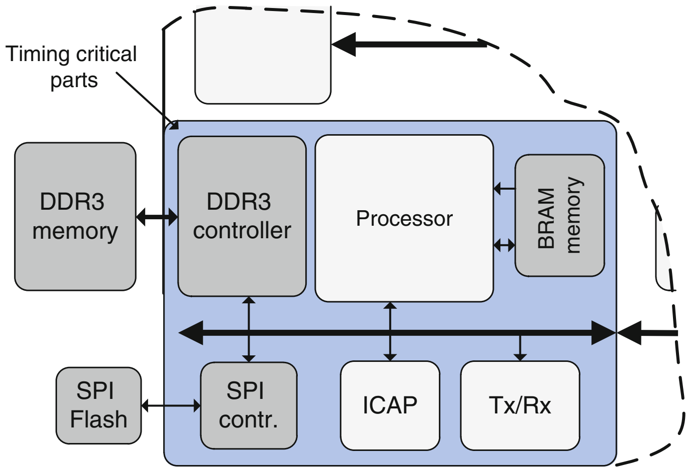


**Fig. 6** Different memory architectures for the timing critical design

components configured at FPGA power-up are application dependent, but in general, we can find the following components: (1) an embedded soft-core microprocessor (e.g., MicroBlaze from Xilinx); (2) a communication interface (e.g., CAN block), to interface with the rest of the system; (3) a volatile memory hierarchy, used for software execution; (4) a block to interface to non-volatile external storage (e.g., SPI flash, which stores FPGA configuration data for non-timing critical blocks); and (5) an internal configuration access port (ICAP), used for FPGA configuration of the non-timing-critical components.

The flexibility of FPGA's allows us to customize these timing-critical components in many different ways. For example, we can configure the embedded microprocessor to include only the required features (e.g., size of cache memories). However, we exploit FPGA's flexibility to implement different memory architectures, which have a *significant* impact on the embedded system start-up time. Different memory architectures differ in used FPGA resources as well as in performance.

One memory architecture is based *only* on on-chip FPGA resources. For current Xilinx FPGAs, those resources (i.e., Block RAM) are able to store up to 36 Kbits each. The memory subsystem is directly connected to the embedded processor, providing an optimal performance. A block diagram of this memory architecture is shown in Fig. 6. However, the amount of memory that can be implemented using this approach is limited. Not only the limited amount of BRAMs per FPGA is critical but also the requirement that those BRAMs have to be located inside the area of the timing-critical hardware components. Accessing more of those resources than available in this area would increase the area and thus increase the bitstream size (i.e., FPGA configuration time). Therefore, this architecture is only viable whenever the footprint of the software is very small.

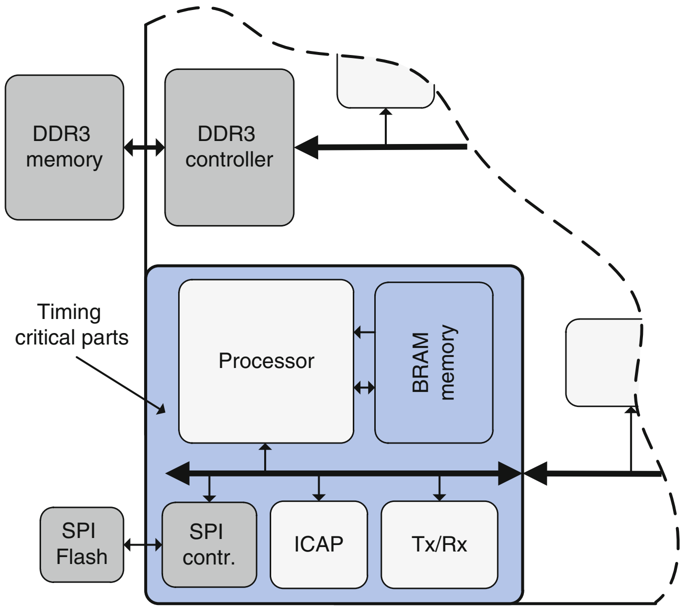


**Fig. 7** Different memory architectures for the timing critical design

The second memory architecture integrates an external memory controller in the timing-critical design (e.g., DDR3 SDRAM memory controller), which removes almost any limitations on software footprint. Figure 7 shows this architecture. However, implementing such a memory controller adds significant amounts of FPGA resources to the timing-critical system, even when using the dedicated Spartan-6 hard memory controller. Hence, when compared to the previous memory architecture, the timing-critical FPGA configuration time will increase for this option.

Before the software can be executed by the processor, it has to be moved from the slow external non-volatile memory to the main memory of the processor subsystem. This is achieved differently depending on the used memory hierarchy. For the memory architecture based on on-chip memory (i.e., BRAM) only, this task is done very efficiently by the configuration logic of the FPGA. The software is included in the initial FPGA configuration, as part of the FPGA bitstream, to initialize the BRAM resources with the correct software. In a traditional full configuration technique this does not increase the configuration time since all the configuration memory of the FPGA is written anyway. However, our *fast FPGA configuration* technique avoids writing BRAM resources which are not used by the timing-critical blocks. Therefore, the more software is included in the timing-critical bitstream, the bigger the footprint will get which of course increases the FPGA start-up time.

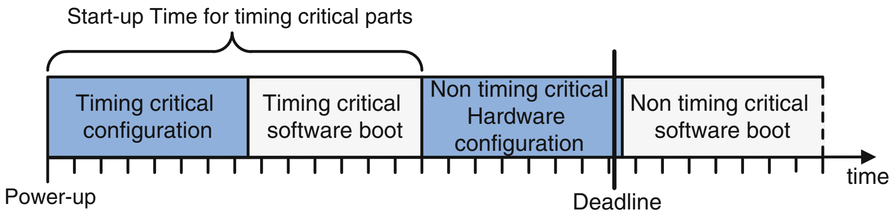

Start-up Time for timing critical parts
Timing critical configuration
Timing critical software boot
Non timing critical Hardware configuration
Non timing critical software boot
Power-up
Deadline
time

**Fig. 8** With the optimized start-up process, timing-critical tasks can meet tight deadlines

In the case of using external memory, moving the data from the flash memory into the DDR3 SDRAM memory requires a boot loader. This very small piece of code usually runs out of BRAM memory. Therefore, even when using external memory, a small amount of internal BRAM memory is always required. The major disadvantage of this architecture is the inefficiency of the software-based boot loader compared to a hardware-based solution (e.g., use FPGA configuration logic to initialize BRAM contents). Using a dedicated DMA controller to move the software executable from flash to the external memory is not the most efficient option, since it is a FPGA resource-intensive component, and it therefore would increase the initial (i.e., timing-critical) FPGA configuration time.

## 3.3 *Software Architecture and Start-Up*

The *fast FPGA configuration* technique comes along with some constraints on the software architecture and boot process. For example, the software architecture has to (1) consider the FPGA configuration of the non-timing-critical components and only when that second FPGA configuration has finished (2) start the software tasks associated to these non-timing-critical hardware components (i.e., it will not work to start a software task for a hardware component that still has not been configured on the FPGA).

To address these two issues, our approach is based on dividing the software in two different parts (i.e., *two-stage software start-up process*): (1) timing-critical software, which has to meet the start-up deadline and (2) non-timing-critical software, which is started after the second FPGA configuration has finished. At the moment it is the job of the system architect to decide which parts of the software belong to the timing-critical part and which will go into the non-timing-critical part. We plan to automate this decision in successive work. Figure 8 shows the concept of a fully optimized start-up process.

This two-stage software start-up process can also be exploited by the boot loader when copying the executable from SPI flash to external DDR SDRAM memory. That is, there is no benefit in moving the *complete* software to external memory at

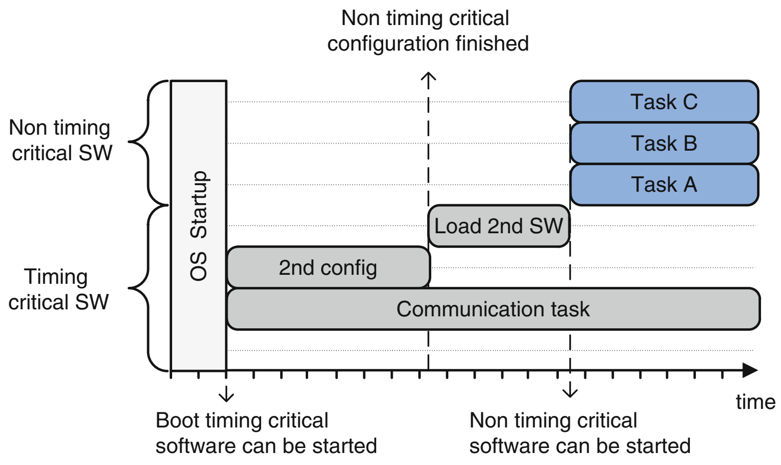

Non timing critical configuration finished
Non timing critical SW
Timing critical SW
OS Startup
Task C
Task B
Task A
Load 2nd SW
2nd config
Communication task
time
Boot timing critical software can be started
Non timing critical software can be started

**Fig. 9** Software architecture and boot sequence

power-up, since there will be sections of code that the processor will not execute at start-up (i.e., the non-timing-critical tasks). Only the timing-critical tasks are moved initially by the boot loader, hence reducing the start-up time.

Figure 9 shows a more detailed view of the software architecture and start-up sequence, after the timing-critical FPGA configuration has finished.

This two-stage software start-up concept becomes optimal when the timing-critical software tasks are executed from on-chip memory, and the non-timing-critical tasks are executed from external memory. This solution has two key benefits: (1) the resource-expensive memory controller is not present in the initial FPGA configuration (i.e., reduced FPGA configuration time) and (2) avoid a software-based boot loader process to copy data from SPI flash to external memory for the timing-critical tasks (i.e., this is hardware based since it is carried out during the FPGA configuration).

This optimized start-up process is shown in Fig. 10, where we can observe three key steps: (1) during the initial configuration phase, the FPGA is configured with the timing-critical hardware components (no external memory controller), and timing-critical software tasks execute out of on-chip memory; (2) while the time-critical application is running, one task is to get a second FPGA configuration out of the external flash memory in order to configure the non-timing-critical hardware components; and (3) the non-timing-critical software tasks are copied from external flash memory to external memory, where they are executed from. All three components, the timing-critical configuration bitstream, the non-timing-critical configuration bitstream, as well as the non-timing-critical software, can be stored in one low-cost SPI flash memory.

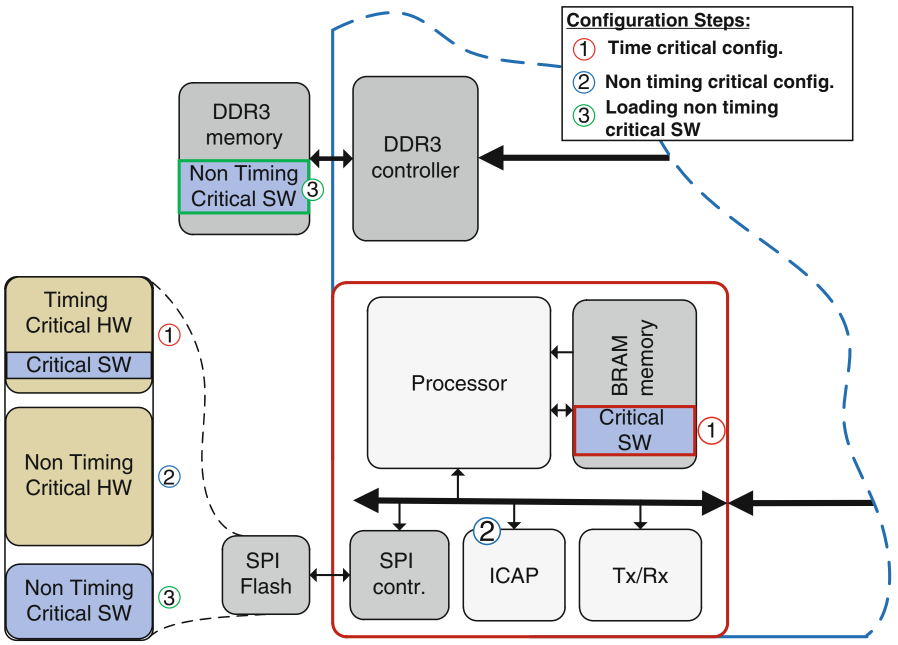


**Fig. 10** Start-up process with memory controller in the non-timing-critical hardware design

### *3.4 Implementing Two-Stage Software Boot*

This section explains how to implement the two-stage software start-up process using a small-footprint operating system (i.e., microkernel), which is the one traditionally used in our target embedded applications. Given the hard timing constraints we are considering, in the order of milliseconds, these real-time operating systems are tightly integrated with the application tasks, creating a single *monolithic* software executable.

In order to implement the two-stage software start-up technique, the single software binary has to be split in two software segments (i.e., timing-critical and non-timing-critical), which have to be located at different memory addresses of the processor's memory map. This is achieved in two main steps, as shown in Fig. 11:

- First, by separating the *sections* of each software segment in the linker script. This is applicable to almost every section type, like *.text*, *.rodata* and *.data*. However, other sections should not be separated, like small sections (e.g., *.sdata*, *.sbss*) or the *.bss* section, since the CRT (C run time) assumes that the *.bss* section resides in a contiguous memory block. Therefore, those sections are not separated but kept in the memory address range of the timing-critical software (i.e., on-chip memory).

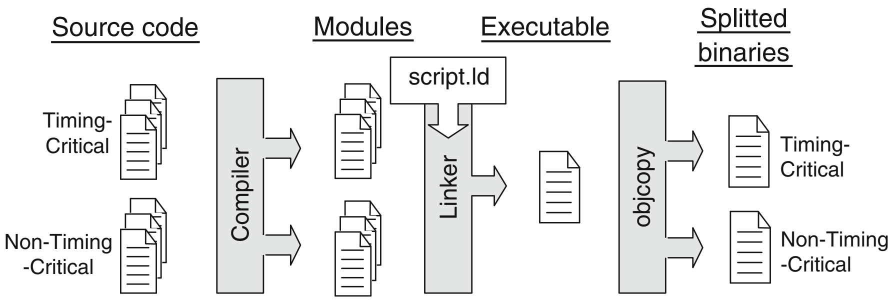


**Fig. 11** Flow for the two-stage software boot

- Second, by using the *objcopy* tool from the GNU binary utilities, which is used to copy specific sections of an executable file to a new file. This tool creates two independent binaries, from a single executable, that can be loaded independently at run time.

## 4 Experiments and Results

### *4.1 Case Study: Automotive ECU Start-Up*

To analyze the different concepts and architectures proposed in this chapter, we used as case study the implementation of an automotive electronic control unit (ECU) on a Xilinx Spartan-6 FPGA. The CAN bus is traditionally used in automotive embedded applications, and it has a start-up requirement of 100 ms (i.e., the ECU should reply to CAN messages from power-up in 100 ms). If a CAN node is not able to boot within 100 ms, it might miss critical messages which cannot be tolerated for automotive systems.

The timing-critical processor subsystem includes a CAN controller for communication purposes, a SPI controller to access the configuration bitstreams and the software binaries, an ICAP controller to have access to the configuration memory of the FPGA and some main memory. The non-timing-critical hardware components include an Ethernet MAC block, a UART, and a timer. The design was implemented in two different ways, depending on the location of the DDR3 memory controller (i.e., timing-critical vs non timing-critical). The first design included a DDR3 memory controller in the timing-critical design, but therefore only 8 KB of memory was implemented using BRAMs. This implementation will be referred to as *DDR3 design*. In a second implementation the on-chip memory implemented using BRAMs was increased to 32 KB, but the DDR3 memory controller was moved to the non-timing-critical hardware design. This one will be referred to as *BRAM design*. A block diagram for these two implementations is shown in Fig. 12.

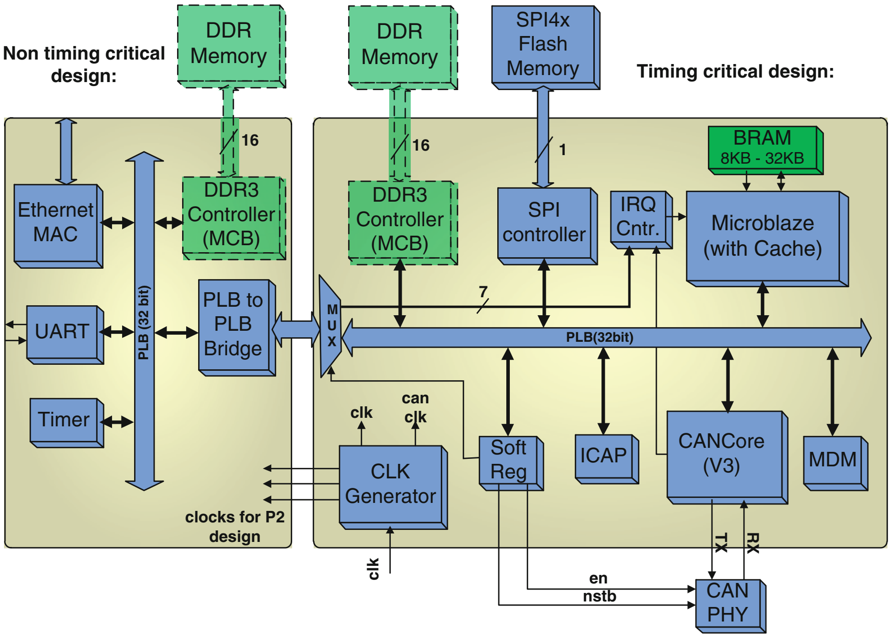


**Fig. 12** Overview of the design. The DDR3 controller is located either in the timing-critical or non-timing-critical design

Figure 13 shows the FPGA Editor view of the DDR3 design. The picture on the left-hand side shows the timing-critical design only. The area location for the timing-critical design was chosen on the left side of the device because we used the hardened memory controller of the left side. Since a lot of nets route to this primitive this was the best location for the DDR3 design. Please note the rectangular configuration frame aligned shape of the major area as well as nets leaving the area to access clock resources, IOs or the ICAP primitive (right bottom of the device). On the right-hand side of Fig. 13 you can see the full DDR3 design.

The BRAM design is illustrated in Fig. 14. We moved the area for the timing-critical part of the FPGA design to the right bottom. There, we had enough BRAMs for the initial memory as well as the ICAP primitive. In this design the DDR3 controller was implemented in the second part (red nets) of the design. This is why the noncritical (red) part of the BRAM design looks bigger than the noncritical (red) part of the DDR3 design.

The software for the designs was built on the embedded operating system OSEK, which is a simple real-time operating system commonly used in automotive ECU's. RTA OSEK was ported to run on Xilinx MicroBlaze embedded processor. Different software tasks to initialize the hardware, to handle CAN communications, to perform the configuration of the non-timing-critical design, to communicate over the UART and to implement several networking protocols were implemented.

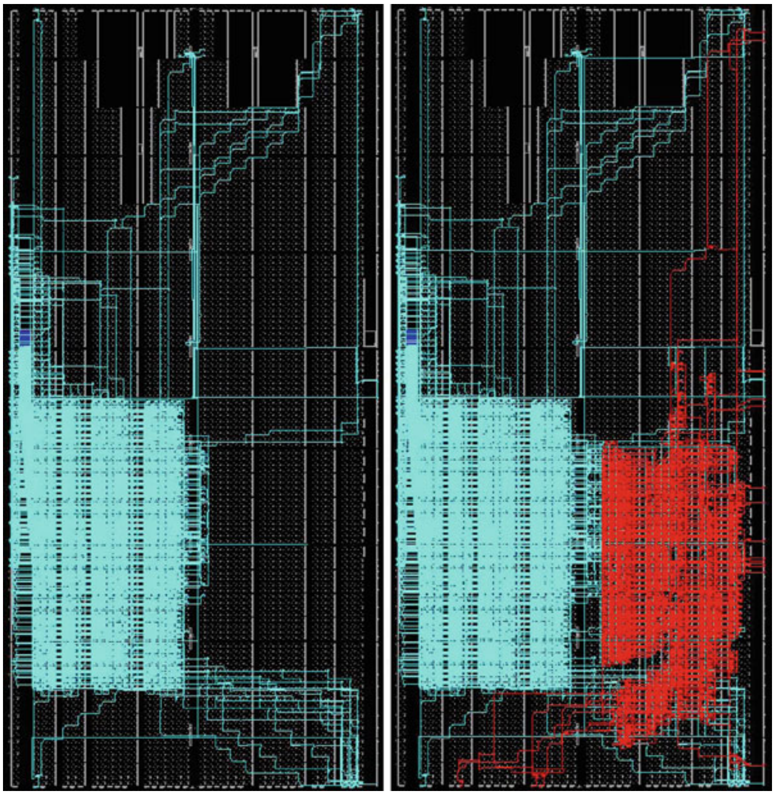


**Fig. 13** FPGA editor view of the timing-critical only (*left side*) and the full (*right side*) FPGA design

If required, there was also a task implementing the boot loader for the non-timing-critical part of the software. Software executables were built using the methodology and tools described in Sect. 3.4.

### *4.2 Experimental Setup*

The start-up time for this case study was measured using a demonstrator we set up in our labs. This demonstrator integrates a PC, a CAN traffic generator, a CAN PHY board, and a SP605 Spartan-6 development board with a XC6S45LXT device; see Fig. 15. The CAN PHY board is able to power-up the SP605 whenever it detects any traffic on the CAN bus. The CAN traffic generator is able to send and receive CAN messages, as well as measure the time between a message was sent and the message getting acknowledged. This is the time the FPGA needs to configure and start the

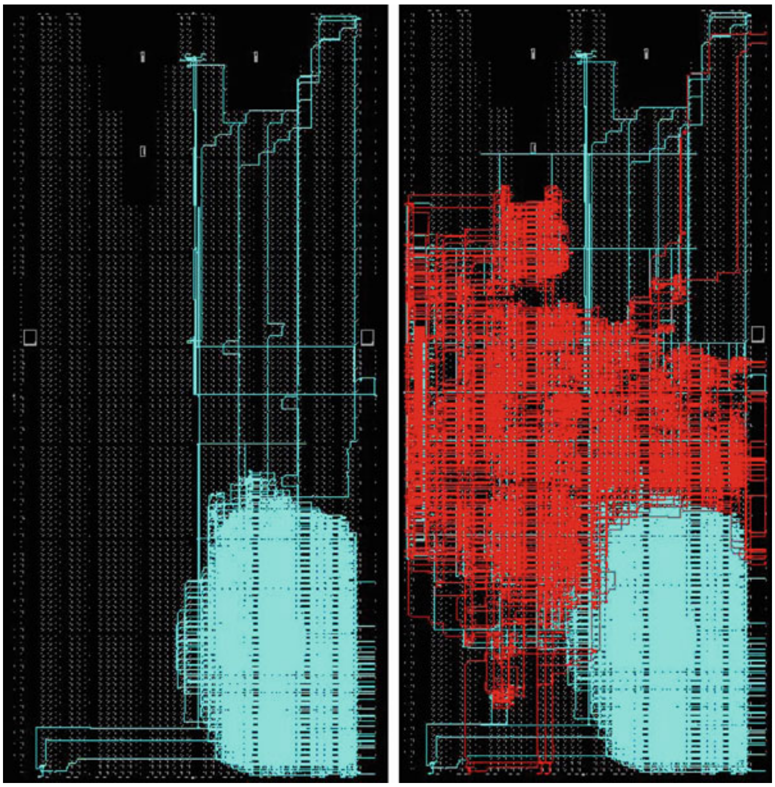


**Fig. 14** FPGA editor view of the BRAM design. Timing-critical components only (*left side*) and the full (*right side*) FPGA design

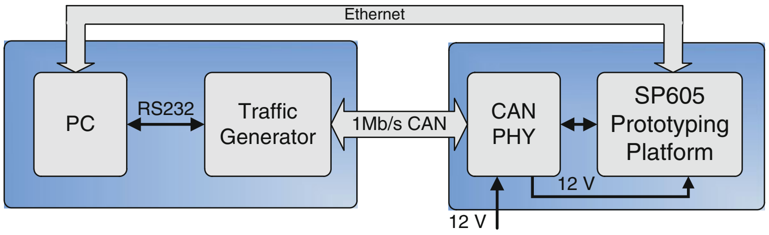


**Fig. 15** Overview of the measurement setup

timing-critical software. The SP605 was used to implement the designs introduced in the previous sections and is connected to the PC using an Ethernet connection. All designs used the Quad-SPI FPGA configuration interface with a configuration rate of 26 to carry out the timing-critical FPGA configuration.

We now briefly explain the methodology used to measure the start-up time on our experimental setup. The SP605 is powered off; the traffic generator sends a

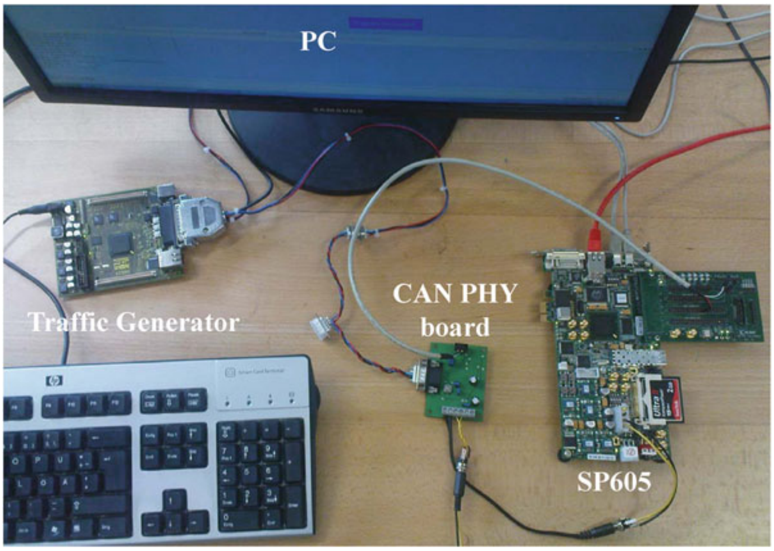


**Fig. 16** Picture of the experimental setup

CAN message and triggers a *hardware timer* used to measure the SP605 start-up time. Whenever the CAN PHY board detects a message on the CAN bus, it powers up the SP605, and the Spartan-6 starts its configuration using the timing-critical bitstream stored on the SPI flash. When this is finished, the timing-critical software CAN task starts to run on the embedded processor and acknowledges the message of the traffic generator. When the traffic generator detects this acknowledgment it stops the hardware timer (i.e., measured time includes FPGA configuration time and software start-up time). Figure 16 shows a photograph of the experimental setup. Please note that the following subsections report measured start-up time of fully working applications on the prototyping platform.

## *4.3 Memory Hierarchy Analysis*

This section summarizes the impact on start-up time of the two memory architectures introduced in Sect. 3.2 (i.e., timing-critical hardware *with/without* external memory controller). We measured the start-up times for both designs using three different concepts: (1) with a traditional full FPGA configuration technique, (2) a compressed version of the timing-critical components only as in [16], and (3) the proposed *fast FPGA configuration* technique. In this example, one of the non-timing-critical software tasks was to simply answer *ping* commands over Ethernet (i.e., small software footprint that fits in the 32KB on-chip memory). Tables 2 and 3 show the results.

**Table 2** DDR3 design memory architecture

| Boot time | Traditional | Compressed | Fast start-up |
|---|---|---|---|
| Critical HW | 1,450 KB | 715 KB | 323 KB |
| Full SW | 23 KB | 23 KB | 23 KB |
| Start-up time | 129 ms | 85 ms | 66 ms |

**Table 3** BRAM design memory architecture

| Boot time | Traditional | Compressed | Fast start-up |
|---|---|---|---|
| Critical HW | 1,450 KB | 637 KB | 263 KB |
| Full SW | 23 KB | 23 KB | 23 KB |
| Start-up time | 118 ms | 69 ms | 46 ms |

**Table 4** Resources for timing-critical designs

| Memory | Resource type | | | |
|---|---|---|---|---|
| architecture | Flip flops | LUTs | Slices | BRAMs |
| BRAM | 2,573 | 3,249 | 1,051 | 24 |
| DDR3 | 3,509 | 4,198 | 1,453 | 16 |

The results demonstrate the significant benefit of the *fast FPGA configuration* technique when compared with a traditional *full* FPGA configuration. Please note that with larger FPGAs, this speed-up would be even higher since the timing-critical design configuration is constant. Additionally, we can observe the start-up time reduction when using on-chip memory only (i.e., BRAMs) in the timing-critical design and moving the resource-expensive external memory controller to the non-timing-critical design (compare Tables 2–4). Although there is additional configuration data needed to initialize the BRAM contents, not including the DDR3 memory controller in the timing-critical hardware components significantly reduces the initial FPGA configuration.

## 4.4 Software Scalability Results

The main goal of this section is to demonstrate the scalability, in terms of software size/footprint, of the optimized start-up approach, where the timing-critical software tasks are executed from on-chip BRAM memory, and the non-timing-critical tasks are executed from external memory.

The results presented in Table 5 show the start-up time when running a non-timing-critical software task able to process UDP packets. This increased the software size to 48 KB, which did not fit into the 32 KB on-chip BRAM memory. On the other hand, the results shown in Table 6 were obtained when running a *webserver* with a software size of 284 KB.

In both examples, using traditional start-up techniques does not achieve the hard start-up deadline (i.e., 100 ms) when using DDR memory controller in the initial configuration; or it is not possible to implement both applications because there is not enough on-chip memory. However, when using the proposed two-stage start-up

**Table 5** Measured start-up time; UDP task

| Boot | Traditional start-up | | Optimized start-up | |
|---|---|---|---|---|
| time | DDR3 | BRAM | DDR3 | Mixed |
| Critical HW | 1,450 KB | – | 323 KB | 263 KB |
| Critical SW | 48 KB | – | 16 KB | 16 KB |
| Non critical SW | – | – | 32 KB | 32 KB |
| Start-up time | 155 ms | – | 60 ms | 46 ms |

**Table 6** Measured start-up time; webserver task

| Boot | Traditional startup | | Optimized startup | |
|---|---|---|---|---|
| times | DDR3 | BRAM | DDR3 | Mixed |
| Critical HW | 1,450 KB | – | 323 KB | 263 KB |
| Critical SW | 284 KB | – | 16 KB | 16 KB |
| Non critical SW | – | – | 269 KB | 269 KB |
| Start-up time | 334 ms | – | 60 ms | 46 ms |

technique, we always meet the required hard deadline with the *mixed BRAM+DDR* implementation providing the minimum start-up time. Please note that the start-up time is now *constant* and *independent* of the complete application software size.

## 5 Conclusion

The presented techniques, methods, and tool flows enable the implementation of embedded systems achieving tight start-up timing constraints using modern FPGAs. FPGAs run-time flexibility (i.e., programmability) is used as the basis for a *two-stage* embedded system start-up, where both the FPGA configuration and embedded software start-up times are reduced.

With the enormous growth of the FPGA size the need for fast-boot techniques grows similarly in order to be able to compete against non-reconfigurable systems. The fast FPGA configuration technique does not only improve the configuration time for timing-critical components of an embedded system; it makes the configuration time for those parts independent from the device size. If a design is able to meet, timing constraints is no longer dependent from the FPGA but from the amount of resources which are timing-critical. This enables to move your design to bigger or smaller FPGAs with almost no influence on the start-up time.

The same is valid for the software part of the start-up time. With the separation into partitions with different priority for start-up, it becomes possible to avoid time-expensive transfers of software between memory hierarchies during start-up. Furthermore this technique enables the start-up from BRAM primitives as main memory for the critical software, even if the overall system software exceeds BRAM limitations.

The proposed techniques for FPGA-based system start-up consider and optimize hardware as well as software start-up and therefore cover and optimize for the first time the complete boot time of such a system. The memory hierarchy analyses shows that the right choice on how to implement the main memory of the embedded system has a significant influence on the start-up time. Due to the constantly growing amount of modern FPGA resources, the fast start-up technique is even more valuable for next-generation Xilinx FPGA devices, like 7 series [19] and Zynq [21], which do not provide a hardware-based solution to meet the PCI Express start-up requirement.

All techniques were used to implement an automotive embedded system on a Spartan-6 FPGA in order to show the feasibility and quantify the benefits of the proposed approach. The results demonstrate that the start-up time for the timing-critical components of the design are nearly independent from the complexity and thus from the start-up time of the complete system.

## References

1. Benavides T, Treon J, Hulbert J, Chang W (2007) The implementation of a hybrid-execute-in-place architecture to reduce the embedded system memory footprint and minimize boot time. In: Information reuse and integration, 2007. IRI 2007. IEEE international conference on, pp 473–479, DOI 10.1109/IRI.2007.4296665
2. Chung KH, Choi MS, Ahn KS (2007) A study on the packaging for fast boot-up time in the embedded linux. In: Embedded and real-time computing systems and applications, 2007. RTCSA 2007. 13th IEEE international conference on, pp 89–94, DOI 10.1109/RTCSA.2007.13
3. Dandalis A, Prasanna V (2005) Configuration compression for fpga-based embedded systems. IEEE Transactions on very large scale integration (VLSI) Systems 13(12):1394–1398. DOI 10.1109/TVLSI.2005.862721
4. Deshpande D, Somani AK, Tyagi A (1999) Configuration caching vs data caching for striped fpgas. In: Proceedings of the 1999 ACM/SIGDA seventh international symposium on field programmable gate arrays, ACM, New York, NY, USA, FPGA '99, pp 206–214, http://doi.acm.org/10.1145/296399.296461, http://doi.acm.org/10.1145/296399.296461
5. Hauck S (1998) Configuration prefetch for single context reconfigurable coprocessors. In: Proceedings of the 1998 ACM/SIGDA sixth international symposium on field programmable gate arrays, ACM, New York, NY, USA, FPGA '98, pp 65–74, http://doi.acm.org/10.1145/275107.275121, http://doi.acm.org/10.1145/275107.275121
6. Huebner M, Meyer J, Sander O, Braun L, Becker J, Noguera J, Stewart R (2010) Fast sequential fpga startup based on partial and dynamic reconfiguration. In: VLSI (ISVLSI), 2010. IEEE computer society annual symposium on, pp 190–194, DOI 10.1109/ISVLSI.2010.19
7. Li Z, Hauck S (2001) Configuration compression for virtex fpgas. In: Field-programmable custom computing machines, 2001. FCCM '01. The 9th annual IEEE symposium on, pp 147–159, DOI 10.1109/FPGM.2001.184258
8. Li Z, Compton K, Hauck S (2000) Configuration caching management techniques for reconfigurable computing. In: Proceedings of the 2000 IEEE symposium on field-programmable custom computing machines, IEEE computer society, Washington, DC, USA, FCCM '00, pp 22 http://portal.acm.org/citation.cfm?id=795659.795918
9. Patel P (2006) Embedded systems design using fpga. In: VLSI design, 2006. Held jointly with 5th international conference on embedded systems and design, 19th international conference on, p 1 DOI 10.1109/VLSID.2006.83
10. PCI-SIG (2005) PCI Express base specification, REV. 1.1. PCI-SIG
11. Schiefer A, Kebschull U (2005) Optimization of start-up time and quiescent power consumption of fpgas. In: Field programmable logic and applications, 2005. International conference on, pp 551–554, DOI 10.1109/FPL.2005.1515783

12. Sellers B, Heiner J, Wirthlin M, Kalb J (2009) Bitstream compression through frame removal and partial reconfiguration. In: Field programmable logic and applications, 2009. FPL 2009. International conference on, pp 476–480, DOI 10.1109/FPL.2009.5272502
13. Stefan R, Cotofana S (2008) Bitstream compression techniques for virtex 4 fpgas. In: Field programmable logic and applications, 2008. FPL 2008. International conference on, pp 323–328, DOI 10.1109/FPL.2008.4629952
14. Wu CH (2008) A time-predictable system initialization design for huge-capacity flash-memory storage systems. In: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, ACM, New York, NY, USA, CODES+ISSS '08, pp 13–18, http://doi.acm.org/10.1145/1450135.1450140, http://doi.acm.org/10.1145/1450135.1450140
15. Xilinx (2009) Virtex-5 FPGA configuration user guide, UG191, v3.8. Available at http://www.xilinx.com
16. Xilinx (2010a) Fast configuration of PCI express technology through partial reconfiguration, XAPP883, v1.0. Available at http://www.xilinx.com
17. Xilinx (2010b) Hierarchical design methodology guide, UG748, v12.1
18. Xilinx (2010c) Spartan-6 FPGA configuration user guide, UG380, v2.1. Available at http://www.xilinx.com
19. Xilinx (2011a) 7 Series FPGAs overview, DS180, v1.5. Available at http://www.xilinx.com
20. Xilinx (2011b) MicroBlaze processor reference guide, UG081, v13.3. Available at http://www.xilinx.com
21. Xilinx (2011c) Zynq-7000 Extensible processing platform product brief. Available at http://www.xilinx.com
22. Yim KS, Kim J, Koh K (2005) A fast start-up technique for flash memory based computing systems. In: Proceedings of the 2005 ACM symposium on applied computing, ACM, New York, NY, USA, SAC '05, pp 843–849, http://doi.acm.org/10.1145/1066677.1066871, http://doi.acm.org/10.1145/1066677.1066871

# Run-Time Scalable Architecture for Deblocking Filtering in H.264/AVC and SVC Video Codecs

**Andrés Otero, Teresa Cervero, Eduardo de la Torre, Sebastián López, Gustavo M. Callicó, Teresa Riesgo, and Roberto Sarmiento**

## 1 Introduction

Benefits of flexible applications force developers and also the industry to design new techniques, methodologies, and tools. The target of this effort is to develop more flexible systems capable of adapting their performance dynamically, saving hardware resources and power consumption but maximizing their performance. However, achieving an optimal design remains a challenge since most real task workloads are dependent on run-time system conditions [14] and environmental issues. In this sense, there are two basic aspects that facilitate reaching a comprising solution. The first one goes through improving the level of parallelism of the system. The second issue requires exploiting the dynamic and partial reconfiguration (*DPR*) benefits that SRAM-based FPGAs offer, mostly Xilinx ones [2, 23]. DPR allows designing IP cores with run-time adaptable parallelism and achieving a flexible assignment of resources. This chapter presents a hardware, modular, and scalable deblocking filter (*DF*) architecture that is capable of modifying its number of modules in one or two dimensions, following a highly modular and regular matrix template of functional units, which was previously presented in [17]. Within the set of features that characterize this architecture, stand out the regularity of communication patterns proposed through the array, since they reduce the number of distributed external memory accesses.

A. Otero (✉) • E. de la Torre • T. Riesgo
CEI, Universidad Politécnica de Madrid, E.T.S.I.Industriales. José Guitérrez Abascal 2, 28006 Madrid, Spain
e-mail: joseandres.otero@upm.es; eduardo.delatorre@upm.es; teresa.riesgo@upm.es

T. Cervero • S. López • G.M. Callicó • R. Sarmiento
IUMA, Universidad de Las Palmas de Gran Canaria, Spain
e-mail: tcervero@iuma.ulpgc.es; seblopez@iuma.ulpgc.es; gustavo@iuma.ulpgc.es; roberto@iuma.ulpgc.es

On the other side, the latest video standards, like the scalable video coding (*SVC*) [10, 22], support different levels of scalability and profiles [16, 26], incorporating a higher degree of flexibility. The disadvantage of such flexibility is an increase of the design and implementation costs to deal with the bunch of possibilities offered by these standards. In this context, those costs can be reduced by breaking the video encoder up into flexible hardware modules in which size can be easily changed independently. Moreover, if this modification is carried out at run time, as it is proposed in this chapter, a highly adaptable scenario can be envisaged. It might be composed of different blocks in charge of video decoding tasks, with the capability of dynamically adapting its size, and accordingly, its performance, to the type and levels of scalability selected by the users, or other run-time environmental conditions.

Going further in the profiling of the SVC decoding process, the deblocking filter is not only one of the most computationally intensive tasks of the standard, but it is also highly dependent on the selected video profile [12]. In addition, a parallelization scheme is proposed that performs the scaling process consistent with the data locality restriction imposed by the architecture. Furthermore, the proposed architecture has been designed to be reused and flexible, such that its general framework might be adapted to process different kinds of applications in which there exist certain kinds of data dependencies.

The rest of this chapter is organized as follows. In Sect. 2, a review of the state of the art on parallel and scalable architectures is shown. Then, the role of the deblocking filter within the H.264/AVC and the SVC video standards is described in Sect. 3. In Sect. 4, the possibilities of parallelizing the DF are discussed. Section 5 describes all the modules belonging to the proposed approach, which as a whole will form a DF. Section 6 focuses on different implementation issues. Section 7 shows details about how the proposed scalable DF is integrated as part of an embedded system, while in Sect. 8, dynamic reconfiguration details are presented. In turn, Sect. 9 generalizes the proposed approach to solve different kinds of problems. In Sect. 10 implementation results are presented. Finally, in Sect. 11, the main conclusions achieved in this chapter are highlighted.

## 2 Related Work

There exists a significant research interest in developing applications able to reach a good trade-off between flexibility and real-time performance. In this sense, numerous works have been focused on addressing the challenges introduced by parallel and distributed computing applications. This has led to develop a diverse number of solutions, in terms of tools, methodologies, and architectures. The key target is to reduce the complexity of design and implementation stages of these resource demanding applications but maximizing the final reachable performance. In any of these cases, the solution goes throughout increasing the level of parallelism of the processing cores, independently of whether these cores are software or hardware based.

Some existing approaches explore scheduling and optimal parallelism selection issues rather than considering architectural challenges. An example of this kind of approach is *PARLGRAN* [1], a framework that deals with mapping and scheduling aspects derived from using dynamic parallelism selection. The purpose of this solution is to maximize the performance of an application task chain by selecting an appropriated parallelism granularity for each task. Different levels of granularity are achieved by instantiating several copies of the same task, allowing that all of them work concurrently. As a consequence, the task workload is shared equitably among the instances, which means that the execution time is reduced proportionally to the number of instances. The disadvantage of this solution is that it only considers tasks without dependencies between disjoint data blocks, which restrict its applicability to many data-parallel tasks, such as the DF. Like et al. [14] tackle a similar problem, but in this case, the solution considers dynamically reconfigurable adaptive accelerators. The authors propose balancing area and execution time while unrolling application loops, dependent on the inputs and tasks running concurrently in the reconfigurable area. The more the loop is unrolled, the higher the parallelism.

Since this chapter is focused on the accelerator selection, no architectural novelties on how different accelerators are built are provided. Recently, Salih et al. have proposed in [21] a scalable embedded multiprocessor architecture implemented on an FPGA. Despite that this architecture is implemented on an FPGA in order to reduce the time to market, it does not exploit the flexibility of this hardware device as much as it could. The architecture is based on a core processor, which controls, synchronizes, and manages the whole system and the memory accesses, and several processing modules working in parallel. These modules are implemented as a scalable embedded concurrent computer (*ECC*) architecture, where one or several modules might work simultaneously. Once again, this architecture is conceived using software techniques, and the scalability is referred to parallelize different kinds of independent tasks, with no data dependencies among them. As a consequence, this solution is not well suited to accelerate algorithms like the DF. Following the same tendency, other examples of architectures that provide scalable characteristics are [8, 13]. Both of them have been developed for supporting real-time multimedia applications. The authors in [8] introduce a new framework, *P2G*, formed by a CPU and several GPUs, that is, *execution nodes*. The topology between the CPU and the nodes is flexible, since it can change at run time accordingly to the dynamic addition or removal of nodes. The workload is distributed following two scheduling approaches; one of them distributes data throughout the nodes (task parallelism), while the other is responsible for maximizing the performance locally in each node (data parallelism). This approach presents an interesting concept of dynamic adaptability, though this idea is again based on software concepts. On the other hand, in [13], a parallel processor for real-time image processing is presented. This solution, named *MX-2 Core*, is composed by a small processor and a parallel processing unit (*PPU*). The design of the PPU is based on 2048 4-bit processing elements (PE) combined with SRAM cells. The scalability of this approach is explored into the PPU, since the number of enabled PEs might vary in numbers of 256, in accordance with the task that it has to process. However, the modification into the number of PEs is programmed statically, which means that it is designed depending on the specific task that is going to perform. As an alternative to these scalable software-based approaches, hardware solutions provide higher levels of flexibility and adaptability by exploiting the DPR capabilities of some FPGAs.

Achieving scalability by means of DPR allows reusing free configurable areas for other cores within the reconfigurable logic. However, most existing scalability-related works are oriented to adapt core functionality or operation quality rather than to set area-performance trade-offs. For instance, the scalable FIR filter provided in [5] offers the capability of adapting the number of taps to adjust the filter order, offering a compromise between filtering quality and required resources. Furthermore, a scalable DCT implementation in [11] allows varying the number of DCT coefficients that are calculated by the core in order to adapt the number of coefficients that will be subsequently quantified and therefore adjust the video coding quality.

Regarding the DF implementations, a scalable DF architecture is proposed in [12]. This approach exploits the properties of the DPR. This chapter explores a block-level parallelism implementation, where the variation of the number of processing units working in parallel impacts on the execution time of a single MB, without dealing with MB dependencies. This approach, while still being interesting for flexibility purposes, is limited by the maximum number of $4{\times}4$ blocks that can be processed simultaneously in one MB, with no further scalability levels. In addition, the designed floor planning does not allow for an easy reuse of the area released when shrinking the filter, so real area-performance trade-offs cannot be easily achieved.

## 3 H.264/AVC and SVC Deblocking Filter

The DF algorithm is part of the H.264/AVC and the SVC video standards, and it is responsible for improving the visual perception of a reconstructed image by smoothing the blocking artifacts generated by previous tasks within the encoding and decoding loops. Those artifacts are mainly due to the division of each video frame into disjoint sets of pixels called macroblocks (*MBs*), which are processed independently.

DF is a highly adaptive algorithm, where the filtering operations performed on all the MBs belonging to a decoded image highly vary depending on the image content and the encoding decisions made during the encoding process. An MB is organized into a matrix of $16 \times 16$ pixels of luminance and two matrixes of $8{\times}8$ pixels of chrominance (blue and red), when using the coding format 4:2:0, as it is the case of almost all the consumer video encoders. These pixels are organized into units of $4 \times 4$ pixels named blocks or into 4 lines of pixels (*LOPs*). Every block is enumerated from zero to 23, where 16 belong to the luminance, 4 to the blue chrominance, and the other 4 to the red chrominance.

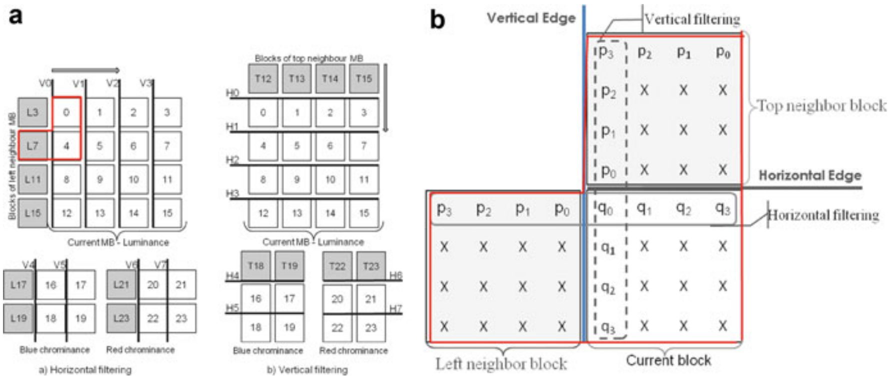


**Fig. 1** DF behavior constraints. (**a**) Data distribution into an MB (**b**) Filtering order

The DF data processing is mainly executed by two blocks: the boundary strength and the filter block. The former one calculates the filtering mode, which is a value between zero and four. The difference among these values, or strengths, is the number of pixels that might be manipulated during the filtering process. The latter one is responsible of filtering data, modifying the current data and its neighbors. List et al. in [15] describe more in detail the operations performed by the DF. In addition, the aforementioned standards constrain the DF behavior, specifying how the data of an MB must be processed depending on the filter strength, as well as the order in which these data have to be filtered, as Fig. 1 represents.

As Fig. 1a shows, an MB is split into vertical and horizontal edges by considering the edges of its blocks. Thus, the *vertical edges* ($V_i$; $0 \leq i \leq 7$) are determined by the left borders of a column of blocks. Similarly, the *horizontal edges* ($H_i$; $0 \leq i \leq 7$) are the top boundary of a row of blocks. As a result, the filtering of a vertical edge between two blocks corresponds to a horizontal filtering, whereas the filtering of a horizontal edge implies a vertical filtering. According to the H.264/AVC and the SVC standards, to consider an MB completely processed, all its blocks have to be filtered (firstly, horizontally (from left to right), and afterwards, vertically (from top to down)). Figure 1b depicts how the pixels of a LOP are processed, starting from the pixel closest to the edge and moving away from it. In this figure, $q_0$, $q_1$, $q_2$, and $q_3$ represent the pixels of the block that is being filtered, while $p_0$, $p_1$, $p_2$, and $p_3$ represent its neighbor pixels.

## 4 Strategies of DF Algorithm Parallelization

The level of parallelism of the DF is directly dependent on the data granularity implemented (LOPs, blocks, or MBs). In this sense, a fine-grain architecture is focused on processing LOPs, whereas a medium-grain one works at block level,

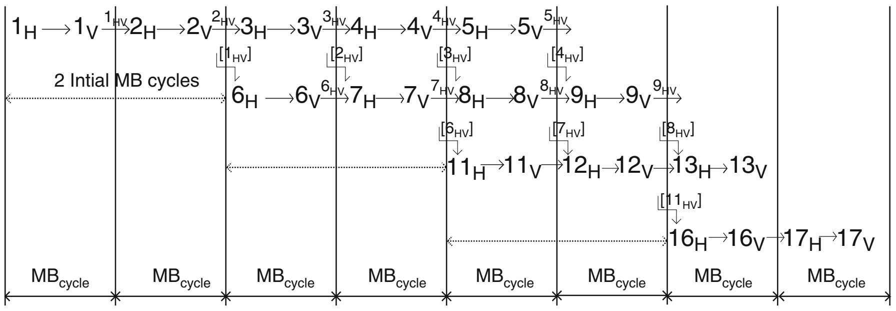


**Fig. 2** Data dependencies between MBs for a 6-MB-width video frame

and a coarse-grain one at MB-level. In the two first granularity levels, the maximum level of parallelism is limited to a full MB, which means that only one MB might be processed concurrently even when it is separated into the smallest units. This fact obligates to process all the MBs sequentially one by one, following a *raster-scan* pattern. This pattern implies a strict order, starting from the top left corner of an image, and moving to the right, repeating the process row by row until image completion. This is the strategy followed by all the fine- and medium-grain approaches, understanding the granularity from a data perspective, independently of the device characteristics. Moreover, static architectures are also constrained to follow this kind of strategy. However, in order to overcome this limitation and to be able to process several MBs in parallel, it is necessary to explore new techniques. It is in this scenario when the *wavefront* strategy comes up. A wavefront pattern is characterized by allowing processing of several MBs simultaneously out of order, but not randomly, fulfilling with the data dependencies. Further details about the general wavefront approach are offered in Sect. 9, while existing dependencies in the case of the DF algorithm are discussed below.

## *4.1 MB-Level DF Parallelization Strategies*

As follows from the DF algorithm description, horizontal filtering must precede vertical filtering, and in both cases, it is necessary information of the data that is being currently filtered and its neighbors. These constraints limit the level of parallelism of the DF.

The specific dependencies between MBs are depicted in Fig. 2. In this figure, $MB_H$ refers to an MB once it has been filtered horizontally, and $MB_{HV}$ represents the MB after it has been filtered both horizontally and vertically. An MB is completely processed when $MB_{HV}$ is filtered again, during the horizontal filtering of its right neighbor (after this process, the MB is represented as $[MB_{HV}]$), and vertically during the filtering of the bottom neighbor.

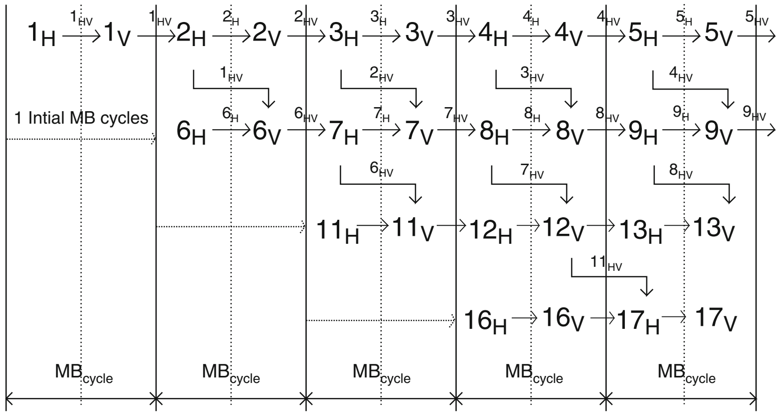


**Fig. 3** Proposed wavefront strategy for a 6-MB-width video frame

Following the example shown in Fig. 2, MB7 needs $MB6_{HV}$ information for being filtered horizontally. Subsequently, it requires both $MB7_H$ and $[MB2_{HV}]$ for being filtered vertically. $[MB2_{HV}]$ is ready once the MB3 has finished its horizontal filtering. Finally, MB7 will be completely processed once MB8 and MB12 have been filtered horizontally and vertically, respectively.

A possible solution to exploit MB-level parallelism might entail using a wavefront order in the same way as the state-of-the-art multiprocessing solutions [24], as can be observed in Fig. 2. Nevertheless, this approach needs to wait twice as many clock cycles for filtering a full MB ($MB_{cycle}$), that is, until $[MB_{HV}]$ is available, before the subsequent core starts processing. To overcome this limitation, an optimized wavefront strategy has been proposed by the authors in [3], in which horizontal and vertical filtering are separated in sequential stages. Thus, the top MB horizontal filtering has already finished when the MB vertical filtering begins, assuming $[MB_{HV}]$ is available. As a result, one $MB_{cycle}$ is saved with respect to previous approaches, as Fig. 3 shows.

Larger architectures that want to process more MBs at a time require its parallelization strategy to be scalable itself in order to obtain consistent results. Actually, an adaptable MB-level parallelism is mandatory to respect the data dependencies between MBs, no matter the size of the architecture in terms of processing units.

Compared with previously reported architectures, in this chapter, we propose a dynamically scalable DF that exploits a coarser granularity than the mentioned state-of-the-art approaches. This architecture works at MB level, and it allows reusing the released area in order to balance area-performance trade-offs. The scalable DF follows a strategy of parallelism fully compatible with the scaling process of the architecture, where the data dependencies between MBs are respected in all cases. The level of scalability varies from just one functional unit (*FU*) up to the maximum number of resources available in the reconfigurable logic area.

## 5 Global Architecture

The core of the proposed architecture is a coarse-grain homogeneous array of processing elements, called functional units (*FU*), as depicted in Fig. 4. Each unit is able to carry out a complete filtering operation on an MB, such that the full array can process in parallel a region of the image. A more detailed description of each FU can be found in [4]. The main strengths of the proposed structure are its inherent parallelism, regular connections, and data processing capabilities. The purpose of the rest of the modules included in the architecture, as described below, is to feed each FU with the required MB as well as to synchronize the array.

A mechanism responsible of the generation of the valid sequence of MB addresses has been included in a hardware module called input controller (IC) in order to respect the processing order defined by the proposed parallelization pattern. This module receives the corresponding sequence of MBs from the external memory and sends them to the array of FUs. In addition, other modules named input memories (IM) have been included at the top of each column of the processing array in order to parallelize data provision to the FUs. Main components of these blocks are FIFO memories that distribute the suitable MBs vertically across each column. With the purpose of capturing the MB associated with the corresponding units, a module called router has been attached to each FU. Once all the FUs capture their corresponding unfiltered MBs, these data blocks are processed in parallel. Once these blocks have been processed, the routers send them back to the output memories (OM). These modules are based on FIFO memories that store the MBs received from the vertical connection and transmit them again in sequential order to the output controller (OC). This OC sends back the processed MBs to the external memory. Data sending, processing, and results transmission stages have been pipelined and overlapped.

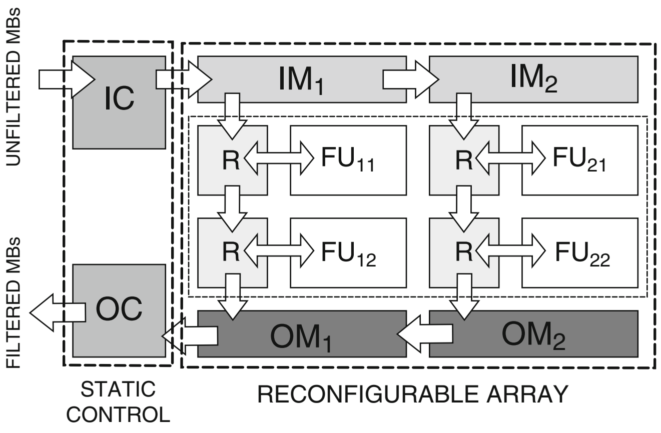


**Fig. 4** Processing array structure

All the mentioned modules of the architecture include distributed control logic in order to manage data transmission, while allowing architecture scalability. Thus, different modules automatically communicate only with their neighbors using shared signals, without having to implement a centralized control, which would reduce scalability due to the fact that the control structure should be designed ad hoc for each possible size of the array.

### *5.1 MB to FU Allocation Policy*

Once the architecture has been described, the policy defining which MB is processed in each FU of the array is discussed, considering the data locality of the algorithm and the regularity of the architecture.

Since horizontal and vertical processes for each MB have to share data corresponding to the full MB, both are carried out in the same FU to minimize communication costs. In addition, according to existing dependencies, semifiltered data ($MB_{HV}$ and $[MB_{HV}]$ according to the nomenclature in Fig. 2) have to be shared between the FU filtering an MB and the FUs in charge of their top and left neighbors. To reduce this overhead, an MB will always be filtered in the same unit as its left neighbor, and its top neighbor will be filtered in the unit below the MB. As explained in the implementation section, specific connections between FUs have been created to allow the exchange of this semifiltered information, without involving routers. The final allocation sequence for filtering all MBs within a frame is dependent on the total number of FUs of the architectural array and also on the number of MBs of the height image frame. Thus, each FU filters all MBs contained in a particular row of the frame while always respecting data dependencies. However, if the number of FUs is smaller than the height of the MBs in a frame, the filtering process is modified. The frame will be processed by stripes with the same height than the number of FUs in the array. This proposal not only reduces the amount of transferred information among FUs, but also, unlike state-of-the-art proposals, only the current MB must be requested from the external memory for each filtering process, as the information related with the neighboring MBs is received during horizontal and vertical filtering stages from other FUs, as explained above. Consequently, data transferred between the external memory and the DF is largely reduced. This allocation strategy is shown in depth in Fig. 9a, included in Sect. 9, where the generalization of the architecture is addressed.

## 6 DF Implementation Details

In this section, details regarding the implementation of the architecture are described, focusing on the issues related with its run-time scalability, including methodological aspects.

## *6.1 Architectural Design for Run-Time Scalability*

One of the main advantages of using this kind of highly modular and parallel architectures, when considering the use of DPR, is the straightforward nature of generating the design partitioning following the modular design flow [28]. Thus, each different module (IM, OM, and FU) is treated as a reconfigurable element on its own. The VHDL description of each module is synthesized, mapped, placed, and routed independently. As a result, three separated partial configuration bitstreams are generated. Following this methodology, since every instantiation of any component in the architecture is equal, a unique version of each one will have to be generated, and afterwards it can be reused in different positions of the array.

The adaptation of the proposed architecture to be dynamically scalable implies the addition of bus macros (*BMs*) [27], as well as the design of an independent floor planning for each module. According to the traditional reconfigurable design flows, the BMs must be placed as part of the interface between the static design and the reconfigurable one. Furthermore, due to the scalability property of this DF architecture, all the reconfigurable modules (IM, FU, and OM) require BMs at their input and output ports. The latest Xilinx dynamic reconfiguration design flow, from the v12 release onward, avoids the use of these fixed macros to guarantee the correctness of communications across modules frontiers [19]. Instead of BMs, elements called partition pins are automatically inserted by the tool in all frontier pins corresponding to all reconfigurable module types to grant communication validity. Even though it reduces DPR overhead, this approach does not allow module relocation in different positions of the FPGA, as this is unsuitable for this kind of scalable architecture because of module replication.

As will be shown, each module has been designed to occupy more area than a static design. This extra area allows routing all the internal signals within the reconfigurable region. Thus, despite the fact that not all the logic resources within the reconfigurable region are occupied, the entire region is necessary to come up with a fully routed design. Values related to the area usage for each module directly impact onto the size of the corresponding configuration bitstream and consequently on the DF reconfiguration time itself. The regularity of this architecture, and throughout the exploitation of the DPR, permits that only eight different configuration bitstreams are necessary for configuring any $m \times n$ size. The scaling of the DF might be done by replicating and relocating the configuration bitstreams of different modules in other positions in the FPGA. As an example, in the case of the addition of an extra row in a DF with two columns, five modules have to be configured (two corresponding to the new FU row, plus the shift of the two OMs, and the OC in their new positions), while the others will remain unchanged.

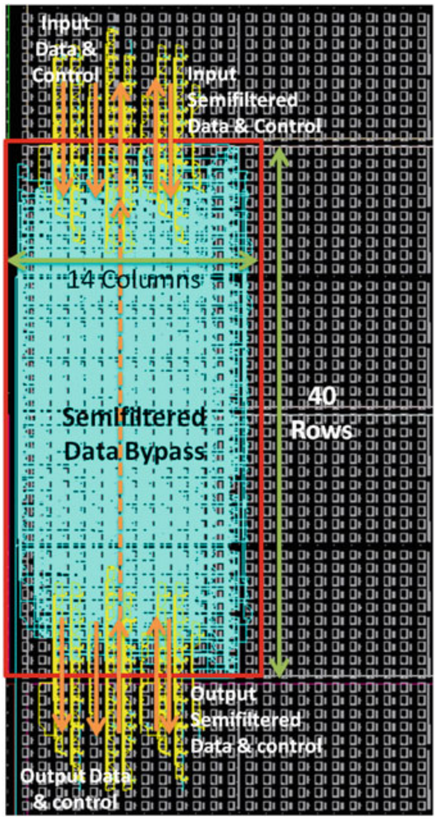


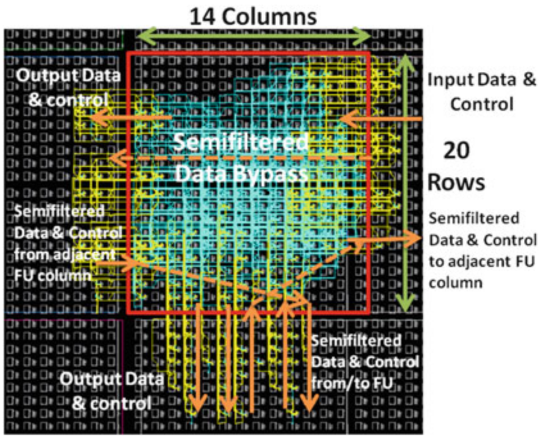
**Fig. 5** Router and FU design

## 6.2 *Implementation of the DF Modules*

In the following subsections, design issues corresponding to each module, as well as main floor planning decisions taken to achieve scalability, will be described.

**Router and Functional Unit (FU)** Since each FU is always attached to its router, both router and FU have been implemented inside a single reconfigurable module. As mentioned before, the role of the router is to capture the first MB received from the IM in each processing stage and then transmit it without changing the subsequent MBs to the FUs below. All these transactions are repeated periodically with each new MB cycle. Both, data and control vertical connections among routers and the IM/OM, have been included in the module border, as shown in Fig. 5. Additionally, specific point-to-point connections with the bottom FU have been created to exchange semifiltered information, as described in the previous section. In the case of the last FU of the column, the next FU is located at the top of the next column. To tackle with this issue, bypass logic has been included both in the OM and the FU blocks to send it upward. In the case of the FU, the bypass connection is highlighted in Fig. 5.

**Fig. 6** IM floor planning design


14 Columns
Output Data & control
Input Data & Control
20 Rows
Semifiltered Data Bypass
Semifiltered Data & Control from adjacent FU column
Semifiltered Data & Control to adjacent FU column
Semifiltered Data & Control from/to FU
Output Data & control

North and south connections of the module are completely symmetric, since both use BMs in the same positions. Consequently, FU and router modules design can be stacked in vertically aligned positions inside the FPGA. It cannot be replicated horizontally because BRAM columns are located in different positions within each right and left side of a region. To overcome this limitation a different module compatible with the FPGA area to each side has been created, including the same local behavior, but with a different floor planning design. Consequently, two different versions of the FU exist. This is shown in Fig. 7, which represents the full architecture.

**Input Memory (IM)** Unfiltered MBs come from the external video frames memory, and they are transmitted across the IM FIFOs. During this process, all the IMs hold the MBs that will be processed by the column of FUs immediately below them. Consequently, the memory size of the IM limits the maximum vertical size of the architecture. In the final implementation, due to the physical restrictions of the FPGA, the architecture was limited to a 2-column $\times$ 3-row size. Once this memory has been filled, the IM distributes the MBs in the vertical direction to the FUs of the same column. Consequently, both horizontal and vertical BMs have been included, as shown in Fig. 6.

Lines to transmit semifiltered MBs have been included for the worst case scenario, as it was explained before. Consequently, the IM receives semifiltered data from the FU bypass output and sends it to the IM on the right side. In addition, this module sends semifiltered data from the left IM to the first unit of the column below. Furthermore, in the case of the last column, a special communication module has been implemented to send this data back to the IC. To make this feasible, a special horizontal semifiltered bypass has been included through the IM.

**Output Memory (OM)** This module includes the logic in charge of receiving the completely filtered MBs from the FU column above, as well as transmitting data to the output controller. It also includes inputs and outputs for transmitting semifiltered

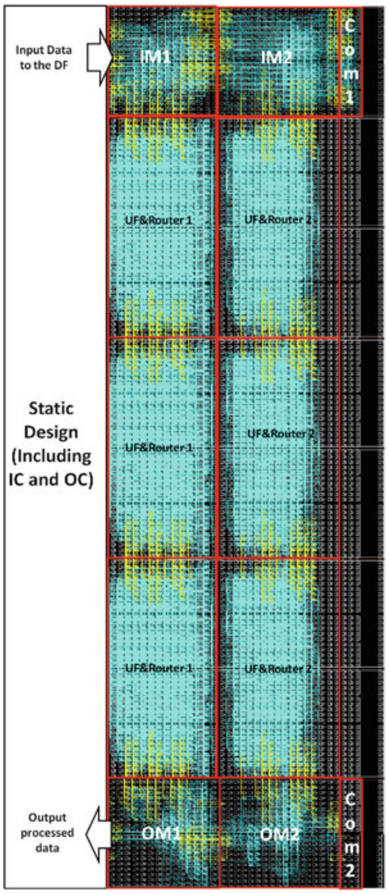


**Fig. 7** Complete architecture design

MBs. Specifically, it bypasses data coming from the semifiltered MB output of the FU immediately above to its semifiltered bypass input, so it can be transmitted to the next column.

**Input Controller (IC)** The IC is the input module of the architecture, and it is the communication point with the static part of the system. This module is not dynamically reconfigured when the DF is scaled. However, certain kinds of its registers are configured from the external embedded processor in order to indicate the current dimensions of the DF and the size of the video frame. With these dimensions, the IC generates the correct MB reading address sequence for any size of the architecture. Bus macros corresponding to the MB data and control signals have been included to communicate with its adjacent IM.

**Output Controller (OC)** The OC is also part of the static design. It receives data from OMs and sends it back to the video memory. Consequently, MBs have to be located across the static-reconfigurable borders to allow for different size architectures. However, future work will be carried out to communicate OC outputs with IMs to have a unique communication module with the static area.

The scalability of the full architecture is shown in Fig. 7. Since the columns are different, eight independent bitstreams have been generated, two per each FU, IM, and OM, as well as two for the communication modules that have been included to close open connections. Both the IC and the OC belong to the static side of the system so that no partial bitstream is generated for those modules.

Thus, the maximum size achieved is 2×3, using the right half of a medium-size Virtex-5 FPGA (xc5vlx110t).

## 7 Embedded System Integration

The purpose of this chapter is to integrate a full H.264/AVC video decoder, together with the DF, as an autonomous embedded system. In this scenario, the DF has been implemented according with the hardware architecture explained along the previous sections, and the rest of the decoder is executed in software by an embedded microprocessor (Microblaze). To guarantee the autonomy of the system, the interface selected to access the configuration memory is the internal configuration access port (*ICAP*). This port is embedded in the very same configurable logic of the FPGA so that the system is able to modify its own configuration. Thus, the final integrated embedded system is composed of a microprocessor, the reconfiguration manager that controls the ICAP, the reconfigurable region (where the DF is implemented), the PLB buses [20] to connect the different blocks among them, as well as two input/output embedded buffers. Therefore, it is necessary to solve the integration issues related to the unfiltered MBs supply from the external memories to the DF core, as well as the transmission of filtered MBs to a specific buffer.

Two isolated buffers have been included into the system to simplify the transmission of MBs between the DF and the rest of the system. One of these dedicated buffers is responsible for storing and loading unfiltered MBs, whereas the other one handles filtered MBs. The former corresponds with the input buffer of the DF, and it is written using a PLB bus interface by other modules of the embedded decoder, and it can be directly read from inside of the DF. The latter buffer is an output buffer that is written by the DF and read by other modules through the PLB interface. The interconnections between the DF and those buffers have been implemented using a direct memory access (DMA) approach. This fact avoids introducing more overheads in the PLB bus, shared with other blocks of the system, something that would reduce the performance of every module, including the DF itself. Furthermore, by incorporating the DMA, any word belonging to an MB might be read in one clock cycle from the memory. Therefore, the DMA is able to supply data fast enough to support the highest data rate demanded by the system, even when the DF core is scaled up to its maximum size. This circumstance could not be guaranteed by using a bus approach, since it would not be able to scale its bandwidth according to the data rate necessary to feed different DF sizes. In addition, the DMA can be directly accessed by the IC of the DF core, without requiring an extra logic. As it was aforementioned, the IC generates the number of required MB, which is equivalent to its address in the input buffer.

In spite of receiving and sending all the data using the DMA interface, the DF has been also connected to the PLB interface in order to receive configuration values and commands. The required values define the dimensions of both the processing array (M×N) and the image (W×H), whereas the configuration commands signal the beginning and the end of each video frame.

## 8 DF Dynamic Reconfiguration Management

In this section, some design aspects related to the DPR controller block, which accelerates the scaling process of the DF at run time, are introduced.

The process of scaling the architecture dynamically is carried out by the customized ICAP controller, or *reconfiguration engine* (RE), proposed by the authors in [18]. Its main task is to control low-level details of the reconfiguration port, including the relocation of configuration bitstreams.

Some features of the reconfiguration engine that are described in this section, together with the regularity and modularity of the DF proposed in this chapter, facilitate a faster scaling process of the DF architecture. On one hand, specific details corresponding to the relocation engine are provided, such as its implementation in hardware for speed, as well as its readback, relocation, and writeback capabilities. Moreover, this RE has been developed to work at 250 MHz. This frequency includes online relocation, beyond the maximum theoretical throughput of the Virtex-5 reconfiguration port, which according to information reported by the manufacturer is up to 100 MHz.

In addition, the configuration bitstreams corresponding to the basic modules of the scalable DF architecture are simplified versions in comparison with the traditional bitstreams generated by vendor tools. In this case, the bitstreams do not include configuration commands. That is, only the body of the logic configuration is stored, while both the header and the tail are obviated. Thus, the final position of any reconfigurable module is not fixed in a determined location in the FPGA. At the end, the complete partial bitstream is composed at run time, as shown in [18], by means of adding the header and the tail information, as well as the specific frame addresses, which are not included either. This strategy provides two advantages:

1. Reducing the bitstream size. Consequently, time required to transfer data from the external memory decreases, and memory storage is minimized.
2. Accelerating and increasing the relocation feasibility.

This relocation technique is much faster than previous proposals in the state of the art. For example, approaches like [6] are based on bitstream parsers, instead of a run-time composition of the bitstream.

Moreover, the reconfiguration engine incorporates a direct and dedicated data link to be communicated with the external DDR2 memory, which acts as a bitstream repository, by using the native port interface (*NPI*). Through this link the reconfiguration port gets new configuration information, fast enough to perform the reconfiguration process, without introducing stall times.

On the other hand, taking advantage of specific features of modular and regular architectures, it is possible to accelerate even more the dynamic reconfiguration. In most cases the scaling process implies the replication of the same element, which means relocating the element in different positions in the architecture. Under these conditions, the possibility of pasting the same reconfigurable module in different positions of the architecture, without having to read its configuration bitstream many times from the external memory, gains in relevance. Moreover, this module might already be configured in other positions of the device. Consequently, also internal memories have been included to allow this readback/reallocation/writeback approach of full modules of the architecture. This approach eliminates the overhead of reading the configuration information from the external memory, sharply reducing the reconfiguration overhead.

RE includes a software application programming interface (*API*) that simplifies the reconfiguration process. Further information about this API can be found in [18].

## 9 Scalable Wavefront Architecture Generalization

The benefits of the architecture proposed in this chapter go further than the hardware implementation of the H.264 DF algorithm. The whole system might be reused to implement several and diverse scalable applications and algorithms following the same parallelization pattern, the wavefront approach. This pattern appears in scientific applications such as [9, 25], as well as in several video processing tasks [24], among others. This section analyzes and brings up general considerations that should be taken into account in order to reuse the proposed architecture and to adapt its modules to cope with other problems.

### *9.1 General Wavefront Pattern and Dependencies Characterization*

The wavefront template is based on the arrangement of data in multidimensional grids so that the elements located in certain positions are fully independent. Hence, these independent data, called data front, can be processed simultaneously. The data dependencies among operations in all the applications determine the data-front shape and, consequently, the processing order of all the elements. In any case, whenever the data front respects the given reading order, the whole array is

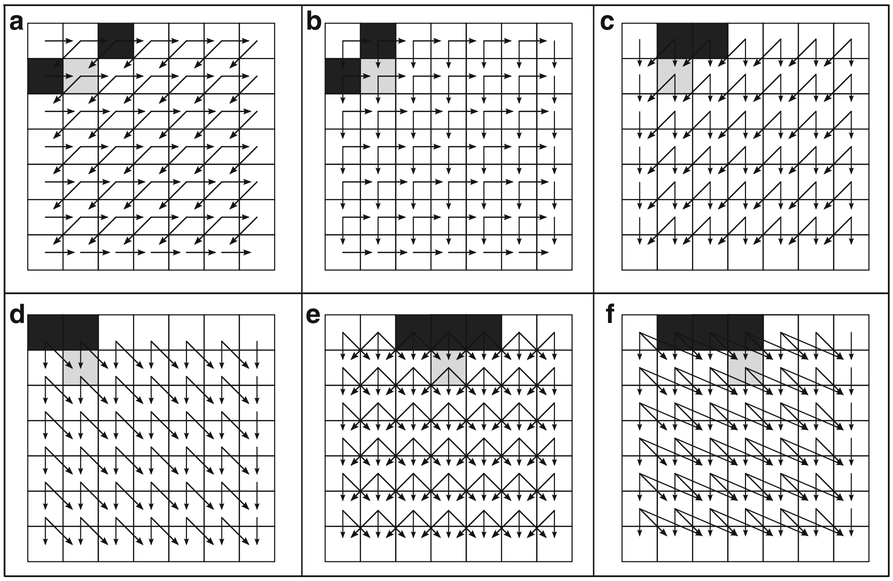


**Fig. 8** Possible wavefront dependence schemes

parallelized while respecting data dependencies among the elements through the entire array. A popular example of the wavefront pattern is the diagonal model, where computation starts at the top, left corner of an array of data, and then the data front is propagated diagonally until the right bottom corner is reached. This is the pattern followed, for instance, in the modified DF proposed in this chapter. The same pattern shows up every time when the computation of each data element depends on its upper and right neighbors. In this particular scenario, all the elements that belong to the antidiagonal can be processed in parallel. However, the wavefront pattern is not limited to this diagonal shape. Some of those examples are shown at Fig. 8, including also horizontal data fronts, with each element depending on two or three cells simultaneously.

Each of those patterns shown in Fig. 8 corresponds to several problems, where the data dependencies have been evaluated to parallelize their execution by following the wavefront pattern, as shown in [7].

According to the schemes of dependencies described in Fig. 8, each case differs on the shape of the data front or the order in which the data fronts are explored. It is always possible to process all the elements located in these fronts at the same time, by means of assigning each element to a different FU in the processing array. Moreover, the rule established in the case of the DF algorithm to arrange the data among the FUs can be kept without losing generality. This means that the first element in the data front must be always processed in the first FU of the array and the rest of elements are distributed throughout subsequent FUs of the architecture. The number of elements which can be processed in parallel coincides with the maximum number of FUs implemented in the array.

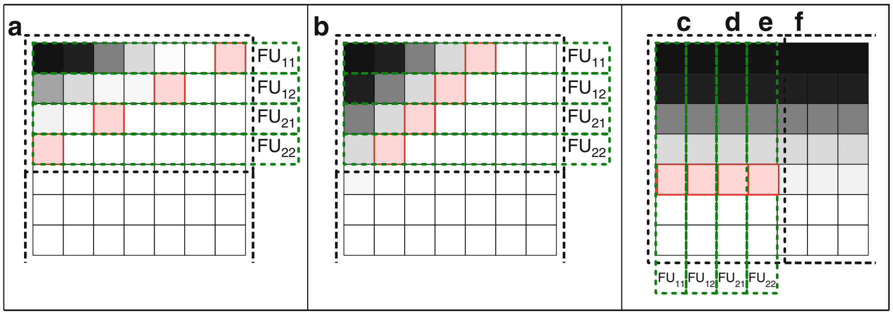


**Fig. 9** Possible wavefront patterns

Regarding the characterization of the architecture, for each neighboring element of which a given one depends, a vector with two components is defined. The first component ($D_t$) describes the distance, in terms of processing cycles, when both the current and the referenced blocks are processed. Regarding the second one ($D_{\mathrm{FU}}$), it defines the distance between the FUs in charge of processing both blocks:

$$D = (D_t, D_{\mathrm{FU}}). \tag{1}$$

Thus, $D_t$ is an integer greater than one, and $D_{\mathrm{FU}}$ is an integer which can take both positive and negative values, depending on whether the FU in charge of processing the neighboring block occupies an upper or lower position in the array, respectively.

For instance, in the case of the Fig. 8a, each element (except those located in the borders) depends on its left and upper-right neighbors. Considering both the parallelization and the FU allocation diagrams shown in Fig. 9, the vector defining the dependence with respect to the left neighbor is $D_1 = (1{,}0)$, since it is processed always in the same FU, but during the previous processing cycle. Regarding the upper-right neighbor, the vector is $D_2 = (1{,}1)$, since it is always processed in the previous functional unit, during the previous cycle. The same vectors can be drawn in the case (b) in spite of having different data dependencies. For the other cases, existing relationships are described below.

For (c): $D_1 = (1{,}0)$, corresponding to the upper neighbor and $D_2 = (1,-1)$, corresponding to the upper-right neighbor

For (d): $D_1 = (1{,}0)$, corresponding to the upper neighbor, and $D_2 = (1{,}1)$, corresponding to the upper-left neighbor

For (e): $D_1 = (1{,}1)$, corresponding to the upper-left neighbor, $D_2 = (1{,}0)$, corresponding to the upper neighbor, and $D_3 = (1,-1)$, corresponding to the upper-right neighbor

For (f): $D_1 = (1{,}0)$, corresponding to the upper neighbor, $D_2 = (1{,}1)$, corresponding to the upper-left neighbor, and $D_3 = (1{,}2)$, corresponding to the upper-left-left neighbor

Regardless the specific dependence pattern, the data mesh is always processed in independent disjoint blocks, corresponding to the dashed box shown in Fig. 9.

## 9.2 Architecture Customization

As it was explained above, the proposed architecture might be generalized to tackle different kinds of problems, such as those characterized by data fronts offered in Fig. 9, by reusing and adapting its structure. This section explains the main changes that should introduce into all the set of modules belonging to the architecture, as well as in its management principles, in order to adapt their behavior and functionality to the new conditions.

**Functional Unit** These modules are the processing cores of the architecture, and it is compulsory to adapt them in order to address different processing problems. In the same manner the specific design of the processing units for the DF is out of the scope of this chapter, the design of the specific units for other problems will not be offered in this section, and they can even be reused from existing nonparallel implementations. Therefore, this chapter provides a general scalable framework suited to parallelize the hardware execution of these tasks.

In addition of changing FUs behavior, it is necessary to adapt their internal memories to rearrange data blocks according to the application demands. Considering that each independent memory is able to store a basic data unit, the number and type of the required memories will be a consequence of the dependence vectors described in the previous section. Thus, two kinds of memories called $M_p$ and $M_q$ are defined. On the one hand, $M_p$ memories store the required data blocks received from the upper and lower FUs, in previous processing cycles, using the FU to TU direct connections. That is, those memories store data blocks with a dependence vector with $D_{\mathrm{FU}} \neq 0$. On the other hand, $M_q$ memories store both the current block and previous blocks processed in the same unit. That is, dependencies with $D_{\mathrm{FU}} = 0$. The general sequence of operations that has to be done in each functional unit are

$M_q{}^0 = DataIN$
$DataTemp = FU(M_p{}^0, \ldots, M_p{}^{p_{max}}, M_q{}^0, \ldots, M_q{}^{q_{max}})$
$M_q{}^{i+1} = M_q{}^i, \forall\, i\ in\ (1, q_{max}\text{-}1)$
$M_p{}^{i+1} = M_p{}^i, \forall\, i\ in\ (0, p_{max}\text{-}1)$
$M_q{}^1 = DataTemp$
$M_p{}^0 = DataInVertical$

*DataOut* and *DataOutVertical* might be equal to *DataTemp*, to *DataIn*, or even partial results of the processing task. In addition, the processing stage might be divided into subsequent phases, carrying out data sharing in between, like in the case of the deblocking filter algorithm (H and V phases). In those cases, temporal memories have to be included in the architecture. With respect to *DataInVertical* and *DataOutVertical*, whether more than one dependence would exist with a fixed $D_t$ and $\mid D_{\mathrm{FU}} \mid > 0$, several *DataInVertical* and *DataOutVertical* channels will exist.

**Data Reading Order** The main purpose of the IC is to read unprocessed data from the external memory. Before reading these data, it is necessary to generate the sequence of addresses that defines the order in which these data will be requested from the external memory. Due to the fact that the data dependencies vary with

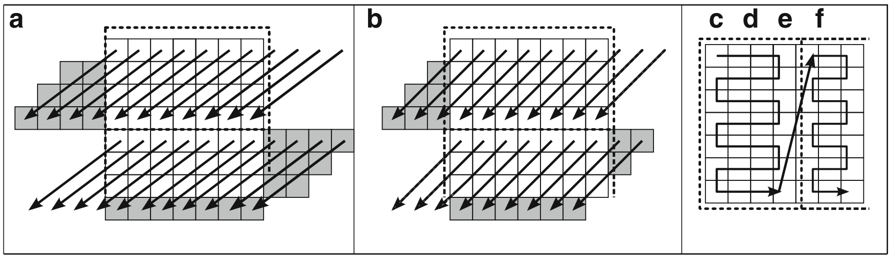


**Fig. 10** Possible wavefront processing structures

any new data front, the reading addresses have to be adapted to fulfill with the data parallelization scheme. This change can be performed by modifying the internal rules that defines the address generation process inside the IC. However, the general structure of the address generation can remain unaltered.

**Data Block Allocation and Distribution** The allocation strategy of unprocessed data blocks inside the architecture might be kept independently of the wavefront processing problem. Whether the data reading sequence is modified according to the previous section, the data distribution rule across the array can be kept as well. Thus, each IM will hold the $N$ first input data, being $N$ the number of vertical FUs, and will transmit the subsequent ones to the subsequent FUs. Regarding the routers, each one will keep the first data, transmitting the rest to the following FUs. This approach guarantees that each unit is fed with the required data. Once the data have been processed, they are transmitted to the external memory without carrying out further modifications to the current architecture. In the proposed architecture, all the FUs will work in a synchronous way, that is, all the units will be in the same state during each time period. Therefore, null data blocks have to be used in order to deal with dead times, as is shown in gray in Fig. 10.

**Semiprocessed Data Sharing** As depicted in Fig. 9, different data reuse requirements arise from every pattern of data dependencies. One of the main features of the proposed architecture is that it is able of tackling diverse kinds of data dependencies by exploiting local point-to-point connections, without requiring accesses to the external memory. The term semiprocessed refers to those data that will be used again after being processed in order to process other data. The number of times and the order in which these semiprocessed data are required depend on the data dependencies. In the case of the DF, to process each MB, the left neighbor is stored in the same FU, while the upper one is received from the upper FU. This strategy can be modified if it is necessary to include data dependencies with the element processed in the FU below [for instance, required in the example (e)], as well as changing the number of cells depending in each direction, north, south, or horizontal.

Considering the description vectors introduced in previous sections, the number of vertical connections is equal to the number of dependencies with a fixed $D_t$ and $|D_{\mathrm{FU}}| > 0$. In case $D_{\mathrm{FU}} > 0$, those dependencies pass through the array from north to south, and, in case $D_{\mathrm{FU}} < 0$, from south to north. Together with the channels through the FUs, a different number of specific resources have to be included in the IC to store temporal data, belonging to different disjoint set of blocks. Mainly, it is necessary to include a FIFO memory and an FSM for controlling the process, corresponding to each vertical line implemented in the scalable architecture.

## *9.3 H.264/SVC Generalization*

The MB wavefront parallelization pattern existing into the DF is not exclusive of this algorithm. Many functional blocks of the H.264 video coding standard, except for the entropy decoder, might be adapted to follow the same pattern [24]. In the case of the DF, the pattern of dependence is the one shown in the Fig. 8 a). Therefore, considering the dependence vectors, both one $M_p$ and one $M_q$ memories are required. In addition, another $M_q$ is used to store the current MB, and an extra memory is also required to store temporal semifiltered data between the H and V filtering phases. In addition, the FU provides two results. The basic operations carried out in each functional unit are described below, following the general template described in the previous sections.

$M_q{}^0 = DataIN$
$(DataTemp1, DataTemp2) = FU(M_q{}^0, M_q{}^1)$
DataOutVertical= DataTemp2
$M_p{}^0 = DataInVertical$
$(DataTemp1, DataTemp2) = FU(DataTemp1, M_p{}^0)$
DataOut = DataTemp1
$M_q{}^1 = DataTemp2$

In the general case of H.264, possible dependencies at a MB-level are described in [24]. In this general case, the dependence scheme can include, in addition of the elements required in the DF, the upper-right and the upper-left macroblock. Those elements are used in certain modes of the *intra prediction* and the *motion vector*. Therefore, the dependence vectors are

- $D_1 = (1,0)$, corresponding to the left neighbor
- $D_2 = (1,1)$, corresponding to the upper-right neighbor
- $D_2 = (2,1)$, corresponding to the upper neighbor
- $D_2 = (3,1)$, corresponding to the upper-left neighbor

In consequence, two $M_q$ memories are required to implement those blocks, one for the current MB and the other for the upper neighbor, with dependence vector (1,0). With respect to $M_p$ memories, 3 units are required, as well as a single descendent communication channel.

## 10 Implementation Details and Results

This section collects some implementation results obtained after the synthesis and the implementation stages, considering both the modular architecture itself and the DPR issues.

### *10.1 Architectural Details*

Within the reconfigurable stage of this architecture, the array might be formed by a different number of FUs. In this section, the architecture has been implemented without considering DPR issues, that is, different sizes are achieved after a new synthesis process of the full architecture. The amount of these units varies according to the performance or the environment constraints on demand. Depending on how the FUs are distributed into the processing array (M×N), different configurations are obtained, in which M and N are the width and height of the array. For a specific amount of FUs there exist several configurations, characterized by having the same computational performance, but different HW resources demands and different data transfers delays. More in detail, the resource occupancy of each basic block of the proposed architecture is shown in Table 1.

Regarding to synthesis results, the FU limits the maximum operation frequency of the whole architecture up to 124 MHz. On the other hand, performance variations are shown in Table 2. These data mean the minimum operation frequency that is necessary for processing different video formats at 30 frames per second (fps) with real-time constraint. Each column is referred to a determined number of FUs, while each row expresses frequency values for a specific video format (W×H), where W and H are its width and height in pixels, respectively.

Using many FUs is not efficient for processing the smallest video formats since some units will keep idle during filtering execution. The maximum number of FUs is determined by the height of the frame expressed in pixels (H). As an example, SQCIF and QCIF formats are 96 and 144 in height respectively; as a consequence, configurations with more than 6 or 9 FUs are not appropriated for these formats. Otherwise, using a low number of FUs is not possible to process the highest multimedia formats, like UHDTV, since its associated configurations need more than 200 MHz for real-time performance, whereas the maximum frequency of this architecture is 124 MHz.

**Table 1** Resources occupancy

| | Synthesis results using V5LX110T | | | | |
|---|---|---|---|---|---|
| | IC | OC | IM | OM | RouterFU |
| Slices reg. | 357 | 116 | 172 | 124 | 2004 |
| Slices LUTs | 444 | 108 | 134 | 226 | 2386 |
| BlockRAM/FIFO (36 kb) | 1 | 0 | 2 | 2 | 8 |

**Table 2** Maximum frequency for real time (30 FPS)

| | Maximum frequency (KHz) | | | | | |
|---|---|---|---|---|---|---|
| Format @30 fps (W×H) | 1 FU | 2 FUs | 3 FUs | 4 FUs | 8 FUs | 16 FUs |
| SQCIF (128×96) | 346 | 180 | 130 | 122 | NA | NA |
| QCIF (176×144) | 713 | 396 | 252 | 238 | 158 | NA |
| CIF (352×288) | 2,851 | 1,432 | 964 | 799 | 482 | 324 |
| 4CIF (704×576) | 11,400 | 5,709 | 3,816 | 2,556 | 1,936 | 972 |
| 16 CIF (1408×1152) | 45,619 | 22,816 | 15,220 | 11,426 | 5,752 | 3,218 |
| HDTV (1920×1072) | 58,320 | 29,383 | 19,879 | 14,709 | 7,797 | 4,341 |
| UHDTV (7680×4320) | 933,120 | 466,567 | 311,054 | 235,010 | 117,540 | 58,845 |

**Table 3** Resources impact of designing for dynamic scalability

| | Elements of the architecture | | |
|---|---|---|---|
| Logical resources | IM | RouterFU | OM |
| Slices LUTs | 2080 | 4160 | 2080 |
| Slices registers | 2080 | 4160 | 2080 |
| Block RAM/FIFO (36 kb) | 4 | 8 | 4 |

**Table 4** Resources occupancy of the full reconfigurable array

| | Size | | | | |
|---|---|---|---|---|---|
| Logical resources | 1×2 | 1×3 | 2×1 | 2×2 | 2×3 |
| Slices LUTs | 12480 | 16640 | 16640 | 24960 | 33280 |
| Slices registers | 12480 | 16640 | 16640 | 24960 | 33280 |
| Block RAM/FIFO (36 kb) | 24 | 32 | 32 | 48 | 64 |

## *10.2 Dynamic Reconfiguration Details*

In this section, details regarding to the reconfigurability of the architecture are explained. Adapting the architecture to be dynamically scalable implies the addition of BMs, as well as the design of an independent floor planing for each module. As it is shown in the previous section, each module has been designed occupying extra area in order to be able to route all internal signals within the reconfigurable region. Thus, even though not all logical resources within the region are occupied, the entire region is necessary to come up with a fully routed design. Consequently, resource occupancy is increased, as it can be seen in Table 3, regarding each element, and in Table 4, regarding different sizes of the full core. Information in Table 3 can be compared with Table 1 to see the overhead of designing the architecture to DPR. Thus, the area of the IM is increased about 15 times, the OM about 9 times, as well as 1.7 times for the case of the functional unit.

Values of area occupancy for each element directly entail the size of the corresponding bitstreams and, consequently, the DF reconfiguration time itself. These aspects are evaluated in the following section.

**Table 5** Reconfiguration time

| | Size | | | | |
|---|---|---|---|---|---|
| | 1×2 | 1×3 | 2×1 | 2×2 | 2×3 |
| Reconfiguration time (us) | 620 | 832 | 834 | 1249 | 1663 |

## *10.3 Reconfiguration Time*

The DF reconfiguration time is a consequence of the area occupied by each module of the architecture. More specifically, it depends on the type of configurable resources used by its modules, since the number of *frames*, the basic reconfigurable units within an FPGA, is different for BRAMS, DSPs, or CLBs columns.

Reconfiguration times are offered in Table 5 for the case of creating each architecture from the scratch. The operating frequency of the reconfiguration engine is 200 MHz.

Values shown in Table 5 already include the latency due to the software API of the reconfiguration engine, which depend on the number of reconfiguration operations that have to be carried out.

Reconfiguration time is about one or two orders of magnitude above processing time for each video frame. This fact has to be taken into account when implementing the policy inside the decoder, deciding when to modify the size of the architecture.

## *10.4 Performance Limits*

The main constraint on the size of the architecture has to do with the relationship between the time each unit requires to process a single data unit ($\mathrm{T_p}$) and the time required to transmit input data to the FUs. Thus, only if $T_p < T_{load} \times N$, being $N$ the number of FUs in each column of the processing array, no overhead is introduced in the performance of the core. To fulfill this condition, input/output data channels have to be dimensioned according to the block data size.

In the case of the DF, the size of the data unit, including all the information required to process each MB, is 108 words of 32 bits. Therefore, 64-bit-width data connections have been included horizontally, between IMs and OMs, and vertically, between routers, to allow enabling up to three rows of functional units without compromising the DF performance.

## 11 Conclusions and Future Work

This chapter addresses the design of spatially scalable architectures which are able to adjust their size at run time, in terms of the number of elements that are implemented onto the FPGA, by means of the DPR feature of commercial FPGAs.

The exploitation of this feature allows achieving a variable data level parallelism that adjusts its properties to the performance fluctuations demanded by the running application. Thus, the area-performance trade-off can be balanced at run time. This chapter takes advantage of these benefits by developing a dynamically scalable deblocking filter architecture, where its number of computation units (FUs) might be adapted online to fulfill the variable requirements of different profiles and scalability levels of H.264/AVC and SVC video coding standards.

Given a frame size as well as certain DF dimensions, each FU will always process the same MBs. This strategy simplifies DF control, but a new video frame cannot be processed until the previous one has been completely filtered, introducing an extra overhead. In addition, memory consumption and its distribution within the FU will be also optimized, reducing the area of each FU module. This improvement will also impact results of designing for DPR, since routing inside each module will be simplified. Accordingly, FUs will be floorplanned in narrower regions, looking for the homogeneity of both FU columns. Furthermore, results from the output controller will be sent upward to the input controller instead of being transmitted to the static region. Thus, DF will have a unique communication point with the rest of the system.

**Acknowledgements** This chapter is supported by the Spanish Ministry of Science and Innovation and European Union (FEDER funds) in the context of Dynamic Reconfigurability for Scalability in Multimedia Oriented Networks (DR. SIMON) project, under contract TEC2008-065846-C02. On the other hand, the development of the reconfiguration engine was supported by the Artemis program under the project SMART (Secure, Mobile Visual Sensor Networks Architecture) with number ARTEMIS-2008-100032.

## References

1. Banerjee S, Bozorgzadeh E, Dutt N (2009) Exploiting application data-parallelism on dynamically reconfigurable architectures: placement and architectural considerations. IEEE Transactions on very large scale integration (VLSI) systems 17(2):234–247
2. Becker J, Hubner M, Hettich G, Constapel R, Eisenmann J, Luka J (2007) Dynamic and partial FPGA exploitation. Proceedings of the IEEE, 95(2):438–452
3. Cervero T, Otero A, Lpez S, De La Torre E, Callic G, Sarmiento R, Riesgo T (2011) A novel scalable deblocking filter architecture for H.264/AVC and SVC video codecs. In: Proceedings of the 2011 IEEE international conference on multimedia and expo (ICME) 2011
4. Cervero T, Otero A, De la Torre E, Lpez S, Callic G, Riesgo T, Sarmiento R (2011) Scalable 2D architecture for H.264 SVC deblocking filter with reconfiguration capabilities for on-demand adaptation. Proceedings of the SPIE, vol 8067, April 2011 (in press)
5. Chang-Seok Choi, Hanho Lee (2006) An reconfigurable FIR filter design on a partial reconfiguration platform. In: 1st international conference on communications and electronics. ICCE ’06., pp 352–355, 10–11 October 2006
6. Corbetta S, Morandi M, Novati M, Santambrogio MD, Sciuto D, Spoletini P (2009) Internal and external bitstream relocation for partial dynamic reconfiguration. IEEE Transactions on very large scale integration (VLSI) systems, I, 17(11):1650–1654

7. Dios A, Asenjo R, Navarro A, Corbera F, Zapata EL (2011) Wavefront template for the task-based programming model. In: The 24th international workshop on languages and compilers for parallel computing (LCPC 2011), Colorado State University, Fort Collins, Colorado, Sept 8–10 2011
8. Espeland H, Beskow PB, Stensland HK, Olsen PN, Kristofferson S, Griwodz C, Halvorsen P (2011) P2G. A framework for distributed real-time processing of multimedia data. In: Conference on parallel processing workshop, pp 416–426
9. Marc Snir (20xx) Wavefront patter (Dynamic Programming). In: Resources on Parallel Patterns, Pattern collection. Available at: http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf. Accessed 2011
10. ITU-T Rec. H.264, ISO/IEC 14496–10. H.264/AVC extension (Scalable Video Coding - SVC). Advanced Video Coding for Generic Audiovisual Services, Version 8: 2007/Version 10: 2009
11. Jian Huang, Jooheung Lee (2009) A self-reconfigurable platform for scalable DCT computation using compressed partial bitstreams and BlockRAM prefetching. IEEE Trans Circ Syst Video Tech 19(11):1623–1632
12. Khraisha R, Jooheung Lee, (2010) A scalable H.264/AVC deblocking filter architecture using dynamic partial reconfiguration. In: IEEE international conference on acoustics speech and signal processing (ICASSP), pp 1566–1569, 14–19 Mar 2010
13. Kurafuji T, Haraguchi M, Nakajima M, Nishijima T, Tanizaki T, Yamasaki H, Sugimura T, Imai Y, Ishizaki M, Kumaki T, Murata K, Yoshida K, Shimomura E, Noda H, Okuno Y, Kamijo S, Koide T, Mattausch HJ, Arimoto K (2011) A scalable massively parallel processor for real-time image processing. IEEE Journal of solid-state circuits 46(10):2363–2373
14. Like Y, Yuan W, Tianzhou C (2010) Input-driven reconfiguration for area and performance adaption of reconfigurable accelerators. In: IEEE 13th international conference on computational science and engineering (CSE), 11–13 December 2010, pp 237–244
15. List P, Joch A, Lainema J, Bjontegaard G, Karczewicz M (2003) Adaptive deblocking filter. IEEE Trans Circ Syst Video Tech 13(7):614–619
16. Ostermann J, Bormans J, List P, Marpe D, Narroschke M, Pereira F, Stockhammer T, Wedi T (2004) Video coding with H.264/AVC: tools, performance, and complexity. IEEE Circ Syst Mag 4(1):7–28
17. Otero A, de la Torre E, Riesgo T, Krasteva YE (2010) Run-time scalable systolic coprocessors for flexible multimedia SoPCs. In: International conference on field programmable logic and applications (FPL), pp 70–76, Aug 31, 2010–Sept 2, 2010
18. Otero A, Morales-Cas, A, Portilla J, de la Torre E, Riesgo T (2010) A modular peripheral to support self-reconfiguration in SoCs. In: 13th Euromicro conference on digital system design: architectures, methods and tools (DSD), pp 88–95, 1–3 September 2010
19. Xilinx, Inc (2011) Partial Reconfiguration User Guide V12.1. In: Product Support and Documentation. Available at: http://www.xilinx.com/support/documentation\swmanuals/xilinx12.1/ug702.pdf. Accessed 2011
20. Xilinx, Inc (2011) Processor Local Bus description. In: CoreConnect Architecture. Available at: http://www.xilinx.com/ipcenter/processorcentral/coreconnect/\coreconnectplb.htm. Accessed 2011
21. Salih MH, Arshad MR (2010) Design and implementation of embedded multiprocessor architecture using FPGA. IEEE simposium on industrial and applications, pp 579–584
22. Schwarz H, Marpe D, Wiegand T (2007) Overview of the scalable video coding extension of the H.264/AVC Standard. IEEE Trans Circ Syst Video Tech 17(9):1103–1120
23. Sedcole P, Blodget B, Becker T, Anderson J, Lysaght P (2006) Modular dynamic reconfiguration in virtex FPGAs. Computers and digital techniques, IEE Proceedings, 153(3):157–164
24. Van der Tol, Erik B.; Jaspers, Egbert G (2003) Mapping of H.264 decoding on a multiprocessor architecture, In: Proceedings SPIE 5022:707–718 (Image and Video Communications and Processing 2003)

25. Weiguo Liu, Schmidt B (2003) Parallel design pattern for computational biology and scientific computing applications. In: Proceedings of IEEE international conference on cluster computing, 2003
26. Wiegand T, Sullivan GJ, Bjontegaard G, Luthra A (2003) Overview of the H.264/AVC video coding standard. IEEE Trans Circ Syst Video Tech 13(7):560–576
27. Xilinx, Inc (2011) Bus Macros description. Application Note [Online] Available at: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev//0038_8.html#wp1103560 Accessed 2011
28. Xilinx, Inc (2011) Module-Based Partial Reconfiguration. Application Note [Online]. Available at: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev//0038_8.html. Accessed 2011

# CAPH: A Language for Implementing Stream-Processing Applications on FPGAs

**Jocelyn Sérot, François Berry, and Sameer Ahmed**

## 1 Introduction

Stream-processing applications—i.e., applications operating on the fly on continuous streams of data—generally require a high computing power. This is especially true for real-time image processing, for instance, in which this computing power is in the range of billions of operations per second, often still beyond the capacity of general-purpose processors (GPPs). Most of the computationally demanding tasks in these applications exhibit parallelism, making them good candidates for implementation on reconfigurable logic such as field programmable gate arrays (FPGAs).

But, in the current state of the art, programming FPGAs essentially remains a hardware-oriented activity, relying on dedicated hardware description languages (such as VHDL or Verilog). Using these languages requires expertise in digital design, and this practically limits the applicability of FPGA-based solutions.

As a response, a lot of work has been devoted in the past decade to the design and development of *high-level* languages and tools, aiming at allowing FPGAs to be used by programmers who are not experts in digital design. Fueled by new behavioral synthesis techniques and ever-increasing FPGA capacities, significant advances have been made in this area. But there is still a gap between what can be described with a general-purpose, Turing-complete, language and what can be efficiently and *automatically* implemented on an FPGA. In this context, we believe that a *domain-specific language* (DSL) can provide a pragmatic solution to this problem.

J. Sérot (✉) • F. Berry • S. Ahmed
Universite Blaise Pascal, Institut Pascal, UMR 6602 CNRS/UBP,
24 Avenue des Landais, F-63171 AUBIERE cedex 1, France
e-mail: Jocelyn.Serot@univ-bpclermont.fr; Francois.Berry@univ-bpclermont.fr; Sameer.Ahmed@univ-bpclermont.fr

In this chapter, we introduce such a DSL, CAPH.[1] By adopting a specific (purely dataflow) model of computation, CAPH aims at reducing the gap between the programming model (as viewed by the programmer) and the execution model (as implemented on the target hardware) and hence obtaining an acceptable trade-off between abstraction and efficiency requirements.

The remainder of this chapter is organized as follows: In Sect. 2, we recall the issues raised by FPGA programming, make a brief survey of some existing solutions and explain why they are not satisfactory. Section 3 presents the general dataflow-/actor-oriented model of computation as a possible solution to the aforementioned issues. Section 4 introduces the CAPH language as a specific incarnation of the general dataflow approach. Section 5 is an overview of the suite of tools supporting the CAPH language. Sections 6–8 give a short and basic account on how the compiler works in order to generate efficient VHDL and SystemC code. Preliminary experimental results are presented in Sect. 9. Related work is described and discussed in Sect. 10. Finally, conclusions are drawn in Sect. 11.

## 2 High-Level Languages for FPGA

Hardware description languages (HDLs), such as VHDL or Verilog, are still widely used for programming FPGAs because they provide flexible and powerful ways to generate efficient logic. But these languages were designed specifically for hardware designers, which makes them unfamiliar for programmers outside this field. To circumvent this problem, a number of tools have been proposed—both from the academic community and the industry—aiming at offering a higher-level programming model for reconfigurable devices.

Many of these tools propose a direct conversion of C code into (V)HDL. These include Handle-C [8], Stream-C [3], SA-C [14], SPARK [6] and Impulse-C [9]. These approaches suffer from several drawbacks. First, C programs sometimes (often) rely on features which are difficult, if not impossible, to implement in hardware (dynamic memory allocation, for instance). This means that code frequently has to be rewritten to be accepted by the compilers. Practically, this rewriting cannot be carried out without understanding why certain constructs have to be avoided and how to replace them by "hardware-compatible" equivalents. So, a minimum knowledge of hardware design principles is actually required. Second, C is intrinsically sequential whereas hardware is truly parallel. So, the compiler has to first identify parallelism in the sequential code and then map it onto the target hardware. In the current state of the art, this cannot be done in a fully automatic way, and the programmer is required to put annotations (pragmas) in the code to help the compiler, which adds to the burden. Finally, the code generally has to undergo

[1]CAPH is a recursive acronym for **C**aph just **A**int **P**lain **H**dl. It is also the name of the second brightest star in the constellation of Cassiopeia.

various optimizations and transformations before the actual HDL generation. These optimizations and transformations vary from high-level parallelization techniques to low-level scheduling. The low-level optimizations can be beneficial to any algorithm, but the high-level optimizations are specifically suggested in the context of one field and would not give performance gains in other domains [20]. Moreover, with most of the existing tools (Handle-C, Impulse-C, Catapult-C), transformations and optimizations require inputs from the programmer [21], who therefore must have a rather good knowledge in digital design.

## 3 Dataflow-/Actor-Oriented Paradigm

We claim that the solution to the problems raised by C-like approaches to FPGA programming requires a shift in programming paradigm. In particular, it seems crucial to reduce the gap between the *programming* model (as viewed by the programmer) and the *execution* model (as implemented on the target hardware). The dataflow/actor paradigm offers a way to achieve this goal. This section recalls the key features of this paradigm and why it is naturally suitable for FPGA or reconfigurable devices.

The common and basic underlying concept is that applications are described as a collection of computing units (often called actors) exchanging streams of tokens through unidirectional channels (typically FIFOs). Execution occurs as tokens flow through channels, into and out of actors, according to a set of firing rules. In the classical, strict, dataflow model, the firing rules specify that an actor becomes active whenever tokens are available on all of its input channels, and token(s) can be written on its output channel(s). When this occurs, input tokens are consumed; result(s) are computed and produced on the output channel(s). This strict firing model has been latter extended to accommodate more complex and sophisticated scheduling strategies (see Sect. 10).

The basic dataflow model is illustrated in Fig. 1 with a very simple example, involving four basic actors. Actor `inc` (resp. `dec`) adds (resp. subtracts) 1 to each element of its input stream. Actor `mul` performs point-wise multiplication of two streams. Actor `dup` duplicates its input stream.[2] Now, if we connect these four actors to build the network depicted in Fig. 2, this network computes $f(x) = (x+1) \times (x-1)$ for each element $x$ of its input stream. That is, if the input stream $i$ is `1,2,3,...`, then the output stream $o$ will be `0,3,8,...`.

The advantages of the dataflow model are well known. First, it basically relies on a representation of applications in the form of *dataflow graphs* (DFGs), which

[2]In Fig. 1, streams are denoted (ordered) from right to left; for example, the actor `ADD` first processes the token `1`, then the token `2`, *etc*. Since streams are potentially infinite, their end is denoted "`...`". However, when describing actors *textually*, streams will be denoted from left to right; for example, `ADD:1,2,...= 2,3,...`

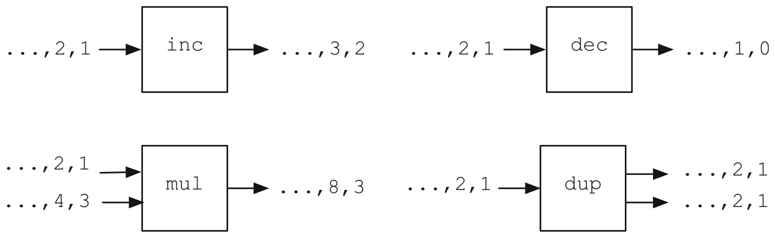


**Fig. 1** Four basic actors

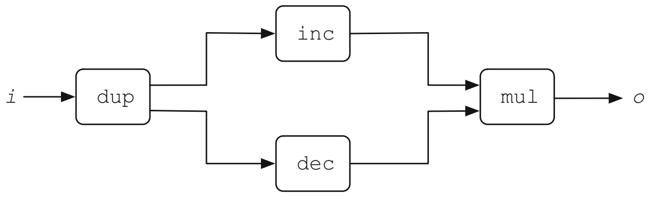
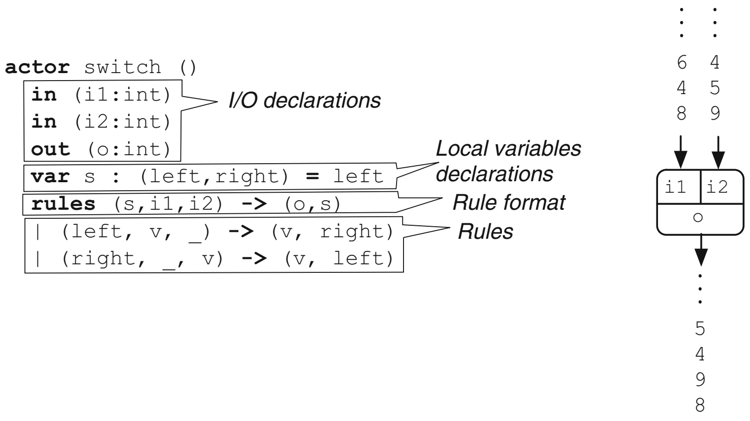


**Fig. 2** A dataflow process network

is intuitive, well understood by programmers (esp. in the field of digital signal processing) and amenable to graphical representation and manipulation. Second, since the behavior of each actor is defined independently of what occurs in the other ones, all parallel operations may be executed concurrently, without the risk of side effects. This allows a full exploitation of intrinsic data and control-level parallelism. Third, since several tokens are allowed to reside simultaneously on a channel (by implementing it using a FIFO typically), execution pipelining may be increased, each actor being able to produce one output token before the previous one has actually been consumed.

## 4 CAPH Language

The dataflow model of computation introduced in the previous section is indeed a very general one, from which many specific instances can be drawn, depending, in particular, on the kind of behavior that can be assigned to actors, the exact nature of tokens exchanged by actors and the way DFGs (networks) are described. In the sequel, we describe the design choices made for the CAPH language regarding these issues. For the sake of brevity and conciseness, the related sections only give an informal description of the main features of the language. The complete language definition, including concrete and abstract syntax and formal semantics, can be found in the *language reference manual* [15].



```
actor switch ()
  in (i1:int)
  in (i2:int)
  out (o:int)
  var s : (left,right) = left
  rules (s,i1,i2) -> (o,s)
  | (left, v, _) -> (v, right)
  | (right, _, v) -> (v, left)
```

Fig. 3 An example of actor description in CAPH

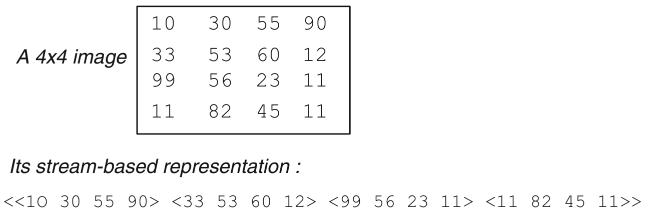
## 4.1 Describing Actors

In CAPH the behavior of actors is specified using a set of *transition rules*. Each rule consists of a set of *patterns*, involving inputs and/or local variables and a set of *expressions*, describing modifications of outputs and/or local variables. The choice of the rule to be fired is done by pattern matching.

Consider, for example, the actor `switch` described in Fig. 3. This actor merges two data streams by alternately copying its first and second input to its output (as depicted on the right). Its behavior description in CAPH is given on the left. It has no parameter, two inputs and one output (of type `int`). For this, it uses a local variable (`s`), which can take only two values (`left` or `right`). The general format of the rule set is defined in the line starting with the `rules` keyword. Here, each rule pattern is a triplet made of the values of the local variable and the two inputs, respectively; when a rule is fired (selected) it updates the variable `s` and produces a value on the output `o`. The first (resp. second) rule says: If `s` is `'left'` (resp. `'right'`) and a value `v` is available on input `i1` (resp. `i2`) then read[3] this value, write it to output `o`, and set `s` to `'right'` (resp. `'left'`). The `'_'` symbol used in the pattern means that the corresponding input is not used.[4]

[3]Pop the value from the connected FIFO.

[4]The `'_'` symbol can also be used in the right-hand side of a rule; it then means that no value is produced on the corresponding *output*.


A 4x4 image

|  |  |  |  |
|---|---|---|---|
| 10 | 30 | 55 | 90 |
| 33 | 53 | 60 | 12 |
| 99 | 56 | 23 | 11 |
| 11 | 82 | 45 | 11 |

Its stream-based representation :

```
<<1O 30 55 90> <33 53 60 12> <99 56 23 11> <11 82 45 11>>
```

**Fig. 4** The structured stream representation of a 4×4 image

## *4.2 Data Representation*

A key property for a programming model is its ability to represent arbitrarily structured data. For stream-processing applications, this structuring can be achieved by dividing the tokens, circulating on channels and manipulated by actors, into two broad categories: *data* tokens (carrying actual values) and *control* tokens (acting as structuring delimiters). In fact, only two control tokens are required: one for denoting the start of a structure (list, line, frame, *etc.*), which will be denoted `SoS` or `'<'`, and another for the end of the structure, denoted `EoS` or `'>'`. For example, an image can be described as a list of lines, as depicted on Fig. 4, whereas the stream

```
<<<41 120> 44><<12 73> 58><<52 211> 7>>
```

may represent, for example, a list of points of interest, each inner pair consisting of its coordinates along with an attribute value.

This structured representation of data nicely fits the stream-processing programming and execution models. Since the structure of the data is explicitly contained in the token stream, no global control and/or synchronization is needed; this has strong and positive consequences both at the programming level (it justifies *a posteriori* the style of description we introduced in the previous subsection for actors) and the execution level (it will greatly ease the production of HDL code). Moreover, it naturally supports a pipelined execution scheme; processing of a line by an actor, for example, can begin as soon as the first pixel is read without having to wait for the entire structure to be received; this feature, which effectively allows concurrent circulation of successive "waves" of tokens through the network of actors, is of course crucial for on-the-fly processing (like in real-time image processing).

Figures 5 and 6 give two examples of actor descriptions based on this data representation.

The actor described in Fig. 5 performs binarization of a structured stream of pixels (an image, for instance). The threshold value is passed as parameter `t`. Pattern-matching is used to discriminate between control and data tokens. The rules can be read as follows: if input is a *control* token (`'<` or `'>`), then write the same token on output; if input is a *data* token, then write 0 or 1 on output depending on whether the associated value is greater than the threshold parameter or not. The quote (`'`) symbol is used to distinguish *structured* values (control or data) from

**Fig. 5** An actor performing binarization on structured streams

```
actor thr (t:unsigned<8>)
  in (a:unsigned<8> dc)
  out (c:unsigned<8> dc)
  rules a -> c
  | '< -> '<
  | '> -> '>
  | 'v -> if v > t then '1 else '0
```

**Fig. 6** An actor computing the sum of values along lists

```
actor suml ()
  in (a: signed<8> dc)
 out (c: signed<16>)
 var st: {S0,S1}=S0
 var s : signed<16>
 rules (st,a,s)-> (st,c,s)
   (S0, '<, _) -> (S1, _, 0)
 | (S1, 'v, s) -> (S1, _, s+v)
 | (S1, '>, s) -> (S0, s, _)
```

unstructured (raw) values (like in the previous `switch` example). This distinction is reflected in the type of the input and output: `unsigned<8> dc`, where `dc` is the type constructor for structured values. Hence, the last rule of the `thr` actor should be read, precisely, as follows: if a value is available on input `a` and this value is a *data* token carrying value `v`, then produce a *data* token on output carrying value `0` or `1` depending on whether `v>t` or not.

The actor described in Fig. 6 computes the sum of a list of values. Given the input stream `<1 2 3> <4 5 6>`, for example, it will produce the values `6, 15`. For this, it uses two local variables: an accumulator `s` and a state variable `st`. The latter indicates whether the actor is actually processing a list or waiting for a new one to start. In the first state, the accumulator keeps track of the running sum. The first rule can be read as follows: When waiting for a list (`st=S0`) and reading the start of a new one (`a='<`), then reset accumulator (`s:=0`) and start processing (`st: =S1`). The second rule says: When processing (`st=S1`) and reading a data value (`a='v`), then update accumulator (`s:=s+v`). The last rule is fired at the end of the list (`a='>`); the final value of the accumulator is written on output `c`.

## 4.3 Describing Networks

A conspicuous feature of CAPH, compared to existing similarly based systems—such as those described in Sect. 10 in particular—lies in the formalism used to describe the way individual actors are instantiated and wired to form networks. This is done in an *implicit* manner, using a set of *functional equations*. In fact, CAPH embeds a small, purely functional language for describing data flow graphs called functional graph notation (FGN). The syntax and semantics of this language have been described in detail in [16], so this section is a minimal description of its possibilities.

The basic idea is that the network of actors is actually a DFG and that a DFG can be described by means of purely functional expressions. For example, the network depicted in Fig. 2—in which `i` denotes an input stream and `o` an output stream—can be described with the following equations:

```
net (x,y) = dup i
net o = mul (inc x, dec y)
```

where `f x` denotes application of function `f` to argument `x`, `(x,y)` denotes a pair of values and the `net` keyword serves to introduce bindings.

Compared to other textual or graphical network languages, this notation offers a significantly higher level of abstraction. In particular it saves the programmer from having to explicitly describe the wiring of channels between actors, a tedious and error-prone task. Moreover, ill-formed networks and inconsistent use of actors can be readily detected using a classical Hindley-Milner polymorphic type-checking phase.

Another advantage of "encoding" dataflow networks in a functional language is the ability to define reusable, polymorphic *graph patterns* in the form of higher-order functions, which offer an easy and safe compositional approach for building larger applications from smaller ones. For example, the network of Fig. 2 could also have been described with the following declarations, in which the `diamond` function *encapsulates* the diamond-shaped graph pattern exemplified here:

```
net diamond (left,top,bottom,right) x =
  let (x1,x2) = left x in
  right (top x1, bottom x2);

net o = diamond (dup,inc,dec,mul) i;
```

The `diamond` function is called a "wiring function" in the CAPH network language. From a functional perspective, this is a *higher-order* function, i.e., a function taking other function(s) as argument(s). Once defined, such a function can be reused to freely instantiate graph patterns. For example, the network depicted in Fig. 7, in which the "diamond" pattern is instantiated at two different hierarchical levels, can be simply described with the following declaration:

```
net o = diamond (dup, inc, diamond (dup,inc,dec,mul), mul) i;
```

## 4.4 Programs

A CAPH program will in general comprise at least three sections (see Sect. 9 for a complete example): one section containing the definition of the actors, another one describing the network description and a last defining the input and output streams. An optional section (not discussed further here) is available to define global constants or functions.

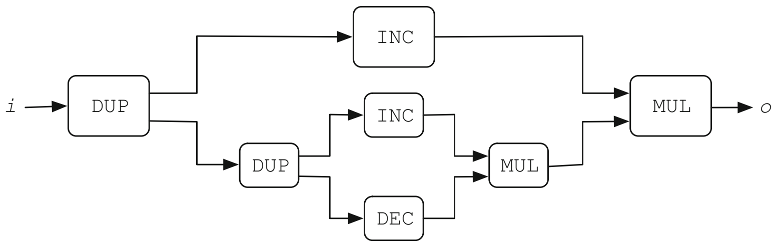


**Fig. 7** A hierarchical network

## 5 The CAPH Toolset

The current tool chain supporting the CAPH language is sketched on Fig. 8. It comprises a graph visualizer, a reference interpreter and compiler producing both SystemC and synthetizable VHDL code.[5]

The **graph visualizer** produces representations of the actor network in the `.dot` format for visualization with the GRAPHVIZ suite of tools [4]. An example is given Fig. 14.

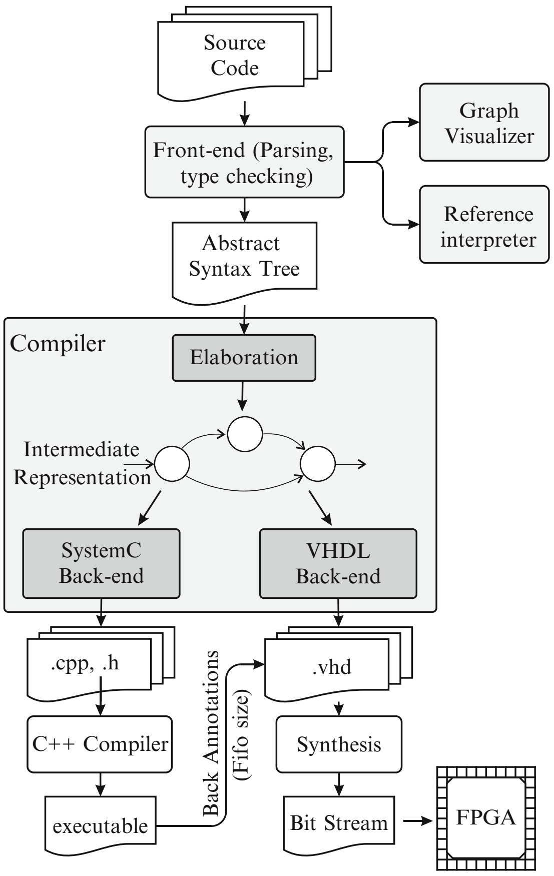
The **reference interpreter** is based on the fully formalized semantics of the language [15], written in axiomatic style. Its role is to provide reference results to check the correctness of the generated SystemC and VHDL code. It can also be used to test and debug programs, during the first steps of application development (in this case, input/output streams are read from/written to files). Several tracing and monitoring facilities are provided. For example, it is possible to compute statistics on channel occupation or on rule activation.

The **compiler** is the core of the system. It relies on an *elaboration phase*, turning the AST into a target-independent intermediate representation, and a set of dedicated back-ends. The intermediate representation (IR) is basically a process network in which each process is represented as a finite-state machine (FSM) and channels as unbounded FIFOs. Two back ends are provided: the first produces cycle-accurate SystemC code for simulation and profiling, the second VHDL code for hardware synthesis. Execution of the SystemC code provides informations which are used to refine the VHDL implementation (e.g., the actual size of the FIFOs used to implement channels).

The graph visualizer, the reference interpreter and the compiler all operate on an abstract syntax tree (AST) produced by the **front-end** after parsing and type checking. The type system is fully polymorphic. Built-in types include signed and unsigned sized integers, enumerated types and mono- and bidimensional arrays.

[5]Synthesis of the generated VHDL code is carried using third-party tools; we currently use the ALTERA *Quartus II* environment.

**Fig. 8** The CAPH toolchain



## 6 Elaboration

Elaboration generates a language and target-independent representation of the program from the high-level dataflow description. This involves two steps: first generating a *structural* representation of the actor network and then generating a *behavioral* description of each actor involved in the network.

### *6.1 Network Generation*

Generating the structural representation of the actor network involves instantiating each actor—viewed as a black box at this level—and "wiring" the resulting instances according to the dependencies expressed by the functional definitions. The CAPH compiler uses a technique known as *abstract interpretation*, described

```
actor b
  in (...)
  out (...)
match
  pats_1 -> exps_1
|  ...
| pats_i -> exps_i
| ...
| pats_n -> exps_n
;
```

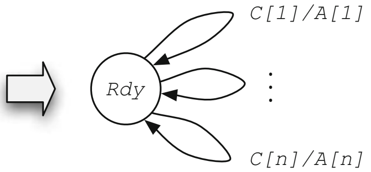


**Fig. 9** Translation of a box into a FSM

in [16], to perform this. The basic idea is that the definitions of the program are "evaluated" and each evaluation of a function bound to an actor creates an instance of this actor in the graph.[6] Each wire is then instantiated as a FIFO channel.

## 6.2 Behavioral Description

Generating the behavioral description of an instantiated actor (box) essentially consists in turning the set of pattern-matching rules of the corresponding actor into a finite-state machine with operations (FSMD level). This process is depicted in Fig. 9.

At each rule $r_i$, consisting of a list of patterns $pats_i$ and a list of expressions $exps_i$, we associate a set of *conditions* $C[\![r_i]\!]$ and a set of actions $A[\![r_i]\!]$. The set $C[\![r_i]\!]$ denotes the firing conditions for rule $r_i$, i.e., the conditions on the involved inputs, outputs and local variables that must be verified for the corresponding rule to be selected. The set $A[\![r_i]\!]$ denotes the firing actions for rule $r_i$, i.e., the read operations and write operations that must be performed on the involved inputs, outputs and variables when the corresponding rule is selected.

There are three possible firing conditions:

- $Avail_r(i)$, meaning that input $i$ is ready for reading (the connected FIFO is not empty)
- $Match_i(i, pat)$ (resp.$Match_v(v, pat)$), meaning that input $i$ (resp. variable $v$) matches pattern $pat$
- $Avail_w(o)$, meaning that output $o$ is ready for writing (the connected FIFO is not full)

and four possible firing actions:

- $Read(i)$, meaning "read input $i$ (pop the corresponding from the connected FIFO)", ignoring the read value

[6]In the sequel, an instantiated actor will be called a *box*.

**Table 1** Rules for computing the $C$ and $A$ sets for actor rules

| | |
|---|---|
| $C\llbracket pat_1,\dots,pat_m \rightarrow exp_1,\dots,exp_n\rrbracket = C_r\llbracket pat_1,\dots,pat_m\rrbracket \cup C_w\llbracket exp_1,\dots,exp_n\rrbracket$ | |
| $A\llbracket pat_1,\dots,pat_m \rightarrow exp_1,\dots,exp_n\rrbracket = A_r\llbracket pat_1,\dots,pat_m\rrbracket \cup A_w\llbracket exp_1,\dots,exp_n\rrbracket$ | |
| $C_r\llbracket pat_1,\dots,pat_m\rrbracket = \bigcup_{j=1}^{m} C'_r\llbracket pat_j,\rho_l(j)\rrbracket$ | $C_w\llbracket exp_1,\dots,exp_n\rrbracket = \bigcup_{j=1}^{n} C'_w\llbracket exp_j,\rho_r(j)\rrbracket$ |
| $C'_r\llbracket \_,\mathsf{In}\ i\rrbracket = \emptyset$ | $C'_r\llbracket \_,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| $C'_r\llbracket pat,\mathsf{In}\ i\rrbracket = \{Avail_r(i), Match_i(i,pat)\}$ | $C'_r\llbracket pat,\mathsf{Var}\ v\rrbracket = \{Match_v(v,pat)\}$ |
| $C'_w\llbracket \_,\mathsf{Out}\ o\rrbracket = \{Avail_w(o)\}$ | |
| $C'_w\llbracket exp,\mathsf{Out}\ o\rrbracket = \{Avail_w(o)\}$ | $C'_w\llbracket exp,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| $A_r\llbracket pat_1,\dots,pat_m\rrbracket = \bigcup_{j=1}^{m} A'_r\llbracket pat_j,\rho_l(j)\rrbracket$ | $A_w\llbracket exp_1,\dots,exp_n\rrbracket = \bigcup_{j=1}^{n} A'_w\llbracket exp_j,\rho_r(j)\rrbracket$ |
| $A'_r\llbracket \_,\mathsf{In}\ i\rrbracket = \emptyset$ | $A'_r\llbracket \_,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| $A'_r\llbracket const,\mathsf{In}\ i\rrbracket = \{Read(i)\}$ | $A'_r\llbracket const,\mathsf{Var}\ v\rrbracket = \emptyset\}$ |
| | $A'_r\llbracket var\ v,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| $A'_r\llbracket pat,\mathsf{In}\ i\rrbracket = \{Bind_i(i,pat)\}$ | $A'_r\llbracket pat,\mathsf{Var}\ v\rrbracket = \{Bind_v(v,pat)\}$ |
| $A'_w\llbracket \_,\mathsf{Out}\ o\rrbracket = \emptyset$ | $A'_w\llbracket \_,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| | $A'_w\llbracket var\ v,\mathsf{Var}\ v\rrbracket = \emptyset$ |
| $A'_w\llbracket exp,\mathsf{Out}\ o\rrbracket = \{Write_o(o,exp)\}$ | $A'_w\llbracket exp,\mathsf{Var}\ v\rrbracket = \{Write_v(v,exp)\}$ |

- $Bind_i(i,pat)$, meaning "read input $i$ (pop the corresponding from the connected FIFO) and match the corresponding value against pattern *pat*", binding the variable(s) occurring in the pattern to the corresponding value(s)
- $Bind_v(v,pat)$, meaning "match variable $v$ against pattern *pat*"
- $Write_o(o,exp)$ (resp. $Write_v(v,exp)$), meaning "evaluate[7] expression *exp* and write the resulting value on output $o$" (pushing the value on the connected FIFO) or in variable $v$.

Table 1 summarizes the rules for computing the sets $C\llbracket r\rrbracket$ and $A\llbracket r\rrbracket$ from the patterns and expressions composing a box rule. In these rules

- _ denotes the "empty" pattern (resp. expression) (see Sect. 4.1 and note 4), *const* a constant pattern and *var* a variable pattern.
- $\rho_l$ (resp. $\rho_r$) is a "qualifying" function: it returns $\mathsf{In}\ i$ or $\mathsf{Var}\ v$ (resp. $\mathsf{Out}\ o$ or $\mathsf{Var}\ v$) depending on whether its argument $i$ (resp. $o$) is an input (resp. output) or a variable.[8]

The intermediate representation for the `suml` actor introduced in Sect. 4.2 is given in Fig. 10. The small number appearing beside each transition is the index of the corresponding rule.

Since we are targeting a RT-level description, all transitions will be triggered by a global `clock` signal. This means that all boxes will actually change state simultaneously. The scheduling algorithm can then be written as follows:

[7]This evaluation takes place in an environment augmented with the bindings resulting from the corresponding firing action; for the sake of readability, environments have been left implicit here.

[8]It operates by inspecting the general rule format of the actor.

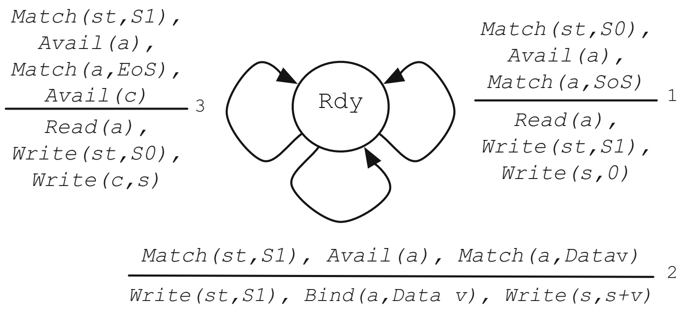

Match(st,S1), Avail(a), Match(a,EoS), Avail(c)
Read(a), Write(st,S0), Write(c,s)
3
Rdy
Match(st,S0), Avail(a), Match(a,SoS)
Read(a), Write(st,S1), Write(s,0)
1
Match(st,S1), Avail(a), Match(a,Datav)
Write(st,S1), Bind(a,Data v), Write(s,s+v)
2

**Fig. 10** Translation of the `suml` actor of Fig. 6 into a FSM

```
At each clock cycle
  For each box b, in parallel, do
    if a fireable rule r can be found in b.rules then
      read inputs for rule r;
      bind variables appearing in pattern matching
      compute expressions appearing in the rule right-hand side
      write outputs and variables
    end if
   end for
```

## 7 Translation to VHDL

The transcription in VHDL of the network derived in Sect. 6.1 boils down to instantiating the boxes forming this network and the FIFOs implementing the connexions between these boxes and wiring them together. The width (in bits) of each FIFO is deduced from the type of the conveyed data. Assigning a depth (in places) to each FIFO is a more challenging issue. A pragmatic approach consists in estimating this value using the code generated by the SystemC back end. For this, an instrumented version of the SystemC code is run, which monitors the run-time occupation of each FIFO and dumps a summary of the corresponding values. This approach is symbolized by the small arrow labeled “back annotations” in Fig. 8. If no back-annotation data is available, a default value[9] can be used.

The complete CAPH program is turned into a VHDL component. The inputs and outputs of this component correspond to the I/O streams declared in this program. This makes it possible to automatically generate a *test bench* for the resulting design, in which the original input (resp. output) data streams are provided (resp. displayed) by specific VHDL processes.

Converting the intermediate representation of boxes into VHDL requires a transformation of the corresponding FSM. First, read (resp. write) operations on

[9]Which is part of the compiler options.

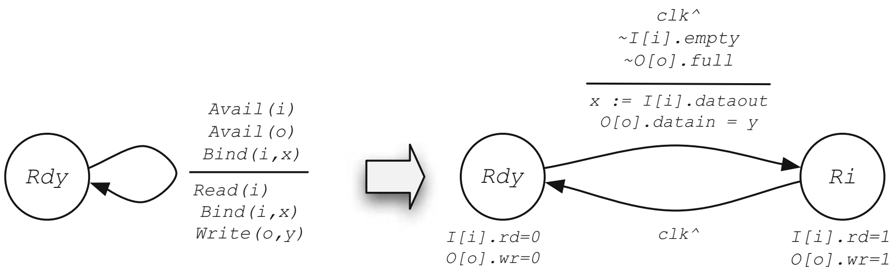


**Fig. 11** Transformation of the FSM to generate the `rd` and `wr` signals

inputs (resp. outputs) are converted into signals controlling the FIFOs connected to these inputs/outputs. There are three signals for each input: `dataout`, `empty` and `rd`, and three signals for each output: `datain`, `full` and `wr`. The signals `empty` (resp. `full`) tell whether the FIFO is empty (resp. full), i.e., ready for reading (resp. writing). The *Avail* condition on an input (resp. output) is reflected directly into the value of the `full` (resp. `empty`) signal connected to this input (resp. output). The `rd` (resp. `wr`) signals trigger the read (resp. write) operation, i.e., actually pops (resp. pushes) the data from (resp. into) the FIFO. But, because asserting these signals is done synchronously, an extra state must be added for each rule. This transformation is illustrated in Fig. 11 on a simple, single-rule, example. Here, `clk^` denotes the occurrence of the synchronizing clock signal, logical negation is denoted with the ˜ prefix operator and `I[i]` (resp. `O[o]`) denotes the FIFO connected to input `i` (resp. output `o`).

The conversion of the expressions appearing in the right-hand side of the rules is handled using a very simple syntax-directed mechanism.

The resulting code for the `suml` actor introduced in Sect. 4.2 and whose intermediate representation has been given in Fig. 10 is given in Appendix 1. In this code, the functions `is_sos`, `is_eos`, `is_data` and `data_from` are part of a package providing operations on structured values. They respectively tell whether their argument is a control value—`'<` (Start of Structure) or `'>` (End of Structure)—or a data value and, in the latter case, extract this value. Physically, the distinction between control and data values is encoded with two extra bits.[10]

## 8 Translation to SystemC

Transcription of the intermediate representation in SystemC basically follows the same principles and techniques than those described in Sect. 7. Boxes are implemented as SystemC modules and box interconnexions as FIFO channels. As stated in the previous section, these FIFO channels can be generated with an option

[10]Hence, pixels are actually encoded on 8 bits (+2 for control) and sums on 16 bits in this example.

to monitor their run-time occupation in order to provide back annotations for the VHDL back end. Similarly to the VHDL back end, the complete CAPH program is turned into a SystemC module, whose inputs and outputs correspond to the I/O streams declared in the program, making it possible to automatically generate a *test bench* for the resulting design.

The resulting SystemC code for the `suml` actor is given in Appendix 2.

## 9 Experimental Results

We have experimented with a prototype version of the compiler using a very simple application as a test bench. The goal is to validate the overall methodology before moving to more complex algorithms and to identify key issues. The application is a simple motion detector operating in real time on a digital video stream. Moving objects are detected by spatio-temporal changes in the grey-level representation of the successive frames, and a rectangular window is drawn around them.

The algorithm involves (1) computing the difference image between two consecutive frames, (2) thresholding this difference to obtain a binary image, (3) computing the horizontal projection (row-wise sum) of this image, (4) thresholding this projection to extract horizontal bands where moving objects are likely to be found, (5) computing the vertical projection (column-wise sum) on each band, and (6) applying a peak detector to the projections to define the position of each moving object in the band (Fig. 12).

The encoding of this algorithm in CAPH appears in Fig. 13. It consists of five sections. The first section is used for defining type abbreviations. The second section defines global constants (global functions can also be defined here). In the third section, the behavior of all the actors involved in the network is defined. Here, due to space limitations (and because several examples have already been given), the text of the descriptions has been omitted. The fourth section is where network input and output streams are defined. In this particular case, the input stream is read from a camera and the output stream is written to a display.[11] Finally, the last section gives the functional equations defining the network.

The code involves eight different actors. The `asub` actor computes the absolute value of the difference between two frames. The `d1f` is a frame delay operator. The `thr` actor has been described in Sect. 4.2. The `hproj` actor computes the horizontal projection of an image. The `vwin` actor extracts a horizontal band from an image according to the profile of the thresholded horizontal projection. The `vproj` computes the vertical projection of an image (the band is represented exactly as an image). The `peaks` actor analyzes the vertical projection and computes a list

[11]Dedicated VHDL processes, transparent to the programmer, handle the insertion (resp. removal) of control tokens after (resp. before) the image date is read from (resp. written to) camera (resp. display).

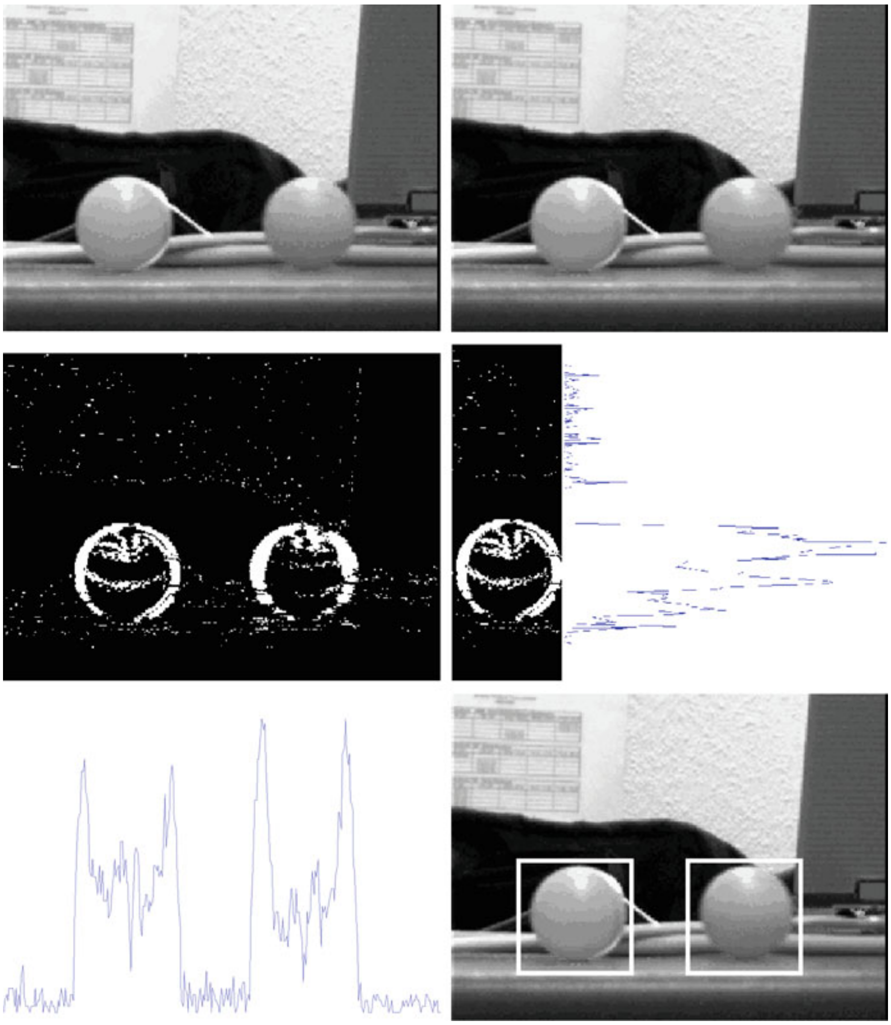


**Fig. 12** Motion detection algorithm. *Top*: two consecutive frames of a sequence. *Middle* and *bottom left*: thresholded difference image. *Middle right*: horizontal projection. *Bottom left*: vertical projection of the detected band. *Bottom right*: final result

of pairs, each pair giving the position of two consecutive peaks. Finally, the `win` actor uses the positions computed by the `vwin` and `peaks` actors to display a frame around the detected objects.

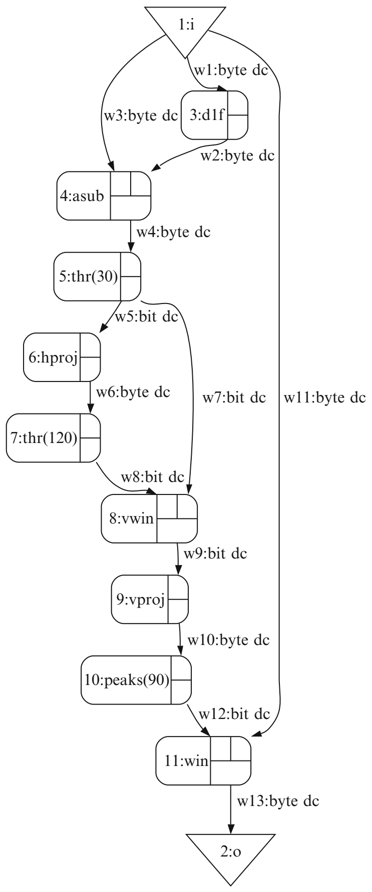
The corresponding DFG, obtained with the graph visualizer, is depicted in Fig. 14 (square boxes represent actors, triangles input and output and edges are labeled with the type of corresponding channel).

**Fig. 13** Source code for the motion detection application (excerpt)

```
-- Type declarations
type byte = unsigned<8>
type bit = unsigned<1>

-- Constants (thresholds)
const k1 = 30    -- for binary image
const k2 = 1200  -- for hor. projection
const k3 = 900   -- for vert. projection

-- Actor declarations
actor asub ()
  in (a:byte dc, b:byte dc)
 out (c: byte dc)
 ...
actor d1f () ...
actor thr (t:byte) ...
actor hproj () ....
actor vwin () ...
actor vproj () ...
actor peaks (t:byte) ...
actor win () ...

-- I/O streams
stream i : byte dc from "camera:0"
stream o : byte dc to "display:0"

-- Network definitions
net diff_im = asub (i, d1f i)
net bin_im = thr k1 diff_im
net hp = thr k2 (hproj bin_im)
net hband = vwin (hp, bin_im)
net vp = vproj hband
net o = win (peaks k3 vp, i)
```

**VHDL Implementation.** Our current target platform is a smart camera integrating an FPGA board, an image-sensing device and communication board. It is fully described in [1]. The FPGA board consists of one FPGA (a Stratix EP1S60), five 1MB SRAM banks, and one 64MB SDRAM block. Two dedicated VHDL processes provide interfacing of the generated actor network to the physical I/O devices (camera and display). The VHDL code produced by the CAPH compiler is compiled and downloaded to the FPGA using the Altera Quartus toolset. Small FIFOs are implemented using logic elements. Midsized ones (e.g., for line delays) use the on-chip SRAM memory banks of the FPGA. Frame delay FIFOs (such as the one required by the `d1f` operator) use the SDRAM. As stated in Sect. 7, the size of each FIFO is currently estimated using the SystemC back end, by running an instrumented version of the generated code in which the occupation of the FIFOs is monitored at run-time. In this particular example, four-place FIFOs are sufficient on all channels except on wire `W7` (see Fig. 14), where a FIFO with a depth of at least

**Fig. 14** Dataflow graph for the motion detection application



one line must be inserted (because the value of the horizontal projection for a given line is only available at the end of this line), and on wire `W11`, where a FIFO with a depth of one frame must be inserted (because the positions of the bounding frames for one frame are only known at the end of this frame).[12]

Table 2 reports the number of lines of code (LOC) both for the CAPH source code and the generated VHDL. The tenfold factor observed between the volume of the CAPH source code and the VHDL code (automatically) generated by the compiler gives an idea of the gain in productivity offered by our approach.

After synthesis, the whole application uses 3,550 logic elements (6 %), 17 kbits of SRAM and 512 kB of SDRAM. It operates on the fly on video streams of $512 \times 512 \times 8$ bit images at 15 FPS, with a clock frequency of 150 MHz.

[12]It is possible to get rid of this FIFO by inserting a `s1f` (skip one frame) operator on the corresponding wire. In this case, the bounding boxes computed on frame number $i$ are actually displayed on frame $i+1$, which is acceptable if the objects do not move too quickly.

**Table 2** Line of code (LOC) for the motion detection application

| Actor | CAPH | Generated VHDL |
|---|---|---|
| asub | 7 | 78 |
| d1f | 15 | 124 |
| thr | 7 | 68 |
| hproj | 11 | 91 |
| vproj | 18 | 136 |
| peaks | 17 | 134 |
| win | 17 | 134 |
| vwin | 20 | 190 |
| Network, decls | 13 | 285 |
| Total | 125 | 1240 |

## 10 Related Work

The approach followed in CAPH bears similarities with that adopted by other stream-processing and/or actor-based languages such as CAL [12] and Canals [2]. All of them share the idea of a network of computational units exchanging data through unidirectional channels following the basic dataflow model of computation. Computational units are called *actors* in CAL and *kernels* in Canals. The differences with CAPH mainly come from the scheduling of execution in actors on the one hand and the syntax and semantics of the network language on the other hand.

Both CAL and Canals allow complex execution scheduling to be specified for actors. For example, CAL provides constructs for expressing guards, priorities or even finite-state machine-based scheduling within each actor. Canals comes with a sub-language to define scheduling of kernels within the network. By contrast, scheduling is kept much simpler in CAPH since it is entirely specified by the pattern-matching rule-based mechanism. This has been made possible by allowing control tokens (namely, the '<' and '>' tokens) within the streams exchanged between actors. This approach in turn greatly simplifies the generation of HDL code, which basically boils downs to finite-state machines, easily encoded in VHDL.

Both CAL and Canals use a dedicated *network language* (NL) to describe the network of actors. The abstraction level of these languages is low: The programmer must manually "wire" the network by explicitly listing all the actors and their connexions, a tedious and error-prone task. The network language embedded in CAPH allows implicit description of network by means of functional expression and naturally supports higher-order constructs.

The concept of match-based transitions used in CAPH for describing actor behavior has been borrowed from the Hume [7] language, in which it is used to describe the behavior of asynchronous *boxes* connected by wires. But the semantics of Hume and CAPH are different. In Hume, *boxes* are stateless, wires provide memorization for a single token, and the execution model is based on the concept of global cycles.

From a more historical perspective, it can be noted that the idea of describing an application as a graph of dataflow operators and then physically mapping this graph on a network on data-driven processing elements (DDPs) is definitely not new. It has been exploited, for example, in [10, 17–19]. But in these projects, the architecture of both the DDPs and the network was fixed.[13] Moreover, the ASIC technology in the 1990s did not allow the integration of large networks of complex elements.[14] The current FPGA technology brings a truly new dimension to this old idea, by allowing very large networks to be embedded in only one chip of a COTS board. This is possible because each actor is now implemented using the exact amount of required resources instead of consuming a pre-implemented DDP.

## 11 Conclusion

We have introduced CAPH, a DSL for programming stream-processing applications on FPGAs. The presentation adopted here is deliberately informal. Its goal is to give an idea of the motivations, basic principles and capabilities of the language. Moreover, many features of the languages have not been presented. These include, for example, the tracing and debug facilities offered by the interpreter, the foreign functions interfacing mechanism for the VHDL and SystemC back ends, the support for 1D and 2D arrays at the actor level, *etc.* More detailed informations, including the full syntax and formal semantics and examples, can be obtained from the CAPH web site [15].

Preliminary examples, such as the one described in this chapter, show that efficient implementations of reasonably complex applications can be obtained with a language whose abstraction level is significantly higher than that of traditional HDL languages such as VHDL or Verilog.

Work is currently undergoing in three directions.

First is assessing our tools on larger and more complex applications – such as H264 video decoding, for example – for which VHDL implementations, either hand-crafted or obtained with some of the tools cited in Sect. 10, are available for comparison.

Second is improving the compiler and optimizing the generated VHDL code. A limitation, deriving from the current elaboration process, is that the expressions on the right-hand side of the rules are evaluated in one clock cycle. This is not a problem for "simple" actors such as the ones used in the application described here, but we anticipate that for actors involving more complex computations, this approach could result in unacceptable critical paths. In this case, the programmer would have to

[13]Programming these "dataflow computers" then meant writing the code of each DDP and configuring the network interconnections.

[14]The dataflow computer described in [17] embedded 1024 DDPs, but it was a dedicated machine, not easily replicated.

"break" complex actors into small-enough actors to reach a given clock frequency. Because CAPH is fully formalized, we think that it should be possible to develop some kind of "actor calculus"—in the vein of the "box calculus" introduced by Grov and Michaelson in [5] for the Hume language—to assist the programmer in carrying out this transformation.

Third is to apply static analysis techniques to actor behaviors in order to statically estimate the size of FIFO channels. As stated in Sect. 9, these sizes are currently estimated by running an instrumented version of the code generated by the SystemC back end (which monitors FIFO usage at run-time). An exact prediction of these sizes at compile time is only possible in the context of a pure *synchronous dataflow* model of computation [11], where the rate at which tokens are produced by actors does not depend on the *value* of these tokens, which is not the case for CAPH in general. Nevertheless, we think that in most of cases, it should be possible to statically compute an upper bound for the FIFO sizes, by using some results described in [13] in the context of synchronous languages, for example.

## Appendix 1: Code Generated by the VHDL Back End for the `suml` Actor

```
entity suml_act is
   port (
    a_empty: in std_logic;
    a: in std_logic_vector(9 downto 0);
    a_rd: out std_logic;
    c_full: in std_logic;
    c: out std_logic_vector(15 downto 0);
    c_wr: out std_logic;
    clock: in std_logic;
    reset: in std_logic
    );
end suml_act;

architecture FSM of suml_act is
    type t_state is (R0,R1,R2,R3);
    type t_enum1 is (S0,S1);
    signal state: t_state;
    signal s : std_logic_vector(15 downto 0);
    signal st : t_enum1;
begin
  process(clock, reset)
    variable p_v : std_logic_vector(7 downto 0);
    variable s_v : std_logic_vector(15 downto 0);
  begin
    if (reset='0') then
      state <= R0;
      st <= S0;
      a_rd <= '0';
      c_wr <= '0';
    elsif rising_edge(clock) then
      case state is
        when R0 =>
          if a_empty='0' and is_sos(a) and st=S0 then
            a_rd <= '1';
            st <= S1;
            s <= "0000000000000000";
            state <= R1;
          elsif a_empty='0' and is_data(a) and st=S1 then
            s_v := s;
            p_v := data_from(a);
            a_rd <= '1';
            st <= S1;
            s <= s_v + p_v;
            state <= R2;
          elsif a_empty='0' and is_eos(a) and st=S1 and
            c_full='0' then
            s_v := s;
            a_rd <= '1';
            st <= S0;
            c <= s_v;
            c_wr <= '1';
            state <= R3;
          end if;
        when R1 =>
            a_rd <= '0';
            state <= R0;
        when R2 =>
            a_rd <= '0';
            state <= R0;
        when R3 =>
            a_rd <= '0';
            c_wr <= '0';
            state <= R0;
      end case;
    end if;
  end process;
end FSM;
```

## Appendix 2: Code Generated by the SystemC Back End for the `suml` Actor

```
#include "dc.h"
#include <systemc.h>
#include "fifo.h"

SC_MODULE(suml_act) {
  sc_in<bool> clk;
  sc_port<fifo_in_if<DC<sc_int<8>  > > > a;
  sc_port<fifo_out_if<sc_uint<16> > > c;
  typedef enum {S0,S1} _enum1;
  void main(void);
  SC_HAS_PROCESS(suml_act);
  suml_act(sc_module_name name_, bool trace_=false  ) :
    modname(name_), sc_module(name_), trace(trace_)
    { SC_THREAD(main);
      sensitive << clk.pos(); }
  ~suml_act() { }
  private:
    bool trace;
    sc_module_name modname;
    sc_uint<8> s;
    _enum1 st;
    sc_int<8>  p_v;
    sc_uint<16>  s_v;
};

void suml_act::main(void) {
    st = S0;
    while ( 1 ) {
      wait(); // clk
      if ( a->rd_rdy() && a->peek().is_sos() && st==S0 ) {
        a->read();
        st = S1;
        s = 0;
        }
      else if ( a->rd_rdy() && a->peek().is_data() && st==S1 ) {
        s_v = s;
        p_v = a->read().data();
        st = S1;
        s = s_v+p_v;
        }
      else if ( a->rd_rdy() && a->peek().is_eos() && st==S1 &&
                c->wr_rdy() ) {
        s_v = s;
        a->read();
        st = S0;
        c->write(s_v);
        }
    }
}
```

## References

1. Chalimbaud P, Berry F (2007) Embedded active vision system based on an FPGA architecture. EURASIP J Embedded Syst 2007:26–26. URL http://dx.doi.org/10.1155/2007/35010
2. Dahlin A, Ersfolk J, Yang G, Habli H, Lilius J (2009) The Canals language and its compiler. In: Proceedings of th 12th international workshop on software and compilers for embedded systems, SCOPES '09, pp 43–52. ACM, New York, NY, USA. URL http://dl.acm.org/citation.cfm?id=1543820.1543829

3. Frigo J, Gokhale M, Lavenier D (2001) Evaluation of the Streams-C C-to-FPGA compiler: an applications perspective. In: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays, FPGA '01, pp 134–140. ACM, New York, NY, USA. URL http://doi.acm.org/10.1145/360276.360326
4. Graph visualisation software. URL http://www.graphviz.org
5. Grov G, Michaelson G (2010) Hume box calculus: robust system development through software transformation. High Order Symbol Comput 23:191–226. URL http://dx.doi.org/10.1007/s10990-011-9067-y
6. Gupta S, Dutt N, Gupta R, Nicolau A (2003) Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In: In international conference on VLSI design, pp 461–466
7. Hammond K, Michaelson G (2003) Hume: a domain-specific language for real-time embedded systems. In: Proceedings of the 2nd international conference on Generative programming and component engineering, GPCE '03, pp 37–56. Springer, New York, Inc., New York, NY, USA. URL http://dl.acm.org/citation.cfm?id=954186.954189
8. Handel-c language reference manual (2009) URL http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf
9. Impulse accelerated technologies. URL http://www.impulsec.com
10. Koren I, Mendelsom B, Peled I, Silberman GM (1988) A data-driven vlsi array for arbitrary algorithms. Computer 21:30–43. DOI 10.1109/2.7055. URL http://dl.acm.org/citation.cfm?id=50810.50813
11. Lee E, Messerschmitt D (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
12. Lucarz C, Mattavelli M, Wipliez M, Roquier G, Raulet M, Janneck J, Miller I, Parlour D (2008) Dataflow/Actor-Oriented language for the design of complex signal processing systems. In: Proceedings of the 2008 conference on design and architectures for signal and image processing, DASIP 2008, pp 168–175
13. Mandel L, Plateau F, Pouzet M (2010) Lucy-n: a n-synchronous extension of Lustre. In: Tenth International conference on mathematics of program construction (MPC 2010). Québec, Canada. URL MandelPlateauPouzet-MPC-2010.pdf
14. Najjar WA, Boehm W, Draper BA, Hammes J, Rinker R, Beveridge JR, Chawathe M, Ross C (2003) High-level language abstraction for reconfigurable computing. Computer 36:63–69. DOI http://doi.ieeecomputersociety.org/10.1109/MC.2003.1220583
15. Sérot J Caph language reference manual. URL http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/caph.html
16. Sérot J (2008) The semantics of a purely functional graph notation system. In: Trends in functional programming. Madrid, Spain. URL http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/fgn.html
17. Sérot J, Quénot GM, Zavidovique B (1993) Functional programming on a data-flow architecture: Applications in real time image processing. Int J Mach Vision Appl 7(1):44–56
18. Sérot J, Quénot GM, Zavidovique B (1995) A visual dataflow programming environment for a real-time parallel vision machine. J Vis Lang Comput 6:327–347
19. Vasell J, Vasell J (1992) The function processor: a data-driven processor array for irregular computations. Future Gener Comput Syst **8**, 321–335. DOI 10.1016/0167-739X(92)90066-K. URL http://dl.acm.org/citation.cfm?id=140466.140484
20. Yankova YD, Bertels K, Vassiliadis S, Kuzmanov G, Chaves R (2006) HLL-to-HDL generation: Results and challenges. In: Proceeding of ProRisc 2006
21. Yankova YD, Kuzmanov G, Bertels K, Gaydadjiev GN, Lu Y, Vassiliadis S (2007) Dwarv: Delftworkbench automated reconfigurable vhdl generator. In: Proceedings of the 17th International conference on field programmable logic and applications (FPL07), pp 697–701

# Compact CLEFIA Implementation on FPGAs

**Ricardo Chaves**

## 1 Introduction

Cryptography is a key service in the current digital communication world, with the digital data being constantly transmitted through public open channels, whether it is an internet network access or through the air, such as in wireless and mobile phone networks. In order to have confidentiality and access management to that same data, ciphering mechanisms need to be employed when sending sensitive information through these public media. Ciphering algorithms have been in use for a long time, but the growing processing capabilities of digital equipment and the growing bandwidth for digital communication channels impose the need for more dedicated and secure algorithms. These algorithms can be divided in two classes, asymmetric and symmetric. While the first ones are based on complex mathematical problems, thus having long processing times, the second ones are implemented using operations, such as byte substitution, bit permutation, and basic arithmetic operations, and can process large amounts of data in small amounts of time.

One of such algorithms is the CLEFIA encryption algorithm, a novel symmetrical block ciphering algorithm proposed and developed by SONY Corporation focused on digital rights management (DRM) purposes [8]. This algorithm improves the security of encryption with the use of techniques such as diffusion switch mechanisms, consisting of multiple diffusion matrices in a predetermined order, to ensure immunity against differential and linear attacks [1, 7, 12], and the use of whitening keys, combining data with portions of the key before the first round and after the last round. In this chapter, FPGAs were selected as the target technology given their advantages in terms of computation adaptability, time to market, development costs, and deployment time for dedicated solutions [2, 5].

R. Chaves (✉)
INESC-ID, IST-TULisbon, Rua Alves Redol 9, 1000-029 Lisboa, Portugal
e-mail: ricardo.chaves@inesc-id.pt

Two structures for the computation of the CLEFIA symmetrical encryption algorithm are presented in this chapter. These structures use the FPGA's embedded memories (BRAMs) allowing for a more compact and high-throughput hardware implementation. The first structure computes one CLEFIA round per clock cycle and is based on the topology presented in [11] for an ASIC technology and adapted in this chapter to FPGA technologies. The second structure, herein presented, further optimizes the area resources by exploring the symmetries of the round computation in this algorithm. This second structure allows to obtain a more compact topology by reusing hardware components, while achieving similar throughputs due to the addition of a pipeline stage. Both the presented structures allow for the computation of the CLEFIA algorithm with all the key sizes defined in the standard [8]. The related CLEFIA state of the art on FPGAs presented in [4] is also considered. This structure performs the CLEFIA computation on a fully unrolled topology, achieving higher throughputs at the expense of area resources and lower flexibility. In order to analyse the gains and costs of performing the key scheduling with dedicated hardware, a structure is also herein presented for the expansion of 128-bit input keys.

While few papers proposing the CLEFIA implementation have been published, and mainly for ASIC technologies, the presented structures are compared with the existing related art. The present analysis suggests improvements in the throughput per slice efficiency metric of 1.75–2.25 times on several FPGA technologies. Hardware resource reductions up to 60 %, at the expense of a throughput reduction of 15 % on a Virtex 4 FPGA, are suggested by the experimental results. Considering the fully unrolled structure proposed in [4], area gains of 40 times can be achieved at a cost of a throughput reduction of 20 times. The structures herein presented are able to achieve throughputs above 1 Gbit/s with a low FPGA resource occupation. Experimental results also suggest that the key scheduling does not significantly affect the ciphering performance; however, it increases the needed area resources by up to 100 %.

This chapter is organized as follows. Section 2 presents a brief description of the CLEFIA algorithm. Section 3 describes the proposed structures for the ciphering and key scheduling computation. Evaluation of the obtained implementation results and comparison with the related state of the art are presented in Sect. 4. Concluding remarks are presented in Sect. 5.

## 2 CLEFIA Encryption Algorithm

The CLEFIA algorithm is a 128 bit block symmetrical ciphering algorithm with a key size of 128, 192, or 256 bits. As in most current block ciphers, it consists of a key scheduling phase and an input data block transformation phase computed over multiple rounds, employing a relatively homogeneous algorithm. This regularity facilitates the development of compact structures, allowing it to be easily deployed in platforms with limited resources [6].

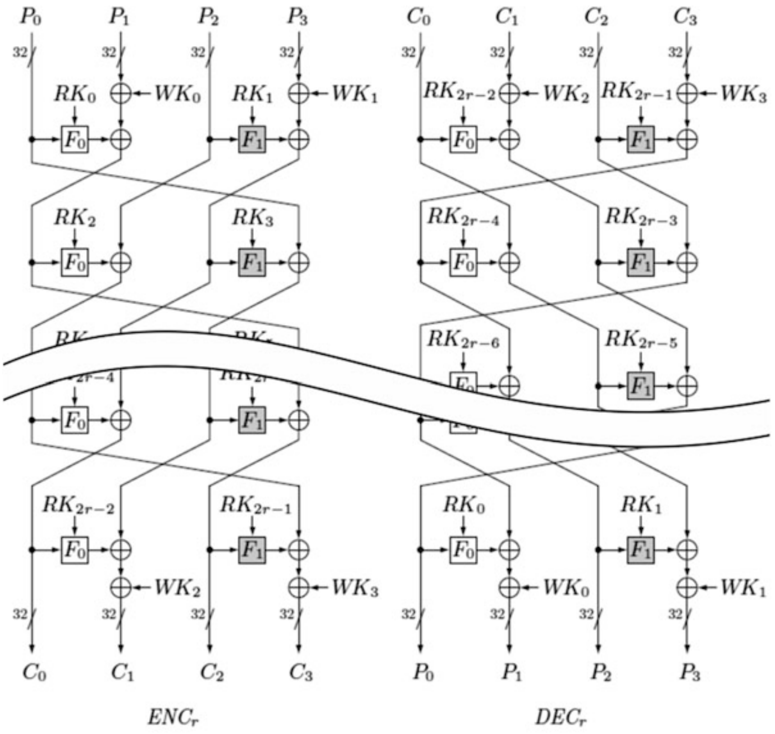


**Fig. 1** CLEFIA datapath

State-of-the-art design techniques, present in recent ciphering algorithms, are also found in the CLEFIA algorithm, namely, (1) whitening keys, a technique used to improve security of iterated block ciphers, consisting in steps to combine data with portions of the key, before the first round and after the last round; (2) Feistel structures, which are the most widely used and the best studied structures for the design of block ciphers, initially proposed by H. Feistel in the early 1970s and adopted by the well-known block cipher DES; and (3) a diffusion switch mechanism, consisting in the usage of multiple diffusion matrices organized in a predetermined order, to ensure immunity against differential and linear attacks [7,9].

The data path of CLEFIA is composed of a four-branch Feistel structure computed for several rounds, defined as $\mathrm{GFN}_{4,n}$. This Feistel structure is an extended version of the traditional two-branch Feistel structure, which uses two different F-functions per round. Each F-function has a 32 bit input/output data path, as depicted in Fig. 1. F-functions $F_0$ and $F_1$ have different diffusion matrices, providing CLEFIA with a diffusion switch mechanism. Additional robustness was added to this algorithm with the addition of four whitening keys (WK), two added before the main computation round and the other two added at the end of the round operations. The different input key sizes that can be used in CLEFIA (128, 192, or 256 bits) directly influence the number of computed rounds, 18, 22, or 26, respectively [8].

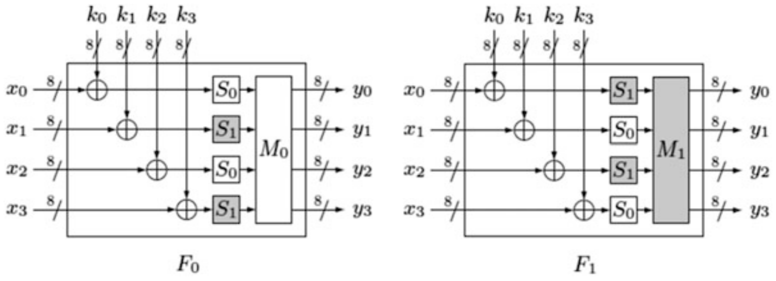


**Fig. 2** CLEFIA: F-functions

As in most ciphering algorithms, operations on data consist of byte swapping, byte substitution, and arithmetic operations. The following describes the main operations performed in the CLEFIA algorithm.

## *2.1 F-Functions*

Two different F-functions ($F_0$ and $F_1$) are employed in each round and used for data confusion. These F-functions consist of additions over GF($2^8$) between the round data and the round keys (RK); substitution boxes ($S_0$ and $S_1$); and diffusion matrices ($M_0$ and $M_1$), one for each F-function, as depicted in Fig. 2.

CLEFIA employs two different types of 8-bit S-Boxes, $S_0$ and $S_1$. $S_0$ is based on four 4-bit S-Boxes, and $S_1$ is based on the inverse function over GF($2^8$) [8]. $S_0$ is generated by combining four 4-bit S-Boxes obtained with operations over GF($2^4$). $S_1$ is obtained as the inverse function performed over GF($2^8$) defined by the primitive polynomial $z^8 + z^4 + z^3 + z^2 + 1$.

Two different diffusion matrices, $M_0$ and $M_1$ defined in (1), are an integral part of the diffusion mechanism present in CLEFIA, improving the resistance of the algorithm to differential attacks. Each one of the four bytes (output of the S-Boxes) are multiplied, by the values in each line of the matrix, and added over GF($2^8$). The constant values used on these matrices suggest some simplifications of the operations needed in these diffusion matrices, as suggested in [11].

$$M_0 = \begin{pmatrix} 0\times01 & 0\times02 & 0\times04 & 0\times06 \\ 0\times02 & 0\times01 & 0\times06 & 0\times04 \\ 0\times04 & 0\times06 & 0\times01 & 0\times02 \\ 0\times06 & 0\times04 & 0\times02 & 0\times01 \end{pmatrix}, \qquad M_1 = \begin{pmatrix} 0\times01 & 0\times08 & 0\times02 & 0\times0a \\ 0\times08 & 0\times01 & 0\times0a & 0\times02 \\ 0\times02 & 0\times0a & 0\times01 & 0\times08 \\ 0\times0a & 0\times02 & 0\times08 & 0\times01 \end{pmatrix}. \quad (1)$$

**Algorithm 1** - $\text{GFN}_{4,n}(RK,X)$

```
Input : X, RK
$T_0 \mid T_1 \mid T_2 \mid T_3 \leftarrow X_0 \mid X_1 \mid X_2 \mid X_3$;
for $i = 0$ to $n-1$ do
    $T_1 = T_1 \oplus F_0(RK_{2i}, T_0)$,   $T_3 = T_3 \oplus F_1(RK_{2i+1}, T_2)$;
    $T_0 \mid T_1 \mid T_2 \mid T_3 \leftarrow T_1 \mid T_2 \mid T_3 \mid T_0$;
end for
$Y_0 \mid Y_1 \mid Y_2 \mid Y_3 \leftarrow T_0 \mid T_1 \mid T_2 \mid T_3$;
Output : Y
```

## 2.2 Data Processing

The encryption process in CLEFIA mostly consists of the $\text{GFN}_{4,n}$ Feistel network, where $n$ represents the number of rounds to be computed. In each round the data is mixed and added with the round keys using bitwise XOR additions, the mentioned F-functions, and bit permutations. The $\text{GFN}_{4,n}$ network [8] can be defined by the function $Y = \text{GFN}_{4,n}(RK,X)$, computed as depicted in Algorithm 1.

The input $X$ and the output $Y$ are 128-bit values, while RK are the $2n \times 32$-bit round keys. A full 128-bit block CLEFIA encryption also requires de addition of four 32-bit whitening keys ($\text{WK}_i$). Two 32-bit whitening keys are added before the $\text{GFN}_{4,n}$ computation, and two more are added after all the $\text{GFN}_{4,n}$ rounds are computed, as depicted in Fig. 1. Given the Feistel network-based structure of this algorithm the decryption process is identical to the encryption one, using the same computational units, only differing in the order that the operations are performed, as depicted in the rightmost side of Fig. 1. This inverse computation is achieved by feeding the round and whitening keys in the inverse order, allowing for the same computational structure to be used [8].

Two 32 bit round keys are employed in each round. These round keys are obtained from the original key, as are the whitening keys. The generation of these values is discussed in the following section.

## 2.3 Key Scheduling

As stated above, the CLEFIA algorithm supports inputs keys of 128, 192, and 256. However, the ciphering process itself requires several 32-bit rounds keys and whitening keys. This means that the input key needs to be expanded into the 36, 44, or 52 round keys, respectively, and the four whitening keys. This expansion is realized by the key scheduling part of the CLEFIA algorithm [8].

The calculation of the round keys is performed by feeding the initial key value through a processing network (GFN) as when ciphering data. The difference lays in the fact that the input key is used rather that an input data block. This GFN network can be a four-branch structure, similar to the one depicted in Fig. 1, used

**Algorithm 2** - Expansion of $K_{128}$ and L

```
Input : L, K
for $i = 0$ to 8 do
    $T = L \oplus (CON^{128}_{24+4i} \mid CON^{128}_{24+4i+1} \mid CON^{128}_{24+4i+2} \mid CON^{128}_{24+4i+3})$
    $L = \Sigma(L)$
    if $i$ is $odd$ then $T = T \oplus K$ end if
    $RK_{4i} \mid RK_{4i+1} \mid RK_{4i+2} \mid RK_{4i+3} \leftarrow T$
end for
Output : $RK$
```

for a 128-bit input key, or an eight-branch GFN, used for the 192 and 256 bit input key sizes [8]. After the GFN calculation is completed, the result is expanded using a Double Swap function (a simple bitwise permutation) and additional constants are added. The resulting values are the needed round keys, used in the ciphering data path.

**Key Scheduling for a 128-Bit Input Key:** For an input key of 128 bits the four 32-bit whitening keys are obtained directly from the input key, by

$$WK_0 \mid WK_1 \mid WK_2 \mid WK_3 \leftarrow K. \tag{2}$$

To obtain the round keys, a more complex computation is needed. This computation is divided in two steps. In the first step, a value $L$ is calculated, being a function of the 128-bit input key. In the second step, this $L$ value is further manipulated in order to obtain the several round keys. A total of $60 \times 32$-bit constant values ($\mathrm{CON}^{128}$) are used. These constant values are precomputed [10] and depend on the size of the input key (128, 192, or 256 bits).

For the computation of the 128-bit $L$ value the $\mathrm{GFN}_{4,12}$ function is used. In this case the input values are the 128 bits of the input key (instead of a 128-bit block of data to be ciphered as in the case of the ciphering process) and 24 input constants (instead of the 24 round keys), thus obtaining

$$L = \mathrm{GFN}_{4,n}(\mathrm{CON}^{128}_{0:23},\ K). \tag{3}$$

Followed by the computation of the $L$ value the resulting 36 round keys can be obtained by Algorithm 2.

In order to have the above algorithm fully specified DoubleSwap function ($\Sigma$) has to be defined. This function swaps several bits of the 128-bit input and returns another 128-bit value, as specified by [10]:

$$Y \leftarrow \Sigma(X) = X[7-63] \mid X[121-127] \mid X[0-6] \mid X[64-120]. \tag{4}$$

**Key Scheduling for a 192-Bit Input Key:** For an input key of 192 bits, the four 32-bit of the whitening keys are no longer obtained directly. The input key is transformed into $K_L$ and $K_R$ (two 128-bit values) and the bitwise XOR operation is applied, resulting in the 128 bits of the whitening keys, as detailed in (7).

$$K_L \leftarrow K_0 \mid K_1 \mid K_2 \mid K_3, \tag{5}$$

$$K_R \leftarrow K_4 \mid K_5 \mid \overline{K_0} \mid \overline{K_1}, \tag{6}$$

$$WK = K_L \oplus K_R. \tag{7}$$

The main difference in the round key expansion for a 192-bit input key lays in the generation of the $L$ value. In this case a GFN network with an eight-branch structure must be used (GFN$_{8,n}$). In this network, four 32-bit round keys are used per round. For the 192-bit input key, the GFN network is composed of ten rounds, requiring $40\times32$-bit constant values (CON$^{192}$). Note that a GFN$_{8,n}$ network receives an input block of 256 bits, which in this case corresponds to the concatenation of K$_L$ with K$_R$, and outputs another 256-bit value, as described by

$$L_L \mid L_R = \text{GFN}_{8,10}(\text{CON}^{192}_{0:40},\ K_L \mid K_R). \tag{8}$$

The second step in the calculation of the 44 round keys is the processing of the L and K values and the addition of the remaining 44 constants. This calculation can be described as

**Algorithm 3** - Expansion of K$_{192}$ and L

```
Input : $L_L \mid L_R,\ K_L \mid K_R$
for $i = 0$ to 10 do
    if $i$ mod 4 = (0 or 1) then
        $T = L_L \oplus (CON^{192}_{40+4i} \mid CON^{192}_{40+4i+1} \mid CON^{192}_{40+4i+2} \mid CON^{(192)}_{40+4i+3})$
        $L_L = \Sigma(L_L)$
        if $i$ is odd then $T = T \oplus K_R$ end if
    else
        $T = L_R \oplus (CON^{192}_{40+4i} \mid CON^{192}_{40+4i+1} \mid CON^{192}_{40+4i+2} \mid CON^{192}_{40+4i+3})$
        $L_R = \Sigma(L_R)$
        if $i$ is odd then $T = T \oplus K_L$ end if
    end if
    $RK_{4i} \mid RK_{4i+1} \mid RK_{4i+2} \mid RK_{4i+3} \leftarrow T$
end for
Output : $RK$
```

**Key Scheduling for a 256-Bit Input Key:** The key expansion process for a 256-bit input key is very similar to the one described for a 196-bit key. The differences are in the computation of the K$_L$ and K$_R$ values, the constants used (CON$^{256}$), and in the loop length of Algorithm 3.

The K$_L$ and K$_R$ values are obtained from the 256 bits of the input key as

$$K_L \leftarrow K_0 \mid K_1 \mid K_2 \mid K_3, \tag{9}$$

$$K_R \leftarrow K_4 \mid K_5 \mid K_6 \mid K_7. \tag{10}$$

The loop length of Algorithm 3 is of 13 ($i = 0$ to 12) resulting in the $52 \times 32$-bit round keys needed for the CLEFIA ciphering process with 256-bit input keys.

## 3 CLEFIA Hardware Structures

Herein, a compact hardware CLEFIA structure is presented, which is still being able to achieve competitive throughput and performance metrics, even on low-cost devices. Two hardware structures are herein presented, one being the derivation of the structure proposed in [11] for ASIC technologies and a second one that further optimizes the data path. Both structures allow for the ciphering and deciphering computations with all three key sizes specified for this algorithm.

As described above, the CLEFIA algorithm computation is divided into the key scheduling computation and the ciphering computation itself. While the ciphering computation needs to be performed for every 128-bit data block, the key scheduling computation only needs to be computed once for the same input key. This is an important factor when deciding to add or not dedicated hardware to perform the key expansion. The main flow of this chapter considers that the key scheduling computation is performed in software and that the resulting round keys are transferred to the hardware core during the initialization procedure. Besides receiving and storing the expanded keys, the hardware core is also responsible for the transfer and computation of the data to be encrypted or decrypted. Nevertheless, in order to properly evaluate the cost of having a dedicated hardware structure for the key scheduling, a structure capable of performing this computation for 128-bit input keys is also proposed.

### *3.1 CLEFIA Data Ciphering Structures*

Regarding the optimization of the computational structure, and as suggested in [9] and validated by the structures proposed in [4, 11], faster implementation of the CLEFIA algorithm can be achieved with the usage of T-Boxes. T-Boxes merge the computation of the S-Box and part of the diffusion matrices operations with the linear transformation layers, compressing the resulting structure into a lookup table, also resulting on a reduction of the critical path [3].

In the CLEFIAs F-functions operation, T-Boxes can be used to replace $S_0$, $S_1$, $M_0$, and $M_1$, by the lookup operations depicted by (11), followed by the bitwise XOR operations ( additions over GF($2^8$)) [11]:

$$\begin{aligned}
T_{00} &= (S_0, 02{\times}S_0, 04{\times}S_0, 06{\times}S_0)\\
T_{01} &= (02{\times}S_1, S_1, 06{\times}S_1, 04{\times}S_1)\\
T_{02} &= (04{\times}S_0, 06{\times}S_0, S_0, 02{\times}S_0)\\
T_{03} &= (06{\times}S_1, 04{\times}S_1, 02{\times}S_1, S_1)\\
T_{10} &= (S_1, 08{\times}S_1, 02{\times}S_1, 0A{\times}S_1)\\
T_{11} &= (08{\times}S_0, S_0, 0A{\times}S_0, 02{\times}S_0)\\
T_{12} &= (02{\times}S_1, 0A{\times}S_1, S_1, 08{\times}S_1)\\
T_{13} &= (0A{\times}S_0, 02{\times}S_0, 08{\times}S_0, S_0)
\end{aligned} \tag{11}$$

The resulting T-Boxes have an 8 bit input bus and a 32-bit data output. These lookup tables can be implemented in two ways: (1) using logic gates (or LUT in FPGAs) [4]; (2) or using dedicated memory blocks. Given that most of the current reconfigurable devices, in particular FPGAs, have dedicated embedded memory blocks designated as BRAMs, the T-Box implementation can be efficiently realized with these components. This allows to achieve faster and less LUT demanding solutions [5]. Further optimizations can be accomplished in terms of resource requirements taking into account that these tables perform identical calculations. Actually, $T_{00}$ and $T_{02}$, presented in (11), perform the exact same lookup operation, given the same input, only differing in a 16-bit shift of the output. The same applies to $T_{01}/T_{03}$, $T_{10}/T_{12}$, and $T_{11}/T_{13}$. Given this characteristic and due to the existence of dual port BRAMs in most FPGA devices, two of these lookup operations can be realized in a single BRAM component. The additional shift operations can be implemented by hardwired routing, without additional area overhead. The remaining hardware required to perform the round computations is composed by a tree of bitwise XOR operations (additions over GF($2^8$)) [11]. Apart from the round computation, the addition of the four 32-bit whitening keys also needs to be performed, two at the beginning and two more at the end of the final round computation. The resulting structure, depicted in Fig. 3, is similar to the one proposed in [11] and herein designated as type I CLEFIA structure.

In order to obtain an even more compact structure for the CLEFIA implementation, the symmetry between the $F_0$ and $F_1$ functions is further explored. The main difference between $F_0$ and $F_1$ resides in the $M_0$ and $M_1$ tables, as depicted in Fig. 2. A more compact structure can be derived by merging the computation of these two tables into a single lookup table. Combining the resulting table for both $M_0$ and $M_1$ and taking into consideration the computation structures of the F-functions, a single merged structure, able to compute both $F_0$ and $F_1$, can be derived.

The resulting merged T-Boxes, capable of computing both the $F_0$ and $F_1$, use a 9-bit input divided in two parts, 8 bits for the data and one other bit for the F-function selection. As in the type I CLEFIA structure, a 32-bit value is outputted by this T-Box. However, for the implementation of these T-Boxes, the BRAMs need to store twice the data. While in type I the T-Box blocks require 256×32 bits=8 kbits,

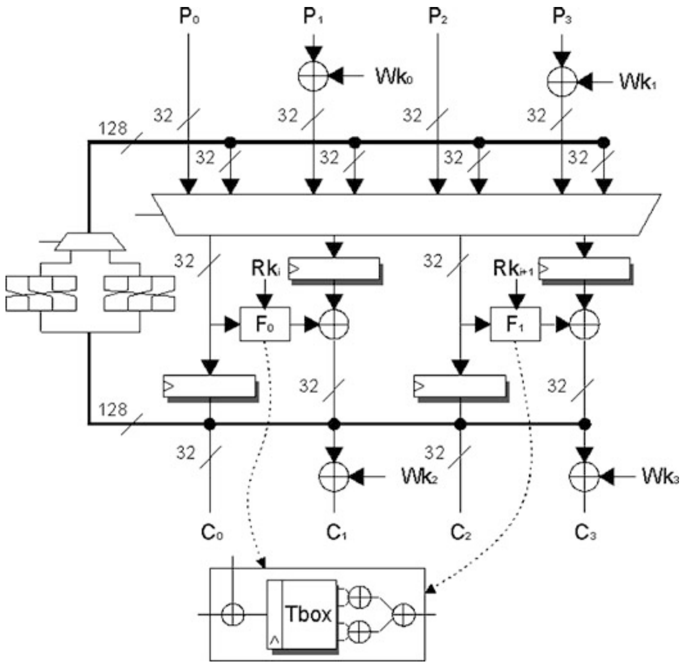


**Fig. 3** Type I CLEFIA structure

in the type II structure, the memory block needs 512×32 bits=16 kbits to store the lookup values. Most FPGA devices have 18 kbit BRAMs units, meaning that for these FPGAs, the resulting T-Box blocks for the type II structure will occupy the entire BRAM unit but will not require any more BRAMs. For the type I structure, only half of each used BRAM is occupied.

In the T-Box of the type II structure, the selection of which function is to be computed within the T-Box is performed by a single bit value at the most significant bit of the address bus of the BRAM, as depicted in Fig. 4 by the $T_0/T_1$ selector in the BRAM.

Being able to perform the lookup operation of the F-functions within a single component, an additional level of folding can be applied, performing the computation of $F_0$ and $F_1$ in the same hardware structure. With this technique, approximately half of the hardware resources are needed, apart from the additional selection logic. Consequently the computation of each round will now require two clock cycles, twice as much as in the type I structure.

Note that, even though a data dependency exists between the data of each round, with a careful scheduling of the round operations and data storage, round $i+1$ can start its computation before round $i$ has completely finished its computation. With this in mind a pipeline stage can be added to type II CLEFIA structure dividing the computation into two stages. Table 1 depicts the proposed computation scheduling,

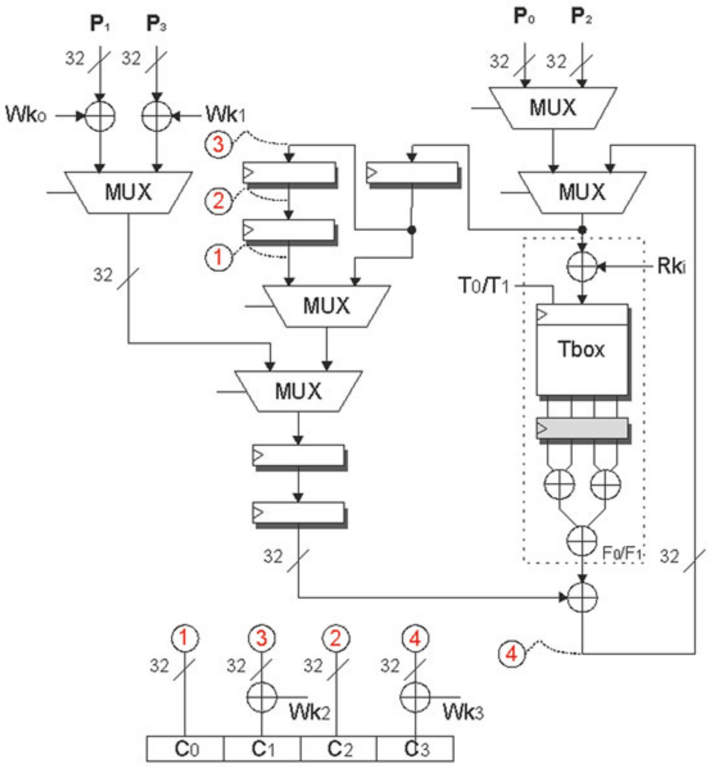


**Fig. 4** Type II CLEFIA structure

where $P^0$ refers to 32 bits of the 128-bit input and *i* refers to the respective round being computed. Note that this computation scheduling represents the $GFN_{4,n}$ along with the addition of the whitening keys.

In this improved structure the computation of each round is performed in two clock cycles. In the first stage, one of the F-functions is computed by the T-Box structures. In the second stage, the remaining data and round key additions are performed. With the proposed schedule, and considering the resulting hardware structure within the FPGA fabric, a pipeline stage can be placed in such a way that stage one and stage two are relatively balanced. The real gain in this structure comes from the fact that while one stage computes one-half of the CLEFIA algorithm, the other stage computes the other half of the CLEFIA algorithm. The resulting computational structure is depicted in Fig. 4.

Note that the computation in each round can now be performed in approximately half of the time as in the type I structure. Thus, it is expected that an approximate ciphering throughput can be achieved, given that no pipeline stalling exists. In order to optimize the data path to the used FPGA technology, the pipeline stage register, depicted in dark in Fig. 4, can be placed in different parts of the data path. Several

**Table 1** Type II structure pipeline scheduling

| i | First stage | Second stage | Output |
|---|---|---|---|
| 1 | $T_0(P_0^0+RK_0)$ | – | – |
| 2 | $T_1(P_2^0+RK_1)$ | $(T_{00}+T_{01}+T_{02}+T_{03})+WK_0+P_1^0$ | $P_0^1$ |
| 3 | $T_0(P_0^1+RK_2)$ | $(T_{10}+T_{11}+T_{12}+T_{13})+WK_1+P_3^0$ | $P_2^1$ |
| 4 | $T_1(P_2^1+RK_3)$ | $(T_{00}+T_{01}+T_{02}+T_{03})+P_1^1$ | $P_0^2$ |
| 5 | $T_0(P_0^2+RK_4)$ | $(T_{10}+T_{11}+T_{12}+T_{13})+P_3^1$ | $P_2^2$ |
| 6 | $T_1(P_2^2+RK_5)$ | $(T_{00}+T_{01}+T_{02}+T_{03})+P_1^2$ | $P_0^3$ |
| 7 | $T_0(P_0^3+RK_6)$ | $(T_{10}+T_{11}+T_{12}+T_{13})+P_3^3$ | $P_2^3$ |
| ... | ... | ... | ... |
| $2\times n-2$ | $T_1(P_2^{n-2}+RK_{2n-4})$ | $(T_{00}+T_{01}+T_{02}+T_{03})+P_1^{n-2}$ | $P_0^{n-1}=C_0$ |
| $2\times n-1$ | $T_0(P_0^{n-2}+RK_{2n-3})$ | $(T_{10}+T_{11}+T_{12}+T_{13})+P_3^{n-2}$ | $P_2^{n-1}=C_2$ |
| $2\times n$ | $T_1(P_2^{n-1}+RK_{2n-2})$ | $(T_{00}+T_{01}+T_{02}+T_{03})+P_1^{n-1}+WK_2$ | $P_1^{n-1}=C_1$ |
| $2\times n+1$ | – | $(T_{10}+T_{11}+T_{12}+T_{13})+P_3^{n-1}+WK_3$ | $P_3^{n-1}=C_3$ |

realized implementations suggested that, in the considered devices, this register is best placed at the output of the BRAMs. In all the families of the considered FPGAs, a register is intrinsically located at the beginning of the BRAMs, which cannot be removed, thus defining the frontier of the second pipeline stage.

On the left side of the resulting structure, a set of registers can be observed. These registers are used to store the temporary round values, needed by the proposed schedule (see Table 1).

## 3.2 *CLEFIA Key Scheduling Structure*

In order to adequately decide whether or not to allocate dedicated hardware to the key scheduling process, this section presents a possible structure to perform this computation for 128-bit input keys.

The expansion of the input key into the several rounds keys uses the already described GFN function. However, only the 128-bit calculation shares the same $\mathrm{GFN}_{4,n}$ function with the CLEFIA ciphering calculation. The expansion of the 192- and 256-bit input keys require the use of a GFN network with an eight-branch structure ($\mathrm{GFN}_{8,n}$). This means that the hardware structure described above for the CLEFIA ciphering computation cannot be use for these last two key sizes. This is the main reason why only 128-bit input key scheduling structures are proposed in the related art [4, 11]. As explained in Sect. 2.3, the key scheduling process is divided into two steps. In the first step, the 128-bit $L$ value is computed using the same GFN structure as the ciphering calculation ($L=\mathrm{GFN}_{4,n}(\mathrm{CON}_{0:23}^{128},K)$). Given that this calculation only differs from the ciphering process in the input values and the nonexistence of the whitening keys, the structure type II (depicted in Fig. 4) can be used. For this re-usage the round keys ($\mathrm{RK}_i$) input must be feed with the needed constant values ($\mathrm{CON}_{0:23}^{128}$), the whitening keys ($\mathrm{WK}_i$) set to zero, and the data input

**Algorithm 4** - Generation of RK$_i$ for 128-bit input Keys

```
**Input** : $L_{0,0}|L_{1,0}|L_{2,0}|L_{3,0} \leftarrow L,\ \ K_0|K_1|K_2|K_3 \leftarrow K$
**for** $i = 0$ **to** 8 **do**
    **for** $j = 0$ **to** 3 **do**
        $T = L_{i,j} \oplus (CON^{128}_{24+4i+j})$
        **if** $i$ is $odd$ **then** $T = T \oplus K_j$ **end if**
        $RK_{4i+j} \leftarrow T$
    **end for**
    $L_{i+1} = \Sigma(L_i)$
**end for**
**Output** : $RK$
```

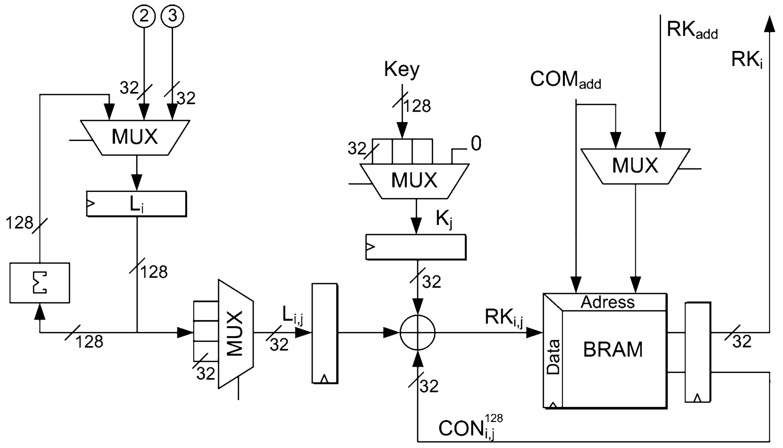


**Fig. 5** CLEFIA 128-bit key expansion structure

block ($P_{0:3}$) must be the 128-bit input key (K). In the second step the 40 round keys are generated using the L and K values. In order to obtain a more compact structure for this computation, Algorithm 2 can be rewritten as depicted in Algorithm 4.

From Algorithm 2, a compact 32-bit data flow can be obtained, resulting in the structure depicted in Fig. 5. Both the needed constant values ($CON^{128}$) and the generated round keys (RK$_i$) are stored in the same BRAM. This BRAM operates both as a RAM, storing and reading the round keys from the lower part of this memory, and as a ROM, where the constant values are initialized in the upper part of this memory. Note that the address input of the RK$_i$ output can be either the round key, when data is being ciphered, or the constant address, when the GFN hardware is being used to generate the L value.

In the leftmost side of the structure depicted in Fig. 5, the L value is loaded and the DoubleSwap function ($\Sigma()$) is processed. Rather than directly loading the entire 128-bit L value from the GFN computation, this value is read from the internal points 2 and 3 depicted in Fig. 4. Given that these values are registered and not in the critical path nor directly feed to the ciphered data output, routing and fanout problems are mitigated. For the calculation of the round keys, the L value is read during rounds 2$n$ to 2$n$+2 of scheduling depicted in Table 1.

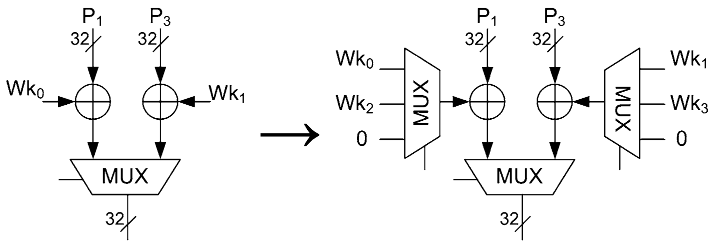


**Fig. 6** Whitening keys multiplexing

For the 128-bit input key, the whitening keys are directly obtained from the input key as depicted in (2). The difference in the encryption and decryption processed is only in the order that the round keys and the whitening keys are feed into the computation. For the round keys this order inversion is accomplished by addressing the depicted BRAM in the reverse order. However, the whitening keys are not in the memory and need to multiplexed according to the ciphering mode. Additionally, the whitening keys added before the GFN computation, depicted in Fig. 4 as $Wk_0$ and $Wk_1$, also need to be zeroed when performing the key scheduling computation. The resulting modification to the addition of the whitening keys at the input is depicted in Fig. 6.

## 4 Evaluation and Related Art Comparison

In this section, experimental results for the proposed structures, on two distinct Xilinx FPGAs technologies, are presented and compared with the work in the existing related art [4,11]. In the first part of this analysis, only the main computation structure is considered, without the key scheduling computation. In order to evaluate the presented CLEFIA structures on low-cost FPGA devices, the Xilinx Spartan 3E technology was selected. For higher end devices, the Virtex 4 technology was selected.

### *4.1 CLEFIA Data Ciphering Analysis*

The implementation results on the Spartan device, presented in Table 2, suggest that, on Spartan 3E devices, throughputs in the order of 700 Mbit/s can be achieved for both structures at a resource cost of 624 LUT and 5 BRAMs for type I structure and 323 LUT and 3 BRAMs for the more compact type II structure. Note that in all implementations of the proposed structures without key scheduling, an extra BRAM storing the round keys is added. This BRAM is used to store the already expanded round keys used in the CLEFIA computation.

**Table 2** Summary of obtained performance results

| | | Key size | Clock cycles | Throughput (Mbps) | Efficiency (Mbps/Slice) |
|---|---|---|---|---|---|
| Spartan 3E | Type-I | 128 | 18 | 768 | 1.2 |
| | | 192 | 22 | 628 | 1.0 |
| | | 256 | 26 | 531 | 0.9 |
| | Type-II | 128 | 36 | 690 | 2.1 |
| | | 192 | 44 | 564 | 1.8 |
| | | 256 | 52 | 477 | 1.5 |
| | | 128 | 18 | 1273 | 2.0 |
| Virtex 4 | Type-I | 192 | 22 | 1042 | 1.7 |
| | | 256 | 26 | 881 | 1.4 |
| | Type-II | 128 | 36 | 1077 | 4.5 |
| | | 192 | 44 | 851 | 3.9 |
| | | 256 | 52 | 720 | 3.0 |

Results on a higher end device were also obtained, namely, for the Virtex 4 technology. In these FPGAs throughputs in the order of 1.0 Gbit/s can be achieved with a relatively low resource cost for both structure types. For the type II structure a resource occupation of 238 LUTs and 3 BRAMs is achieved. Throughput/Slice efficiency metrics up to 4.5 (Mbps/Slice) are achieved for the proposed type II structure. Table 2 presents the obtained throughputs for the two presented CLEFIA structures according to the key size. As expected when longer ciphering keys are used the performance efficiency of the ciphering computation decreases.

Considering the related art, the presented structures implemented on FPGAs cannot be directly compared with the ones proposed in [11], since these authors focused their work on ASIC technology. Nevertheless, as mentioned above, the presented type I structure is similar to the type A structure proposed in [11], which suggests the best throughput/area efficiency metric. With this, comparing the presented type I structure with the proposed type II structure allows us to evaluate the improvements of the proposed modification to the computation structure. Experimental results suggest an area reduction between 48 % and 61 %, at the expense of a throughput reduction between 10 % and 15 %, for the Spartan 3E and Virtex 4 technologies, respectively. These values suggest an improvement of the throughput/slice efficiency metric of 1.75 times on the Spartan 3E technology and more significantly of 2.25 times for the Virtex 4 technology.

This significant efficiency improvement is due to the component reutilization accomplished by the pipeline and data path rescheduling. Even though the number of cycles needed to cipher a data block doubles, the operating frequency also increases (194 MHz instead of 108 MHz for the Spartan 3E), given that the computational data path is evenly divided in two.

Note that, while the original structure (type A) proposed in [11] performs the key scheduling, the above analysis was performed between the two structures,

**Table 3** Hardware performance comparison of CLEFIA implementations

| | Takeshi [11] | Ours Type-I | Ours Type-II | Ours Type-I | Ours Type-II | Kryjak [4] |
|---|---|---|---|---|---|---|
| Device | ASIC | XC3S1200-4 | | XC4LX200-11 | | |
| # Slices | 21[a] | 624 | 323 | 625 | 238 | 9896 |
| # BRAMs | n.a. | 4+1 | 2+1 | 4+1 | 2+1 | 0 |
| Frequency (MHz) | 746 | 108 | 194 | 179 | 303 | 167 |
| Latency (cycles) | 18 | 18 | 36 | 18 | 36 | 18 |
| Throughput (Mbps) | 5306 | 768 | 690 | 1273 | 1077 | 21376 |
| Throughput/Slice (Mbps/S) | n.a. | **1.2** | **2.1** | 2.0 | **4.5** | **2.1** |

[a]Kgates not Slices

without the key expansion hardware. The analysis considering the structures with key scheduling is performed at the end of this section.

Considering the implementation of the described T-Boxes using LUTs, a total of 185 slices would be needed for each T-Box, using a Spartan 3E device. Thus 1,480 slices would be needed to implement the eight required T-Boxes in type I structure. Also, the delay imposed by this kind of implementation, would lead to a longer critical path and consequently lower frequencies and throughputs.

In [4] a fully unfolded structure is proposed, justifying the extremely high throughput obtained (21 Gbit/s). However, this throughput comes at the expense of an excessively high area resource usage, as depicted in Table 3. Moreover, this structure does not allow for the use of encryption modes other than ECB or an input key different than a 128-bit one. On a Virtex 4 FPGA, a throughput/slice efficiency metric of 2.1 is obtained for [4] in comparison with a throughput/slice of 4.5 for the proposed type II structure. The comparison between the two implementations suggests a 2.14 times better throughput/slice metric on Virtex 4 devices. Throughputs significantly above 1 Gbit/s were not usually a target, since most FPGA applications do not require such high-throughput values.

In the above discussion the BRAMs needed by the proposed structures were not considered. However, the number of used slices is within the percentage of used BRAMs that are available in the FPGA, either we use them or not. In the structure propose in [4], no BRAMs are used.

## 4.2 *CLEFIA Key Scheduling Analysis*

The above discussion only considered the hardware implementation of the CLEFIA ciphering, without the key scheduling computation. However, when no auxiliary processing units exists to perform this computation or when the key scheduling needs to be accelerated, a ciphering core with key scheduling capability must be deployed.

**Table 4** Hardware performance of the 128-bit input key scheduling

| | Key expansion structure | | CLEFIA with key expansion | | CLEFIA without key expansion | |
|---|---|---|---|---|---|---|
| Device | XC3 | XC4 | XC3 | XC4 | XC3 | XC4 |
| # Slices | 218 | 171 | 574 | 481 | 323 | 238 |
| # BRAMs | 1 | | 3 | | 1 | |
| Frequency (MHz) | 193 | 285 | 184 | 287 | 194 | 303 |
| Latency (cycles) | 60 | | 36 | | 36 | |
| Throughput (Mbps) | 412 | 608 | 654 | 1020 | 690 | 1077 |
| Throughput/Slice (Mbps/S) | n.a. | n.a. | 1,1 | 2,1 | 2,1 | 4,5 |

Table 4 presents the obtained results for the key expansion structure presented in Fig. 5 and the resulting CLEFIA ciphering core with 128-bit key scheduling. The obtained experimental results suggest that the resulting CLEFIA structure with 128-bit key scheduling requires between 78 % and 100 % more area resources regarding the CLEFIA structure without key scheduling. Regarding the maximum achievable operating frequency, a degradation of about 5 % can be observed. This degradation may be justified by a more demanding routing caused by the additional logic added by the key expansion logic.

In conclusion, the 128-bit key scheduling can be added to the CLEFIA ciphering, while maintaining approximately the same ciphering throughput, but with significant area usage increase. Nevertheless, with this dedicated hardware structure, a key scheduling of up to 4.75 million 128-bit input keys per second can be performed on a Virtex 4 device.

## 5 Conclusion

This chapter presents two compact CLEFIA structures and analyses the existing related art. The presented structures were designed having in mind reconfigurable technologies, in particular FPGA with BRAMs, but can be easily targeted to other technologies such as ASIC. The main focus of this work is given to the computation of the CLEFIA ciphering without key scheduling. Nevertheless, the implementation of the key scheduling for 128-bit input keys is also considered and analysed.

Herein, three approaches are studied: one using a fully unrolled approach [4]; one folded structure computing one cipher round per clock cycle, similar to the one presented in [11]; and finally, one which collapses the round computation by carefully scheduling the operations with the use of an additional pipeline stage. The result analysis suggests that with the collapsing of the computation, as realized in the presented type II structure, significant gains in the resource usage can be achieved while maintaining identical ciphering throughput metrics. Experimental results suggest that efficiency improvements of 2.25 times can be achieved as well as a reduction in the required area resources by up to 60 % at a performance cost of at most 15 %. When comparing the fully folded structure with the fully unfolded one, a clear difference in throughput and area is obtained. The fully unfolded structure is able to achieve throughputs up to 21 Gbit/s but at an extremely high area cost and lack of flexibility. In terms of throughput/slice, the fully folded structure is able to achieve 2.14 times better efficiency results.

Regarding the key scheduling in hardware, experimental results suggest that no significant impact is imposed in the operating frequency. However, area resource usage significantly increases. Due to its cost, key scheduling should only be performed in dedicated hardware structures when truly necessary.

In conclusion, compact structures for the implementation of the CLEFIA encryption algorithm can be developed and throughputs near 1 Gbit/s can be achieved with low resource usage, even on low-cost FPGA devices.

**Acknowledgements** This work was supported by the Portuguese Foundation for Science and for Technology (INESC-ID multi-annual funding) through the PIDDAC Program funds and by the QREN Program under contract N$^o$ 3487.

## References

1. Chen H, Wu W, Feng D (2007) Differential fault analysis on CLEFIA. Information and communications security, pp 284–295 http://direct.bl.uk/bld/PlaceOrder.do?UIN=221532696&ETOC=RN&from=searchengine
2. Elbirt A, Yip W, Chetwynd B, Paar C (2001) An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. IEEE Trans Very Large Scale Integration (VLSI) Syst 9(4):545–557 http://direct.bl.uk/bld/PlaceOrder.do?UIN=098438233&ETOC=RN&from=searchengine
3. Good T, Benaissa M (2005) AES on FPGA from the fastest to the smallest. Cryptographic hardware and embedded systems–CHES 2005, pp 427–440 http://direct.bl.uk/bld/PlaceOrder.do?UIN=173568703&ETOC=RN&from=searchengine
4. Kryjak T, Gorgon M (2009) Pipeline implementation of the 128-bit block cipher CLEFIA in FPGA. In: International conference on field programmable logic and applications, FPL 2009, pp 373–378. IEEE
5. Rodriquez-Henriquez F, Saqib N, Díaz-Pérez A, Koc C (2006) Cryptographic algorithms on reconfigurable hardware, vol 978. Springer, New York
6. Shirai T, Mizuno A (2007) A compact and high-speed cipher suitable for limited resource environment. In: Third ETSI security workshop presentation. Sophia-Antipolis, France
7. Shirai T, Shibutani K (2006) On Feistel structures using a diffusion switching mechanism. In: Fast software encryption, pp 41–56. Springer, New York
8. Shirai T, Shibutani K, Akishita T, Moriai S, Iwata T (2007) The 128-bit blockcipher CLEFIA. In: Fast software encryption, pp 181–195. Springer, New York
9. SONYCorporation (2007) The 128-bit block cipher CLEFIA security and performance evaluations. URL http://www.sony.net/Products/cryptography/clefia/technical/data/clefia-eval-1.0.pdf. Cited 3 December
10. SONYCorporation (2007) The 128-bit blockcipher CLEFIA - algorithm specification. URL http://www.sony.net/Products/cryptography/clefia/technical/data/clefia-spec-1.0.pdf. Cited 3 December

11. Sugawara T, Homma N, Aoki T, Satoh A (2008) High-performance ASIC implementations of the 128-bit block cipher CLEFIA. In: IEEE international symposium on circuits and systems, ISCAS 2008, pp 2925–2928. IEEE
12. Tsunoo Y, Tsujihara E, Shigeri M, Suzaki T, Kawabata T (2008) Cryptanalysis of CLEFIA using multiple impossible differentials. In: International symposium on information theory and its applications, ISITA 2008, pp 1–6. IEEE

# A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions

**Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont**

## 1 Introduction

An on-chip physical unclonable function (PUF) is a chip-unique challenge-response mechanism exploiting manufacturing process variation inside integrated circuits (ICs). The relation between a challenge and the corresponding response is determined by complex, statistical variation in logic and interconnect in an IC. A PUF has several applications in the field of hardware-oriented security. For example, it can be used in device authentication and secret key-generation [20]. Guajardo et al. discussed the use of PUFs for intellectual property (IP) protection, remote service activation, and secret-key storage [6]. A PUF-based RFID tag has also been proposed to prevent product counterfeiting [2, 3].

Since the inception of the idea of PUFs, different types of PUFs have been proposed so far. For example, Lim et al. proposed Arbiter PUF that exploits the delay mismatch between a pair of identically laid-out delay paths [11]. A PUF which is based on random start-up values of SRAM cells was proposed by Guajardo et al. [5]. A PUF based on an array of identically laid-out ring oscillators has also been proposed [20]. There are several other PUFs which are either enhanced versions of a previously proposed PUF or introduce new methods of generating PUF challenge-response pairs (CRPs).

The availability of several different PUFs gives us choices to select a particular one suitable for an application. However, it also raises few practical questions: how do we know if a PUF is efficient or not? How do we compare one PUF with another? Currently, there is no readily available method to fairly compare one PUF with another. A concrete as well as easy-to-use evaluation–comparison method will be useful for a designer who may want to employ a PUF in her design. Armknecht et al. expressed the same view in their work on formalizing the security features of

A. Maiti (✉) • V. Gunreddy • P. Schaumont
Virginia Tech, 1185 Perry Street, Blacksburg, VA 24061, USA
e-mail: abhranil@vt.edu; gvikash7@vt.edu; schaum@vt.edu

PUFs [1]. We propose a systematic method to evaluate the performance of PUFs and to make a fair comparison among them. As part of this work, we have identified the following goals:

First, we need to clearly define parameters that will quantify the performance of a PUF in a concrete manner. In order to do that, we not only propose our own PUF parameters but also explore several other parameters defined by other researchers. No analysis has been carried out so far to compare these parameters in order to estimate how effective they are in evaluating performance of a PUF. It might also be possible that many of these parameters are similar in nature or redundant. We aim to find any such cases to define a compact set of PUF parameters while removing redundancy.

Second, we aim to make the comparison method independent of the underlying PUF technique. For example, it should be able to compare a delay-based PUF like the RO PUF [20] with a memory-based PUF such as the SRAM PUF [5]. Therefore, we focus on the statistical properties of the binary PUF responses. This is possible because every PUF produces binary responses, or responses can be converted to binary form irrespective of the underlying technique.

Third, we aim to make our experiments repeatable not only by proposing PUF performance evaluation parameters but also by providing access to the raw data of our measurements. The measurements have been done on PUFs implemented in a low-cost, off-the-shelf reconfigurable device, Spartan 3E FPGA. We believe that the PUF research community may benefit from access to such data.

With these goals in mind, we present the following contributions:

- We first propose three measurement dimensions of a PUF. They are device, time, and space. The PUF performance parameters will be defined along these dimensions. We explain the significance of these dimensions in detail and show how several PUF parameters can be defined based on them.
- We propose that a set of $m$ parameters be used to evaluate and compare the performance of different PUFs while each PUF may have different number of challenge and response bits. This is possible as the proposed parameters purely rely on the statistical properties of the binary PUF responses. A simple view of the idea is presented in Fig. 1. As a preliminary effort, we propose seven parameters as part of the method: uniqueness, reliability, randomness, steadiness, bit-aliasing, diffuseness, and probability of misidentification (PMSID). We have defined some of these parameters while rests have been selected from the works done by other researchers. We explain in detail why these parameters are useful in evaluating the performance of PUFs.
- We compare two different PUFs: the RO PUF and the APUF using the above-mentioned parameters. We used measured PUF data for this comparison. A detailed comparison result is presented to validate the proposed method.
- Finally, we present an online database that holds the results of our experiments. We present measurements in 193 FPGAs under standard operating conditions and in five FPGAs under varying operating conditions. The database is scripted and enables a user to easily formulate queries to extract specific record sets.

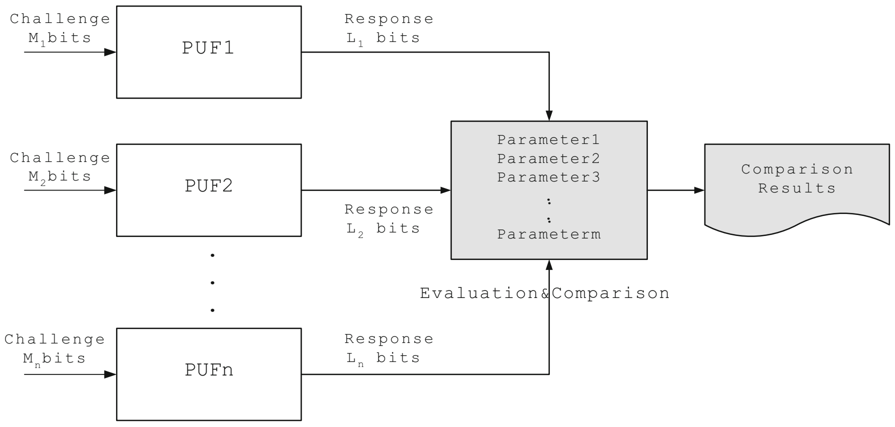


**Fig. 1** Basic idea of a PUF evaluation and comparison method

The PUF parameters proposed in this book chapter were developed using the measurements of this database.

The rest of this chapter is organized as follows. Section 2 gives an overview of different PUFs proposed so far in the research community. It also discusses several works related to the evaluation and comparison of performance of different PUFs. Section 3 introduces the dimensions of PUF measurements, defines four PUF parameters, and analyzes few PUF parameters from the literature. Based on our analysis, we propose a final set of parameters as the main building blocks of the evaluation–comparison method. Section 4 presents an analysis to compare the RO PUF with the APUF. In Sect. 5, we describe our online PUF database. Finally, we present some concluding remarks in Sect. 5.

## 2 Background

In this section, we briefly discuss the history of the PUF technology since it was introduced. We also discuss several research contributions that are related to PUF performance evaluation.

### *2.1 Chronology of PUFs*

One of the seminal works in the area of PUF is that of Lofstrom et al. in 2000. It exploited mismatch in silicon devices for identification of ICs [12]. Though the authors did not call it a PUF, the objective of their work was very similar

**Table 1** Different types of PUFs

| | |
|---|---|
| 2000 | IC identification using device mismatch [12] |
| 2001 | Physical one-way function [18] |
| 2002 | Physical random function [4] |
| 2004 | Arbiter PUF [11] |
| 2006 | Coating PUF [22] |
| 2007 | Ring oscillator PUF [20], SRAM PUF [5] |
| 2008 | Butterfly PUF [10] |
| 2009 | PUF using power distribution system of an IC [7] |
| 2010 | Glitch PUF [21] |
| 2011 | Mecca PUF [9] |

to that of a PUF. In 2001, Pappu et al. presented the concept of physical one-way function which led to the idea of PUF [18]. After that, many different types of PUFs have been proposed. Almost every year, at least one new PUF circuit was proposed. Table 1 shows a year-wise list of different PUFs starting from the year 2000. Though this list is not exhaustive, it shows us a picture about how the researchers tried to build different PUFs. For more information, one may refer to the work by Maes et al. which presents a comprehensive discussion on different PUFs proposed so far [14].

Variety of PUFs does support the need to build a systematic method to evaluate and compare their performance. Despite the existence of multiple PUF techniques, not many of them have been actually integrated in a system so far. An evaluation–comparison method may make it easier for a system designer to select a PUF that suits the best for a particular application leading to more utilization of the PUF technology.

## *2.2 Related Work*

We discuss related research on the performance measurement of PUFs. Majzoobi et al. proposed several parameters to test the security of PUFs [16]. They tested three security properties of a PUF: predictability, sensitivity to component accuracy, and susceptibility to reverse engineering. Two variants of the Arbiter PUF, linear and feed-forward, were tested. However, no comparison of different PUFs were made.

In another work, Armknecht et al. formalized three properties of a PUF: robustness, unclonability, and unpredictability [1]. The analysis result presented in this work is based on the SRAM-based PUF proposed by Gujardo et al. [5]. No comparison between multiple types of PUFs was presented in this work.

Van der Leest et al. tested the performance of D flip-flop-based PUF (DFF-based PUF) implemented in ASIC using several parameters [23]. This PUF was originally proposed by Maes et al. using flip-flops in FPGAs [13]. This work applied Hamming weight test, inter-chip uniqueness test, and NIST randomness test on PUF responses. This work presented a comparison between the SRAM-based PUF, the

Butterfly PUF [10], and the DFF-based PUF. However, this comparison did not use the statistical properties of the PUF responses except the entropy. Apart from the entropy, the comparison was made based on the number of gates used to implement the PUF, the platform used (ASIC/FPGA), and the spread on a die.

A comprehensive performance evaluation of the APUF has been done by Hori et al. [8]. In this work, several PUF parameters were defined systematically and were validated based on a large set of PUF data. The PUF parameters proposed by this work are uniqueness, randomness, correctness, steadiness, and diffuseness. We have studied this work as a part of the PUF comparison effort in this work. We will discuss this work in detail in the subsequent sections. However, this work also did not present any comparison analysis on different types of PUFs.

The work by Maes et al. presented a detailed comparative analysis between several PUFs [14]. One of the comparisons presented in this work was made using several parameters of PUFs such as evaluability, uniqueness, reproducibility, physical unclonability, mathematical unclonability, unpredictability, one-way-ness, and tamper evidence. Another comparison analysis, presented in this work, includes randomness, challenge-response mechanism, CRP space, implementation platform, average inter-chip and average intra-chip variation in PUF responses, entropy, tamper evidence, and model building as the comparison metrics. The results presented for the CRP space, the entropy, and the inter- and intra-chip variation were quantitative while the others were qualitative.

Our contribution in this work is different in many ways from the related work discussed. First, we define a basis for the PUF performance measurement/evaluation. We propose three measurement dimensions for this purpose. The PUF measurement dimensions have not been formally defined before. We explain how these dimensions capture useful information in order to evaluate the performance of PUFs. We, then, define few parameters using the proposed dimensions to evaluate the statistical property of the PUF responses. Hori et al. defined their PUF parameters using a similar approach, but they did not explicitly define the dimensions [8]. Second, we analyze several PUF evaluation parameters to point out any redundancy that may exist among them. Based on our analysis, we propose a compact set of parameters for PUF evaluation as well as comparison. We did not find any work in the literature that has made a similar effort. Finally, we compare two different PUFs: the RO PUF and the APUF. Unlike previous efforts, our comparison is done entirely quantitatively using the parameters we proposed.

## 3 PUF Evaluation and Comparison Method

In this section, we first introduce the PUF measurement dimensions. Then we define four PUF parameters to quantify several important quality factors of a PUF. Next, we analyze few parameters defined by other researchers. We compare them with the parameters we defined. Finally, we propose a set of parameters as the components of the evaluation–comparison method.

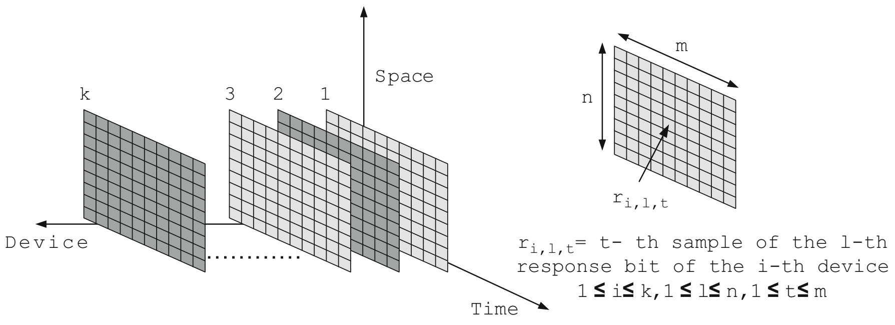

Space
k
3
2
1
Device
Time
m
n
$r_{i,l,t}$
$r_{i,l,t}$= t- th sample of the l-th response bit of the i-th device
$1 \le i \le k, 1 \le l \le n, 1 \le t \le m$

**Fig. 2** Dimensions of PUF measurement

## *3.1 PUF Measurement Dimensions*

Figure 2 shows three different dimensions of PUF measurement along three axes: device, space, and time. The inter-chip variation in PUF responses is captured using the device axis. The two other axes are used to capture the intra-chip variation.

The device axis represents the population dimension of PUF measurements. A PUF not only needs to generate random responses for a chip, the generated responses also need to uniquely identify the chip among several other chips of the same type. To estimate this property, one needs to measure a set of PUF instantiations in several chips/devices. Hence, we included the device axis. A set of $k$ devices have been shown in Fig. 2 to represent a population.

The space axis stands for the location of a single-bit response, $r$ in an $n$-bit response string, $R$. The rationale behind naming it the space axis is that the PUF response bits are generated at different physical locations on a chip. For example, in an SRAM PUF, the SRAM cell that produces a response $i$ has an on-chip location that is different from that of another SRAM cell that produces a response $j$. For Arbiter PUF, the location of the Arbiter that produces the response is fixed. However, the locations of the stimulated delay paths change depending on the challenge (whether straight connections or crisscross connections through a switch). To estimate the randomness of a PUF, we examine multiple response bits from a PUF. Hence, the space dimension becomes useful.

Finally, the time-dependent properties of a PUF are captured along the time axis. A critical attribute of a PUF is the reliability of the responses. It estimates how consistently the responses can be generated against varying operating conditions such as variable ambient temperature and fluctuating supply voltage. To estimate the reliability, we take multiple samples of the responses at different instances of time. Samples of PUF responses are also useful in estimating the circuit aging effect on PUFs.

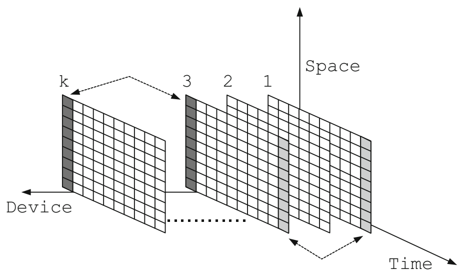


**Fig. 3** PUF uniqueness evaluation

## 3.2 *Defining PUF Parameters*

Now, we will define four PUF parameters based on the measurement dimensions introduced. These parameters are uniqueness, reliability, uniformity, and bit-aliasing. We formalized these parameters in one of our previous works on PUF characterization [15]. Each of these parameters quantifies an essential quality factor of a PUF as we will explain them in detail.

### 3.2.1 Uniqueness

Uniqueness represents the ability of a PUF to uniquely distinguish a particular chip among a set of chips of the same type. We use Hamming distance (HD) between a pair of PUF identifiers to evaluate uniqueness. If two chips, $i$ and $j$ $(i \neq j)$, have $n$-bit responses, $R_i$ and $R_j$, respectively, for the challenge $C$, the average inter-chip HD among $k$ chips is defined as

$$\text{Uniqueness} = \frac{2}{k(k-1)} \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \frac{\text{HD}(R_i, R_j)}{n} \times 100\,\%. \tag{1}$$

It is an estimate of the inter-chip variation in terms of the PUF responses and not the actual probability of the inter-chip process variation. In Fig. 3, it is shown that the inter-chip HD is estimated along the device axis. One comparison is shown in dark gray between the device 3 and the device $k$. Another comparison is shown in light gray between the device 1 and the device 3.

### 3.2.2 Reliability

PUF reliability captures how efficient a PUF is in reproducing the response bits. We employ intra-chip HD among several samples of PUF response bits to evaluate it. To estimate the intra-chip HD, we extract an $n$-bit reference response ($R_i$) from the chip $i$ at normal operating condition (at room temperature using the normal supply

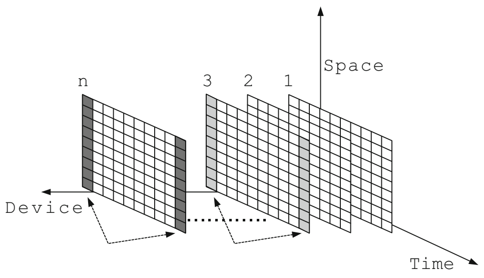


**Fig. 4** PUF reliability evaluation

voltage). The same $n$-bit response is extracted at a different operating condition (different ambient temperature or different supply voltage) with a value $R'_i$. $m$ samples of $R'_i$ are collected. For the chip $i$, the average intra-chip HD is estimated as follows:

$$\mathrm{HD}_{\mathrm{INTRA}} = \frac{1}{m}\sum_{t=1}^{m}\frac{\mathrm{HD}(R_i, R'_{i,t})}{n} \times 100\,\% \tag{2}$$

where $R'_{i,t}$ is the $t$th sample of $R'_i$. $\mathrm{HD}_{\mathrm{INTRA}}$ indicates the average number of unreliable/noisy PUF response bits. In other words, the reliability of a PUF can be defined as

$$\mathrm{Reliability} = 100\,\% - \mathrm{HD}_{\mathrm{INTRA}}. \tag{3}$$

Figure 4 shows how the reliability of a PUF is evaluated using the time dimension of PUF measurement. The two intra-chip Hamming distance measurements are shown along the time axis for the device 3 and the device $k$ with light gray and dark gray, respectively.

### 3.2.3 Uniformity

Uniformity of a PUF estimates how uniform the proportion of '0's and '1's is in the response bits of a PUF. For truly random PUF responses, this proportion must be 50 %. We define uniformity of an $n$-bit PUF identifier as its percentage Hamming weight (HW):

$$(\mathrm{Uniformity})_i = \frac{1}{n}\sum_{l=1}^{n} r_{i,l} \times 100\,\%$$

where $r_{i,l}$ is the $l$th binary bit of an $n-$bit response from a chip $i$. (4)

In Fig. 5, the uniformity of the device $k$ and the device 3 is evaluated along the space axis (marked in dark gray and light gray respectively).

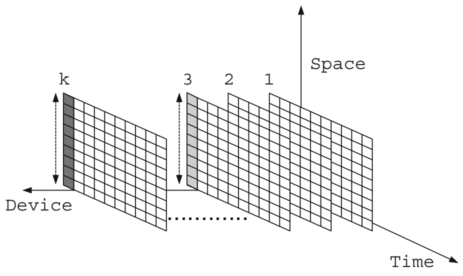


**Fig. 5** PUF uniformity evaluation

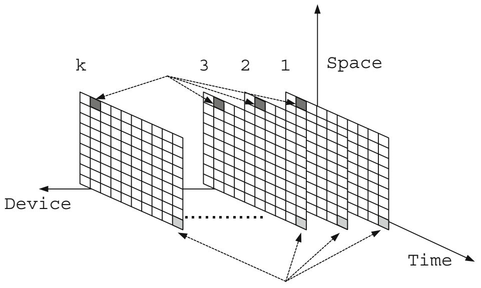


**Fig. 6** PUF bit-aliasing evaluation

### 3.2.4 Bit-Aliasing

If bit-aliasing happens, different chips may produce nearly identical PUF responses, which is an undesirable effect. We estimate bit-aliasing of the $l$th bit in the PUF identifier as the percentage Hamming weight (HW) of the $l$th bit of the identifier across $k$ devices:

$$(\mathrm{Bit}-\mathrm{aliasing})_l = \frac{1}{k}\sum_{i=1}^{k} r_{i,l} \times 100\,\%$$

$$\text{where } r_{i,l} \text{ is the } l\text{th binary bit of an } n-\text{bit response from a chip } i. \qquad (5)$$

In Fig. 6, the bit-aliasing is evaluated along the device axis for two different bit locations (marked in dark gray and light gray).

## 3.3 *Analysis of PUF Parameters Defined by Other Researchers*

In this section, we explore different PUF parameters defined by other researchers in the community. We also try to find out if there is any redundancy among these parameters. Table 2 shows few examples of different PUF parameters proposed by several research groups. In this work, we will be presenting an analysis of the parameters proposed by Hori et al. in [8] with a comparison of the parameters

**Table 2** Different PUF parameters

| | |
|---|---|
| Hori et al. [8] | Randomness<br>Steadiness<br>Correctness<br>Diffuseness<br>Uniqueness |
| Maiti et al. [15] | Uniformity<br>Bit-aliasing<br>Uniqueness<br>Reliability |
| Su et al. [19] | Probability of misidentification |
| Majzoobi et al. [16] | Single-bit probability<br>Conditional probability |
| Yamamoto et al. [25] | Variety |

introduced in Sect. 3.1. The reason we chose this work for analysis is that it has a similar effort to define PUF parameters like ours. Moreover, the authors of this work made a large PUF dataset, based on APUF, available for analysis. This motivated us to use the APUF dataset in carrying out a detailed comparison analysis with a similarly large dataset that we generated using the RO PUF. We also analyze the parameter "probability of misidentification" proposed by Su et al. [19]. This parameter estimates the rate of false positives in chip identification for a given number of noisy bits in the PUF responses.

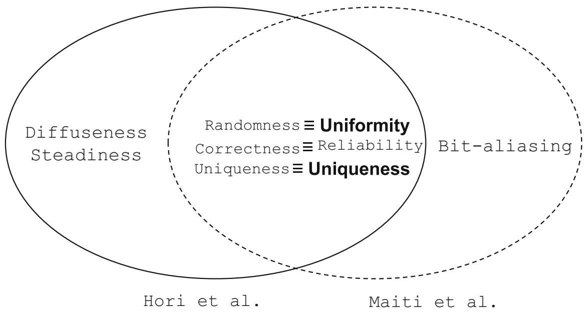
There are several other parameters existing in the literature. For example, Majzoobi et al. defined parameters such as single-bit probability and conditional probability [16]. A parameter called variety has been proposed by Yamamoto et al. [25].[1] As part of the future effort, we plan to analyze many of these parameters to enhance the evaluation–comparison method.

At first, we introduce few notations that will be used to describe the parameters. The notations are:

$N =$ total number of chips
$n =$ index of a chip $(1 \leq n \leq N)$
$K =$ total number of identifiers(IDs) generated per chip
$k =$ index of an ID in a chip $(1 \leq k \leq K)$
$T =$ total number of samples measured per ID
$t =$ index of a sample $(1 \leq t \leq T)$
$L =$ total number of response bits in an ID
$l =$ index of a response bit $(1 \leq l \leq L)$
$M =$ total number of ring oscillators
$m =$ index of an oscillator $(1 \leq m \leq M)$

[1]This term was introduced in the slides for the presentation of the paper [25] by the authors in CHES, 2011 [24].

**Fig. 7** Relation between parameters defined by Hori et al. and this work



The above notations, apart from $m$ and $M$ for the ring oscillators, have been proposed by Hori et al. [8]. We decide to keep these notations to define any PUF parameters except the fact that we use $r$ in place of $b$ (used in [8]) to denote a single response bit from a PUF. We notice that the parameters in Sect. 3.1 used $k$ as the total number of chips, $n$ as the total number of response bits from a PUF, and $m$ as the number of samples of the response bits. While we compare the parameters defined by the two groups, we will express all the PUF parameters using the notations above.

We have proposed four parameters as described in Sect. 3.1. They are uniqueness, reliability, bit-aliasing, and uniformity. The five parameters proposed by Hori et al. are uniqueness, randomness, correctness, steadiness, and diffuseness. Upon analyzing these parameters, we have found that there are similarities among these parameters. Figure 7 shows the relation between these parameters. There are three parameters from each group that are similar in definition while others are different. We explain all these parameters in detail.

### 3.3.1 Randomness Versus Uniformity

The randomness by Hori et al. is defined below:

$$-\log_2 \max(p_n, 1-p_n) \tag{6}$$

where

$$p_n = \frac{1}{K.T.L} \sum_{k=1}^{K} \sum_{t=1}^{T} \sum_{l=1}^{L} r_{n,k,t,l} \tag{7}$$

On the other hand, the uniformity parameter by us has been defined as follows:

$$\frac{1}{K.L}\sum_{k=1}^{K}\sum_{l=1}^{L} r_{n,k,l} \tag{8}$$

It can be noticed that the randomness includes $T$ samples of the response bits while the uniformity does not include the samples and is based on the reference/correct bits only. Also, the randomness is expressed in the min-entropy form. Otherwise, the two definitions are very similar in nature. Both of them estimate the ratio of ‘1’ vs. ‘0’ across all the response bits generated by a PUF.

### 3.3.2 Correctness Versus Reliability

The correctness parameter is defined as follows:

$$1-\frac{2}{K.T.L}\sum_{k=1}^{K}\sum_{t=1}^{T}\sum_{l=1}^{L}(r_{n,k,l}\oplus r_{n,k,t,l}). \tag{9}$$

On the other hand, the reliability is defined as below:

$$1-\frac{1}{K.T.L}\sum_{k=1}^{K}\sum_{t=1}^{T}\sum_{l=1}^{L}(r_{n,k,l}\oplus r_{n,k,t,l}). \tag{10}$$

The reliability parameter is different from the correctness only in terms of the factor by which it is normalized. The reliability is calculated based on the average value of the intra-chip HD, whereas the correctness is normalized by the maximum value of the fractional Hamming distance between the correct ID (the ID which is considered as the reference) and the sample IDs. There are few points that are not clearly mentioned in the definition of the correctness parameter proposed in [8].

It is defined using the sum of Hamming distances (SHD) normalized by $T$, $K$ and $L$. A parameter $c_{n,k,l}$ has been defined as the SHD between the correct bit $r_{n,k,l}$ and the generated bit $r_{n,k,t,l}$ through $T$ tests:

$$c_{n,k,l}=\sum_{t=1}^{T} r_{n,k,l}\oplus r_{n,k,t,l}. \tag{11}$$

However, it is not clear from the definition what the time instances of the measurements are. Since it is used to capture the effect of device defect, or aging, it can be assumed that the correct bit $r_{n,k,l}$ is measured before the aging or defect whereas $r_{n,k,t,l}$ is measured after the aging effect. If that is the case, then the following inequality (mentioned in [8]) does not necessarily hold good:

$$0\le c_{n,k,l}=\sum_{t=1}^{T} r_{n,k,l}\oplus r_{n,k,t,l}\le\frac{T}{2}. \tag{12}$$

This is because there might be a case when the aging or device defect might flip a correct bit in such a way that the subsequent $T$ samples produce a complementary value for more than $T/2$ samples. This inequality holds good *always* only if both $r_{n,k,l}$ and $r_{n,k,t,l}$ are measured during the same sampling instance which contradicts the definition of correctness.

### 3.3.3 Uniqueness by Hori et al. Versus Uniqueness by Maiti et al.

The uniqueness is defined by Hori et al. as

$$\frac{1}{K.L}\frac{4}{N^2}\sum_{k=1}^{K}\sum_{l=1}^{L}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}(r_{i,k,l}\oplus r_{j,k,l}) \tag{13}$$

The uniqueness is defined by us as

$$\frac{1}{K.L}\frac{2}{N(N-1)}\sum_{k=1}^{K}\sum_{l=1}^{L}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N}(r_{i,k,l}\oplus r_{j,k,l}). \tag{14}$$

In the case of the uniqueness, two definitions differ from each other with respect to the normalization factor. Hori et al. used the SHD of all the possible combinations of the PUF identifiers as the normalization factor. For a set of $n$ chips, the value of that factor is $K.L.N^2/4$. On the other hand, we used the total number of all possible pairwise combinations of response bits as the normalizing factor whose value is $K.L.N.(N-1)/2$. For a large value of $N$, the normalization factor used by us is approximately two times bigger than that used by Hori et al.

### 3.3.4 Parameters Uniquely Defined by Both the Groups

The term bit-aliasing is uniquely defined by us. On the other hand, the steadiness and the diffuseness are uniquely defined by Hori et al. The diffuseness is same as the uniqueness except the fact that the diffuseness is defined inside a single chip among several different IDs while the uniqueness is measured across several chips.

#### Steadiness

The steadiness measures the degree of bias of a response bit towards ‘0’ or ‘1’ over $T$ samples. It is defined as

$$S_n = 1 + \frac{1}{K.L}\sum_{k=1}^{K}\sum_{l=1}^{L}\log_2 \max(p_{n,k,l}, 1-p_{n,k,l}) \tag{15}$$

where

$$p_{n,k,l} = \frac{1}{T}\sum_{t=1}^{T} r_{n,k,t,l}. \tag{16}$$

This parameter is somewhat similar to the correctness parameter. A lower value of steadiness will produce a lower correctness. In this case also, the time stamps of the sample measurements are important. This is because the steadiness of a PUF may change when operating conditions change. However, Hori et al. did not discuss the effect of time on the steadiness parameter [8].

Diffuseness

The diffuseness is defined as

$$\frac{1}{L}\frac{4}{K^2}\sum_{l=1}^{L}\sum_{i=1}^{K-1}\sum_{j=i+1}^{K}(r_{n,i,l}\oplus r_{n,j,l}). \tag{17}$$

One question arises regarding the diffuseness parameter. Since the diffuseness is estimated among $K$ signatures ($L$ bits each) generated in a chip, its value might change depending on how we create a group of bits to produce an ID. For example, there are a total of $K \times L$ binary bits in $K$ $L$-bit signatures. These $K \times L$ bits can be divided in many possible $K$ groups with $L$ bits each. Therefore, the same set of $K \times L$ bits can lead to different values of diffuseness based on the combination we select. Since the PUF challenges are selected by a software program [8], the diffuseness can be controlled deterministically.

Bit-Aliasing

The bit-aliasing parameter is defined as

$$(\mathrm{Bit-aliasing})_{k,l} = \frac{1}{N}\sum_{n=1}^{N} r_{n,k,l}. \tag{18}$$

(This parameter has been defined once in Sect. 3.2. We express it again here in terms of the common notations that have been introduced in the beginning of this section.)

### 3.3.5 Probability of Misidentification

Here, we introduce another parameter, PMSID, defined by Su et al. [19]. It is a useful parameter to estimate the probability of a chip being falsely identified as another chip due to noise in the response bits.

Suppose a chip $X$ has a reference PUF identifier $R_X$ with $L$ bits. At some other point in time, it produces an identifier $R'_X$. Therefore, the number of unreliable bits in that PUF is $\mathrm{HD}(R_X, R'_X)$. Now, if there exists another chip $Y$ such that $\mathrm{HD}(R_Y, R'_X) \leq \mathrm{HD}(R_X, R'_X)$, there will be a misidentification. For a PUF identifier with $L$ response

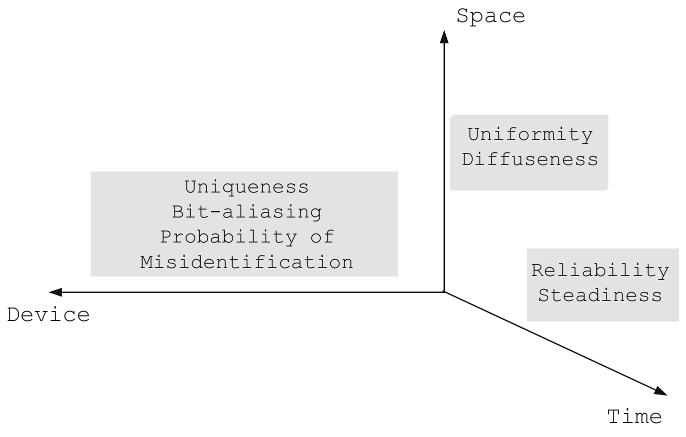


**Fig. 8** Final parameters mapped on the PUF measurement dimension

bits, if $p$ is the fraction of the unreliable bits (fractional intra-chip HD) and $h$ is the value of HD between the $L$-bit identifier of the chip and that of another chip ($h \leq L$), the PMSID is defined as [19]

$$\sum_{h=0}^{L} \left[ \binom{L}{h} 0.5^h (1-0.5)^{L-h} . \sum_{h/2}^{h} \binom{h}{h/2} p^{h/2} (1-p)^{h-h/2} \right]. \tag{19}$$

## 3.4 Final Set of PUF Parameters

We have discussed several parameters that quantify the quality/performance of a PUF. As a conclusion, we suggest a set of parameters with some modifications as the components of the PUF evaluation–comparison method. It includes seven parameters: uniformity, reliability, steadiness, uniqueness, diffuseness, bit-aliasing, and PMSID. Basically, we excluded the redundant parameters while analyzing the parameters proposed by Hori et al., Su et al., and us. This set of parameters will serve as the starting point of the evaluation–comparison method. However, we believe that this set is subject to further modification and enhancement. Figure 8 shows these seven parameters mapped along the proposed PUF measurement dimensions. We briefly discuss why we included these parameters as the components of the evaluation–comparison method.

1. *Uniformity* As the analysis shows, uniformity does not include samples of response bits unlike randomness does. If samples are included, it becomes dependent on time. Since we want to evaluate the ratio of ‘1’s and ‘0’s on an average, i.e., based on the reference response bits, uniformity is a good fit.

Moreover, to estimate the time dependency of a PUF response, we have other parameters such as reliability and steadiness.

2. *Reliability* Our analysis showed that both the correctness and the reliability are defined in a similar way. However, the time reference is not defined well in case of the correctness. Moreover, the inequality (12) that determines the normalization factor for the correctness has been shown to be not valid always. Hence, we propose to include the reliability parameter defined in Eq. (10).
3. *Steadiness* Even though the steadiness parameters seem to be closely related to the reliability/correctness parameter, it represents the bias of individual response bits on an average. Therefore, we suggest to include this parameter as well. However, this parameter needs to be defined based on time stamps, i.e., a chip may have different steadiness values depending on the time when it is measured.
4. *Uniqueness* For the uniqueness parameter, both Hori et al. and us employ average inter-chip HD of the PUF identifier except the normalization factor. The normalization factor used by Hori et al. represents the upper bound of the SHDs among all possible IDs which is a useful information about a PUF. Hence, we suggest to include the uniqueness defined by Hori et al.
5. *Diffuseness* The diffuseness, as discussed earlier, is very similar to the uniqueness. This parameter becomes useful when a PUF has a large CRP space like APUF, and several identifiers can be produced from a single chip. Therefore, we suggest to use the diffuseness parameter when the PUF has a large CRP space.
6. *Bit-aliasing* The bit-aliasing parameter is very useful in estimating the bias of a particular response bit across several chips. It may also give us information about any systematic, spatial effect across devices.
7. *Probability of misidentification* Finally, we also propose to include the PMSID to estimate the rate of error in identification by a PUF. This parameter shows how chip identification may be affected by noise in the response bits.

## 4 Comparison of the RO PUF with the Arbiter PUF: A Test Case

In this section, we compare the RO PUF with the APUF using the parameters defined by Hori et al. and us as well as the PMSID. We used two datasets for this purpose: (a) one dataset consists of RO PUF responses from 193 FPGAs measured by us and (b) the other dataset consists of APUF responses from 45 FPGAs measured by Hori et al. First, we briefly describe the two PUFs: the RO PUF and the APUF.

An RO PUF has $m$ identically laid-out ROs and was proposed by Suh et al. [20]. A pair of frequencies, $f_a$ and $f_b$ $(a \neq b)$, out of $m$ RO outputs are selected as a challenge. Due to random process variation, $f_a$ and $f_b$ tend to differ from each other randomly. A response bit $r_{ab}$ is produced by a simple comparison method:

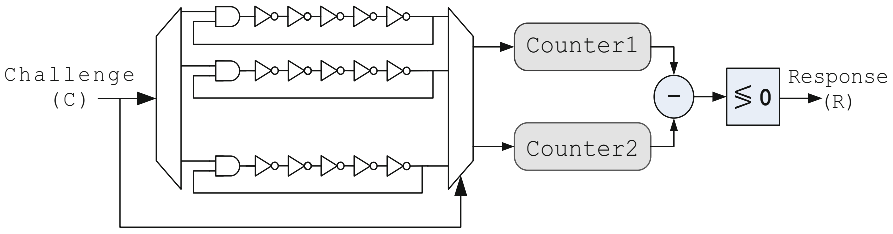


**Fig. 9** Ring-oscillator-based PUF

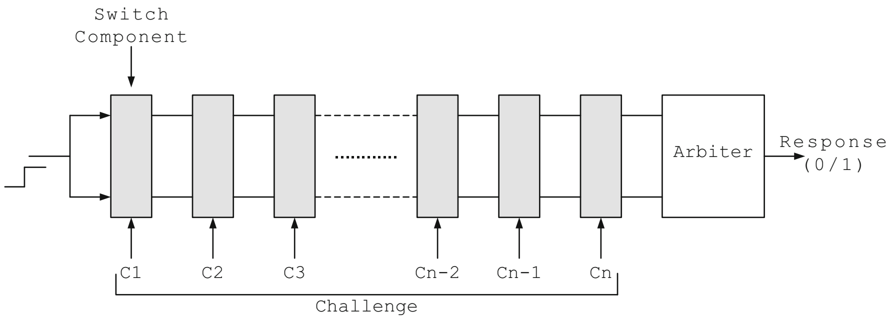


**Fig. 10** Arbiter PUF

$$r_{ab} = \begin{cases} 1 & \text{if } f_a > f_b \\ 0 & \text{otherwise} \end{cases} . \tag{20}$$

Since the variability in frequency is random, the response bits produce random binary values. Figure 9 shows a ring-oscillator-based PUF (RO PUF).

On the other hand, an APUF, proposed by Lim et al., exploits the delay mismatch between a pair of identically routed paths to generate a response bit [11]. Depending on which of the delay path is faster, the Arbiter flip-flop produces a '0' or '1' as a PUF response. Due to random variation in delay paths, this response bit is random. Several pairs of delay paths can be configured by setting the inputs of the switch components (shown in Fig. 10) used as challenge inputs.

Table 3 describes both the datasets that have been used for the analysis. It also includes the device technology used for the respective FPGA implementations. We measured the RO PUF at normal operating conditions. Since Hori et al. did not mention any variation in the operating condition during their measurement, we assume the APUF dataset is also measured under normal operating condition.

We first evaluate the parameters defined by Hori et al. using the RO PUF dataset and compare them with the results from APUF reported in [8]. Since the RO PUF we implemented produces only *one* 511-bit identifier ($K$=1), the diffuseness is not calculated for the RO PUF dataset. After that, we evaluate the parameters defined by

**Table 3** Detail of the datasets used

| | APUF | RO PUF |
|---|---|---|
| N | 45 | 193 |
| T | 1024 | 100 |
| K | 1024 | 1 |
| L | 128 | 511 |
| M | – | 512 |
| Device technology | 65 nm (Virtex 5) | 90 nm (Spartan 3E) |

**Table 4** Comparison of RO PUF and APUF using parameters defined by Hori et al.

| | APUF (%) | RO PUF (%) | Ideal value (%) |
|---|---|---|---|
| Average randomness | 84.69 | 96.81 | 100 |
| Average probability | 55.61 | 49.82 | 50 |
| Average steadiness | 98.48 | 98.51 | 100 |
| Average correctness | 98.28 | 98.29 | 100 |
| Average uniqueness | 36.75 | 94.07 | 100 |

us using the APUF dataset and compare them with the results based on the RO PUF dataset. We also compare both the datasets using the PMSID. Finally, we summarize the comparison based on the set of seven parameters we selected.

## 4.1 Comparison Using Parameters Defined by Hori et al.

Table 4 shows the parameters defined by Hori et al. evaluated on both the datasets. The parameter values based on the APUF dataset have been taken from [8]. It can be noticed that the RO PUF shows better randomness compared to the APUF. This is supported by the fact that the average bit probability of RO PUF is close to 0.5, i.e., the RO PUF responses are more equally likely between '0' and '1'. Normally it is expected that a Virtex 5 FPGA on a 65-nm technology will have more variability (hence more randomness in PUF responses) than a Spartan 3E FPGA on 90-nm technology. However, this result shows that the RO PUF has better randomness than the APUF. This indicates that individual PUF technique may have significant influence on extracting the variability information. Another significant difference can be observed in the value of the uniqueness. The uniqueness of the RO PUF is distinctly much higher than that of the APUF. Lower value of the randomness in APUF is one of the reasons why the uniqueness is lower in it. In the next section, we will evaluate the bit-aliasing parameter that may explain this contrast in the uniqueness more. Apart from that, both the PUFs show similar values of the steadiness and the correctness. It implies that the two PUFs are in general highly tolerant to noise at normal operating condition.

Table 5 shows the confidence interval (CI) proposed by Hori et al. for a confidence level of 95 % for both the datasets. It can be noticed that the RO

**Table 5** Confidence interval comparison results with 95 % confidence level

| | APUF | | RO PUF | |
|---|---|---|---|---|
| | Confidence interval | Width | Confidence interval | Width |
| Randomness | [0.8388, 0.8546] | 0.01586 | [0.9892, 0.9990] | 0.00986 |
| Bit probability | [0.5530, 0.5591] | 0.00611 | [0.4962, 0.5003] | 0.00407 |
| Steadiness | [0.9626, 1.0000] | 0.04134 | [0.9846, 0.9857] | 0.00110 |
| Correctness | [0.9579, 1.0000] | 0.04206 | [0.9822, 0.9834] | 0.00121 |
| Uniqueness | [0.2127, 0.5222] | 0.30950 | [0.9334, 0.9481] | 0.02940 |

**Table 6** Comparison of RO PUF and APUF using parameters defined by Maiti et al.

| | APUF (%) | RO PUF (%) | Ideal value (%) |
|---|---|---|---|
| Uniformity | 55.69 | 50.56 | 50 |
| Bit-aliasing | 19.57 | 50.56 | 50 |
| Uniqueness | 7.20 | 47.24 | 50 |
| Reliability | 99.76 | 99.14 | 100 |

PUF dataset shows significantly narrower CI compared to the APUF dataset. This indicates that the size of the PUF dataset (RO PUF having a larger dataset compared to APUF) has substantial impact on determining the confidence interval of the parameters.

## 4.2 Comparison Using Parameters Defined by Maiti et al.

Table 6 shows the average values of the parameters defined by us for both the datasets. We considered 511 response bits for the RO PUF, whereas $1024 \times 128 = 131{,}072$ response bits were considered for the APUF.

The uniformity result resembles with the average probability reported in Table 4. For the bit-aliasing, the average in the RO PUF is close to the ideal value of 50 % while the average in the APUF deviates significantly from 50 %. Moreover, we found that the minimum value of bit-aliasing is 0 % in case of APUF. This shows that there are bit positions that produce a value of 0 for all the 45 chips. In fact, we found 21,314 out of 131,072 bit positions (nearly 16 %) produced a value of 0 %. These bits do not contain any useful information. This is consistent with a very low value of the uniqueness in APUF reported in Table 4. One reason for this may be the difficulty in ensuring routing symmetry in an APUF implemented in an FPGA [17]. The sharp difference in the value of the uniqueness is visible in this case also. The reliability values are comparable for both the datasets indicating that both the RO PUF and the APUF are equally reliable.

**Table 7** Comparison of probability of misidentification

| | Minimum | Maximum | Average |
|---|---|---|---|
| RO PUF | $2.81 \times 10^{-71}$ | $4.42 \times 10^{-39}$ | $1.18 \times 10^{-40}$ |
| APUF | $3.03 \times 10^{-13}$ | $3.91 \times 10^{-12}$ | $1.50 \times 10^{-12}$ |

**Table 8** Summary of comparison between the RO PUF and the APUF

| | APUF (%) | RO PUF (%) | Ideal value (%) |
|---|---|---|---|
| Uniformity | 55.69 | 50.56 | 50 |
| Reliability | 99.76 | 99.14 | 100 |
| Steadiness | 98.48 | 98.51 | 100 |
| Uniqueness | 36.75 | 94.07 | 100 |
| Diffuseness | 98.39 | – | 100 |
| Bit-aliasing | 19.57 | 50.56 | 50 |
| PMSID | $1.50 \times 10^{-12}$ | $1.18 \times 10^{-40}$ | 0 |

### 4.3 Comparison Using the Probability of Misidentification

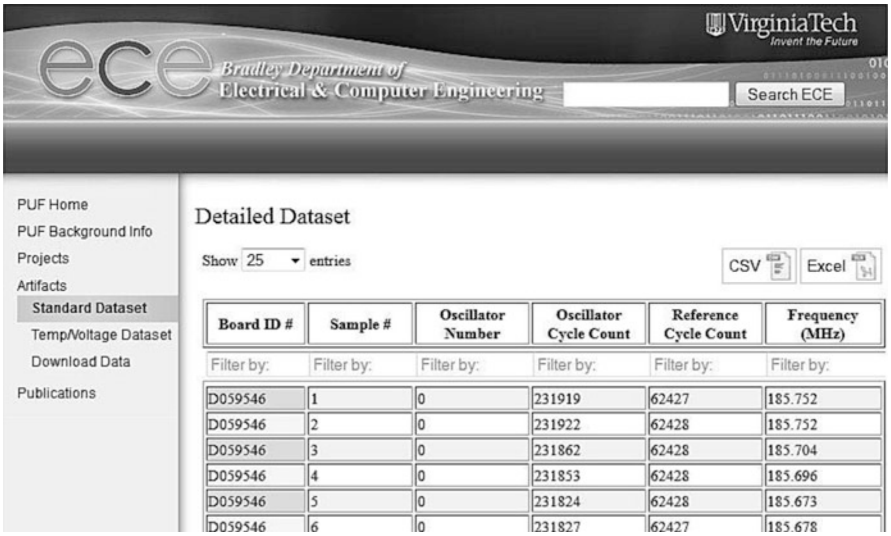
Table 7 shows the value of PMSID for the two datasets. The RO PUF dataset shows a much lower value compared to the APUF dataset. However, this is due to the fact that the length of the identifier for the RO PUF is 511, whereas it is 128 for the APUF. In any case, even the Arbiter PUF shows a very low PMSID. This is consistent with the fact that both the PUFs showed a very high value of reliability indicating that the proportion of the noisy bits in both the PUFs is low.

### 4.4 Summary of Comparison Between the RO PUF and the APUF

Table 8 shows the summary of the comparison between the APUF and the RO PUF in terms of the seven parameters we selected as part of the proposed evaluation–comparison method. The two PUFs exhibit comparable performance in terms of the uniformity, the reliability, and the steadiness. However, the RO PUF shows much better performance compared to the APUF in terms of the uniqueness, the bit-aliasing, and the PMSID. The diffuseness parameter could not be evaluated for the RO PUF.

## 5 Online Database

In our research, we frequently observed the need for larger datasets for use in the PUF research community. Indeed, the large-scale characteristics of a PUF circuit can only be studied indirectly, through the dataset of a large population. Earlier, we have published our measurements as a batch download [15].



Detailed Dataset

| Board ID # | Sample # | Oscillator Number | Oscillator Cycle Count | Reference Cycle Count | Frequency (MHz) |
|---|---|---|---|---|---|
| Filter by: | Filter by: | Filter by: | Filter by: | Filter by: | Filter by: |
| D059546 | 1 | 0 | 231919 | 62427 | 185.752 |
| D059546 | 2 | 0 | 231922 | 62428 | 185.752 |
| D059546 | 3 | 0 | 231862 | 62428 | 185.704 |
| D059546 | 4 | 0 | 231853 | 62428 | 185.696 |
| D059546 | 5 | 0 | 231824 | 62428 | 185.673 |
| D059546 | 6 | 0 | 231827 | 62427 | 185.678 |

**Fig. 11** Web interface to access the PUF database

These measurements are for an RO PUF with 512 ring oscillators configured in a Spartan 3E FPGA. For each ring oscillator, we have taken 100 successive measurements. As each ring oscillator measurements maps into a single record, the database of 193 FPGAs holds almost ten million records. The use of a reconfigurable device such as the Spartan 3E FPGA made this large collection of data significantly easier in terms of shorter time as well as lower cost of on-chip PUF implementation compared to a custom-designed circuit.

In order to provide a more fine-grained access to the data, we ported the dataset to a MySQL database and developed a website interface to access it. The website can be reached through www.rijndael.ece.vt.edu/puf. Figure 11 illustrates our web interface. Each ring oscillator frequency is stored as a tuple of two integers, one representing a cycle count for the ring oscillator and a second representing the cycle count for a 50 MHz reference frequency. The ratio of these two integers captures the ring oscillator frequency.

Each record in the database contains the board (i.e., FPGA chip) serial number, the oscillator number with the FPGA, the measurement number, and the two oscillator counters that represent the ring oscillation frequency. Specific records in the table can be queried through the search fields on top of the table. For example, it is possible to quickly extract the frequency of the 25th ring oscillator across all boards and across all measurements, by filling out '25' in the oscillator number search field.

We plan to further extend the database to accommodate other types of PUFs as well as other types of PUF parameters, such as area and performance.

## 6 Conclusions

In this work, we aimed at defining a method to evaluate as well as compare the performance of several PUFs irrespective of the underlying PUF techniques. Our approach relies on the statistical properties of the binary response bits of a PUF. We first proposed three dimensions of PUF measurement. Based on our analysis on parameters defined by us as well as by others, we proposed a set of seven PUF parameters as the primary building block of the evaluation–comparison method. Subsequently, we compared two different PUFs, namely the RO PUF and the APUF, using these parameters based on measured PUF data. The RO PUF shows better performance than the APUF in terms of the uniqueness, the bit-aliasing, and the PMSID while other parameters yielded comparable results from both the PUFs.

**Acknowledgements** This work was supported by the National Science Foundation by grant no. 0964680 and grant no. 0855095. A special credit goes to Michael Cantrell for his contribution in developing the PUF database as well as the web interface to access it.

## References

1. Armknecht F, Maes R, Sadeghi A-R, Standaert F-X, Wachsmann C (2011) A formal foundation for the security features of physical functions. IEEE Security and Privacy 2011(1):16
2. Bolotnyy L, Robins G (2007) Physically unclonable function-based security and privacy in rfid systems. In: Fifth annual IEEE international conference on pervasive computing and communications, PerCom 2007, pp 211–220, March 2007
3. Devadas S, Suh E, Paral S, Sowell R, Ziola T, Khandelwal V (2008) Design and implementation of puf-based "unclonable" rfid ics for anti-counterfeiting and security applications. In: IEEE international conference on RFID 2008, pp 58–64, April 2008
4. Gassend B, Clarke D, van Dijk M, Devadas S (2002) Silicon physical random functions. In: Proceedings of the 9th ACM conference on computer and communications security, CCS 2002. ACM, New York, NY, USA, pp 148–160
5. Guajardo J, Kumar S, Schrijen G-J, Tuyls P (2007) Fpga intrinsic pufs and their use for ip protection. In: Proceedings of the 9th international workshop on cryptographic hardware and embedded systems, CHES 2007. Springer, Berlin, Heidelberg, pp 63–80
6. Guajardo J, Kumar S, Schrijen G-J, Tuyls P (2008) Brand and ip protection with physical unclonable functions. In: IEEE international symposium on circuits and systems, ISCAS 2008, pp 3186–3189, May 2008
7. Helinski R, Acharyya D, Plusquellic J (2009) A physical unclonable function defined using power distribution system equivalent resistance variations. In: Proceedings of the 46th annual design automation conference, DAC. ACM, New York, NY, USA pp 676–681
8. Hori Y, Yoshida T, Katashita T, Satoh A (2010) Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas. In: International conference on reconfigurable computing and FPGAs (ReConFig) 2010, pp 298–303, Dec 2010
9. Krishna AR, Narasimhan S, Wang X, Wang X Mecca: a robust low-overhead puf using embedded memory array. In: Proceedings of the 13th international conference on Cryptographic hardware and embedded systems, CHES 2011. Springer, Berlin, Heidelberg, pp 407–420

10. Kumar S, Guajardo J, Maes R, Schrijen G-J, Tuyls P (2008) Extended abstract: The butterfly puf protecting ip on every fpga. In: IEEE international workshop on Hardware-oriented security and trust, HOST 2008, pp 67–70
11. Lim D, Lee J, Gassend B, Suh G, van Dijk M, Devadas S (2005) Extracting secret keys from integrated circuits. IEEE Trans Very Large Scale Integration Syst 13(10):1200–1205
12. Lofstrom K, Daasch W, Taylor D (2000) Ic identification circuit using device mismatch. In: IEEE international Solid-state circuits conference. Digest of Technical Papers. ISSCC 2000, pp 372–373
13. Maes R, Tuyls P, Verbauwhede I (2008) Intrinsic pufs from flip-flops on reconfigurable devices. In: 3rd Benelux workshop on information and system security (WISSec 2008). Eindhoven, NL, p 17
14. Maes R, Verbauwhede I (2010) Physically unclonable functions: A study on the state of the art andfuture research directions. In: Towards hardware-intrinsic security. Springer, New York
15. Maiti A, Casarona J, McHale L, Schaumont P (2010) A large scale characterization of ro-puf. In: IEEE international symposium on hardware-oriented security and trust (HOST) 2010, pp 94–99
16. Majzoobi M, Koushanfar F, Potkonjak M (2008) Testing techniques for hardware security. In: IEEE international test conference, ITC 2008, pp 1–10
17. Morozov S, Maiti A, Schaumont P (2010) An analysis of delay based puf implementations on fpga. In: Sirisuk P, Morgan F, El-Ghazawi T, Amano H (eds) Reconfigurable computing: architectures, tools and applications of lecture notes in computer science, vol 5992. Springer, Berlin, pp 382–387
18. Pappu RS, Recht B, Taylor J, Gershenfeld N (2002) Physical one-way functions. Science 297:2026–2030
19. Su Y, Holleman J, Otis B (2008) A digital 1.6 pj/bit chip identification circuit using process variations. IEEE J Solid-State Circ 43(1):69–77
20. Suh GE, Devadas S (2007) Physical unclonable functions for device authentication and secret key generation. In: Proceedings of the 44th annual design automation conference, DAC 2007. ACM, New York, NY, USA, pp 9–14
21. Suzuki D, Shimizu K (2010) The glitch puf: a new delay-puf architecture exploiting glitch shapes. In: Proceedings of the 12th international conference on Cryptographic hardware and embedded systems, CHES 2010. Springer, Berlin, Heidelberg, pp 366–382
22. Tuyls P, Schrijen G-J, Škorić B, van Geloven J, Verhaegh N, Wolters R (2006) Read-proof hardware from protective coatings. In: Cryptographic hardware and embedded systems workshop of LNCS, vol 4249. Springer, New York, pp 369–383
23. van der Leest V, Schrijen G-J, Handschuh H, Tuyls P (2010) Hardware intrinsic security from d flip-flops. In: Proceedings of the fifth ACM workshop on Scalable trusted computing, STC 2010. ACM, New York, NY, USA, pp 53–62
24. Yamamoto D, Sakiyama K, Iwamoto M, Ohta K, Ochiai T, Takenaka M, Itoh K Variety enhancement of puf responses based on the locations of random outputting rs latches. http://www.iacr.org/workshops/ches/ches2011/presentations/Session%208/CHES2011_Session8_3.pdf
25. Yamamoto D, Sakiyama K, Iwamoto M, Ohta K, Ochiai T, Takenaka M, Itoh K (2011) Uniqueness enhancement of puf responses based on the locations of random outputting rs latches. In: Proceedings of the 13th international conference on Cryptographic hardware and embedded systems, CHES 2011. Springer, Berlin, Heidelberg, pp 390–406

# Index