Embedded Systems Group
Department of Computer Science
Technical University of Kaiserslautern

Out-of-Order Execution Within Functional Units of the SCAD Architecture

Master Thesis

Frederik Walk

June 6, 2016

Supervisors
Prof. Dr. Klaus Schneider
M. tech. Tripti Jain

Zusammenfassung

Moderne Prozessoren nutzen verschiedene Optimierungstechniken um Befehle in sequentiellen Programmen parallel auszuführen. Da der damit erreichbare Grad an Nebenläufigkeit in herkömmlichen Prozessorarchitekturen jedoch aus verschiedenen Gründen eingeschränkt ist, sind neue Architekturen entwickelt worden, um diese Einschränkungen zu überwinden. Die *Synchronous Control Asynchronous Dataflow* (SCAD) Architektur ist eine von diesen. Sie gibt dem Compiler direkten Zugriff auf alle Recheneinheiten und die Datenpfade dazwischen, und führt Berechnungen entlang des Datenflusses aus. Auf diese Weise wird mehr Nebenläufigkeit zwischen einzelnen Befehlen möglich, allerdings leiden die Recheneinheiten selbst weiterhin unter ähnlichen Problemen wie herkömmliche Registerarchitekturen. Aus diesem Grund stellt diese Arbeit eine Erweiterung für Recheneinheiten der SCAD Architektur vor, welche auf Tomasulos dynamischem Scheduling-Algorithmus basiert und eine Umordnung der Befehlsreihenfolge ermöglicht. Der Algorithmus arbeitet lokal innerhalb der Recheneinheiten und benötigt keine Modifikationen am Compiler, wodurch er sehr flexibel eingesetzt werden kann. Auf Basis mehrerer Implementierungen werden in dieser Arbeit verschiedene Anwendungsszenarien diskutiert und evaluiert. Diese zeigen, dass die Umordnung von Befehlen zu hohen Leistungssteigerungen führen kann, die passende Implementierung mitsamt Parametrisierung allerdings sorgsam ausgewählt werden muss, um zu einem konkreten Prozessor und der gewählten Anwendung zu passen.

Abstract

Modern processors use different optimization techniques to dynamically utilize instruction-level parallelism (ILP) in sequential programs. Because the amount of ILP that can be exposed is however limited for various reasons in common processor architectures, new architectures have been developed, to overcome these limitations. One of them is the Synchronous Control Asynchronous Dataflow (SCAD) architecture, which exposes its processing units and the data paths between them to the compiler and schedules execution along dataflow. While this way higher levels of ILP are possible, processing units on their own still suffer from similar problems like widely used register-based architectures. Therefore, this thesis proposes an addition based on Tomasulo's dynamic scheduling algorithm to processing units of the SCAD architecture, enabling out-of-order execution. The algorithm works local to the processing units and requires no modifications to the compiler, allowing it to be used very flexibly. Based on different implementations, this thesis discusses and evaluates a set of application scenarios. These show, while the out-of-order extension can yield a high performance boost, the correct implementation and its parametrization has to be carefully chosen to match a concrete processor design and application.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Frederik Walk Kaiserslautern, den 6. Juni 2016

Contents

List of Figures

1. Introduction

1.1. Motivation

For many years, most commercially successful high performance processors have been based on the classical von Neumann architecture [35]. Many improvements were added to this architecture since it was introduced. They were made possible by advances in the field of manufacturing techniques, allowing more sophisticated circuitry on the processor. Those improvements mainly concern the utilization of parallelism within programs. Recently, this is being addressed by the use of processors with multiple, mostly independent, processor cores. While those achieve high theoretical performance gains, the actual exploitable parallelism highly depends on the computed problem itself.

For this reason, processor cores are equipped with additional methods to exploit parallelism inherent to sequential programs, so-called instruction-level parallelism (ILP), dynamically. Examples for such methods are *pipelining*, where the execution process is split up into different stages, allowing multiple instructions to be worked on at the same time, and *dynamic scheduling*. The dynamic scheduling technique is used to execute instructions in data dependency order rather than program order.

Those methods however are limited in their usage, mainly due to physical reasons: Chip area, power consumption and clock speed cannot be increased indefinitely. To overcome those limits, new approaches have been introduced to processor design over the years, trying to exploit ILP in different ways. Prominent examples are dataflow architectures [14, 34, 18], executing programs based on data dependencies rather than control flow, and exposed datapath architectures, like TRIPS [11], TILE64 [7], transport triggered architectures [12] and SCAD [9]. Exposed datapath architectures have a large number of processing units and delegate instruction scheduling as well as moving data between them to the compiler.

While exposed datapath architectures circumvent limits of classical architectures by eliminating the explicit use of registers and allow a more dataflow driven execution, utilizing much higher levels of ILP, processing units themselves can again suffer from similar problems like common processor architectures.

Therefore it is obvious to integrate optimization methods used in today's commercial processors into these processing units. Not only can this improve overall performance, but through the dynamic nature of those methods the compiler and existing code do not need any modifications. This further means that each processing unit can be indepen-

dently equipped with optimizations, allowing those to be used more targeted thereby reducing the added chip area.

In this work, a method is presented that adds dynamic scheduling to individual processing units of the SCAD architecture. Also the feasibility of different implementations of this method is explored.

1.2. Contribution and Organization

This thesis discusses problems hindering dataflow driven execution that can arise in the SCAD architecture. It presents a way to address these problems, incorporating dynamic scheduling mechanisms used in traditional processors to the SCAD architecture. Also different ways of implementing such an algorithm with focus on either speed or circuit size are given. Along with this, a first implementation of the in- and output buffers, which are to be extended with out-of-order execution, is created for reference. The focus of both, algorithm and implementation, is to extend the flexibility of the SCAD architecture, by keeping changes local, modular and parametric. By means of the implementation, feasibility of the algorithm and the different approaches is explored on a local level.

The implementation of a full SCAD architecture processor and, since such a processor does not exist at this time, the impact of out-of-order execution on a global scale are not a goal of this work. Further, any optimizations involving the compiler are not in the scope of this thesis, the focus is purely on dynamic execution within the processor.

The remainder of this thesis is organized as follows: In Chapter 2, preliminaries for this thesis are explained. The chapter covers instruction-level parallelism and how it is achieved in other architectures, as well as dataflow process networks as a purely dataflow oriented model of computation. Then, a detailed description of the SCAD architecture is given, to show possible attachment points for the out-of-order algorithm. Finally, the base for the presented algorithm, Tomasulo's dynamic scheduling algorithm, is outlined.

Beginning in Chapter 3, the fundamental problem which the out-of-order algorithm can solve, is described and analyzed. Section 3.2 gives an overview of how and why such an algorithm works, not considering an actual implementation. A rough estimation on different aspects of the algorithm's complexity is done in Section 3.3.

Chapter 4 continues with the actual implementation. First some remarks on the targeted hardware and used tools are given. Implementation approaches for three different in- and output buffer combinations are presented. These are evaluated in Chapter 5. Both, performance and resulting circuit size, are examined by defining testing scenarios and parameters. A discussion of the results follows respectively. Finally, Chapter 6 concludes this thesis.

1.3. Related Work

In [14] Dennis and Misunas propose a processor architecture executing programs written using a dataflow language. Here, nodes of a dataflow graph are stored as so-called instruction cells in memory along with a token for each in- and output of a node. Nodes support a fixed set of functions and are limited to two in- and outputs. Instruction cells can be assigned to an operation unit when enough input tokens are available. Results are then transferred back to the corresponding node defined by the dataflow graph, allowing more nodes to be scheduled. This way, execution can happen out-of-order along the dataflow.

Baudisch describes a way to apply out-of-order execution to dataflow process networks on multiprocessors in [4],[5] and [6] by executing different iterations of the same node, called a task, in parallel on different processor cores based on data dependencies. Since this can lead to out-of-order reads and writes, buffers between nodes are replaced by a central buffer station (CBS) allowing random access. The CBS stores the system state for a sliding window of iterations, while tasks read and write to the iteration they are currently in. Dependencies between tasks are tracked, and tasks are scheduled as soon as all of their dependencies are fulfilled.

Rosière et al. [29] present a FPGA[1]-friendly implementation of reorder buffers usable for out-of-order execution in soft-core processors. It can work using a two-port RAM by switching multiple register banks and reduce the overall use of logic and registers in contrast to classical designs. Similarly, Aasaraai and Moshovos propose a register-renaming scheme, optimized for soft-core designs on FPGA hardware in [2], again being able to use two port RAM modules and flattening the circuit depth to allow higher clock rates. Mesa-Martinez et al. discuss further problems with using existing, mainly ASIC optimized, out-of-order cores on FPGAs at the example of their SCOORE architecture [27], also proposing the usage of banked memory architectures.

[1] Field Programmable Gate Array

2. Preliminaries

2.1. Instruction-Level Parallelism

Instruction-level parallelism (ILP) is a measure for the amount of instructions within a given program that can be executed in parallel by the processor. Since the 1980s, many ways of exposing ILP have been developed and nearly all modern processor architectures benefit from some form of ILP. It can be exposed by compilers, statically deciding which instructions are executed in parallel, e.g., in VLIW architectures [15], or through dynamic techniques within the processor. Such techniques include pipelining, partially overlapping instructions in different stages of execution, and the use of multiple execution units in parallel in so-called superscalar architectures. Those can be further improved by introducing optimizations like dynamic scheduling or register renaming, leading to better utilization of available computation cycles. The general idea behind those methods is to work on programs written for sequential machines and analyze their data dependencies, therefore being a form of parallel processing mostly transparent to the user [28].

```
r1 := m[1024]
r2 := r1 + r2
r3 := 39 * 5
r4 := r1 + r3
```

(a)

```
r1 := m[1024]
r1 := r1 * 39
m[1024] := r1

r1 := m[2048]
r1 := r1 * 5
m[2048] := r1
```

(b)

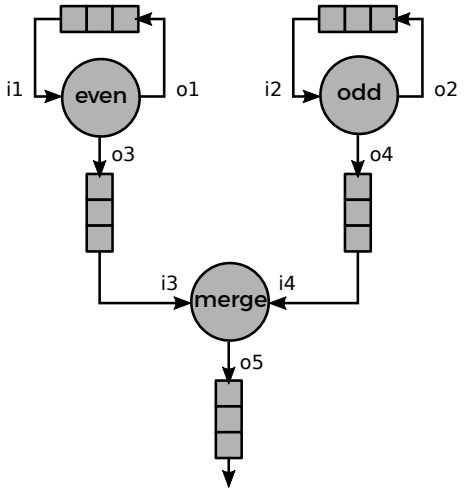
Figure 2.1.: Examples for ILP

Examples for programs where ILP is possible [36] are given in Figure 2.1. In Figure 2.1a, instructions one and three do not depend on each other and therefore can be executed in parallel, while two and four need their results. A dynamic scheduling algorithm detects such cases and can issue the instructions to different execution units or insert them into the pipeline consecutively. The remaining two can be executed as soon as all needed data is available.

The example in Figure 2.1b shows a false dependency. The upper and the lower three instructions have no data dependencies. Even so, they can not be executed in parallel, since both use the same register for intermediate storage. With register renaming, this can be circumvented by using an additional register.

2.2. Dataflow Process Networks

Dataflow process networks are a classical model of computation based on data dependencies. As proposed by Kahn in [25] and formalized further by Lee and Parks in [26], they describe a network of processing nodes that execute sequential functions. Such functions are defined through a set of *firing rules*, which transform input configurations to a set of outputs. Nodes are connected by unbounded first-in-first-out (FIFO) buffers, each buffer having exactly one consuming and one producing node. As soon as a node has enough data available in its input buffers to trigger one of the firing rules, it begins execution. Therefore, each node can work independently and no further synchronization is required, making this model of computation predestined to reach high levels of fine grained parallelism.



(a) graph

merge

i3	i4	o5
(a :: A)	(b :: B)	[a, b]
[ ]	(b :: B)	[b]
(a :: A)	[ ]	[a]

even

i1	o1	o3
initially	[0]	[0]
(i :: A)	[i+2]	[i+2]

odd

i2	o2	o4
initially	[1]	[1]
(i :: A)	[i+2]	[i+2]

(b) firing rules

Figure 2.2.: Example DPN [30]

Figure 2.2 shows an example DPN including firing rules. Nodes *even* and *odd* initially output zero or one respectively to themselves and the *merge* node. In the following, they remove one value from the input and write it to the output, increasing it by two. The *merge* node takes one value out of every input available and then outputs all read values, merging data streams coming from both of the other nodes.

While hardware architectures based on DPNs [18, 34] have been developed, substantial problems arise in real world implementations. Most problematic is the determination of the maximum buffer sizes and their general boundedness. Solutions have been proposed, e.g., the restriction to (cyclo-)static DPNs [10]. However, those restrictions also reduce the amount of computable functions, losing Turing completeness.

2.3. SCAD Architectures

The *Synchronous Control Asynchronous Dataflow* (SCAD) architecture [9, 24] is an *exposed datapath architecture* allowing a larger amount of ILP than common architectures through eliminating the need for a central register file.

Most modern processor architectures, as well as compilers, which try to expose ILP in programs, depend on the amount available registers to store intermediate results and to facilitate register renaming. Those are, however, a limited resource and form an upper bound for the amount of ILP that is possible. Simply enlarging the number of available registers is only possible to a certain degree, mostly for plain technical reasons, e.g., the instruction length of VLIW instructions or the physical wiring of the register file on the chip itself.

Exposed datapath architectures address this problem. They not only give direct access to their functional units to the compiler to schedule instructions to, but also expose the data paths between those units, allowing the compiler to directly move data between them, effectively eliminating the need for the explicit use of registers. Programs for these architectures are therefore not a sequence of different operations, like in classical register based architectures, but rather a series of move instructions, describing the data flow between functional units.

A machine based on the SCAD architecture is built of a set of arbitrary processing units, each consisting of queues for its in- and outputs and an execution unit computing an arbitrary function. Those are called *functional units*. A restriction that applies to the computed function, is that it has to consume and produce a number of values, which is deterministic and does not depend on the input values, therefore being known at compilation time. Otherwise it is impossible to guarantee that there will be a value available for every scheduled move instruction during execution. In this work, only functions which consume and produce exactly one value for every output buffer are considered. A basic overview of such a functional unit is given in Figure 2.3.

Values can be sent from each output queue to each input queue through the *data transport network (DTN)*. The DTN can be an arbitrary network connecting all functional units, ranging from simple or hierarchical buses to blocking or non-blocking permutation networks, like Banyan [17] or Beneš networks [8].

In- and output queues store pairs of addresses and values. In the inputs, the addresses describe the origin of a value, in the outputs they describe the target. In the following,

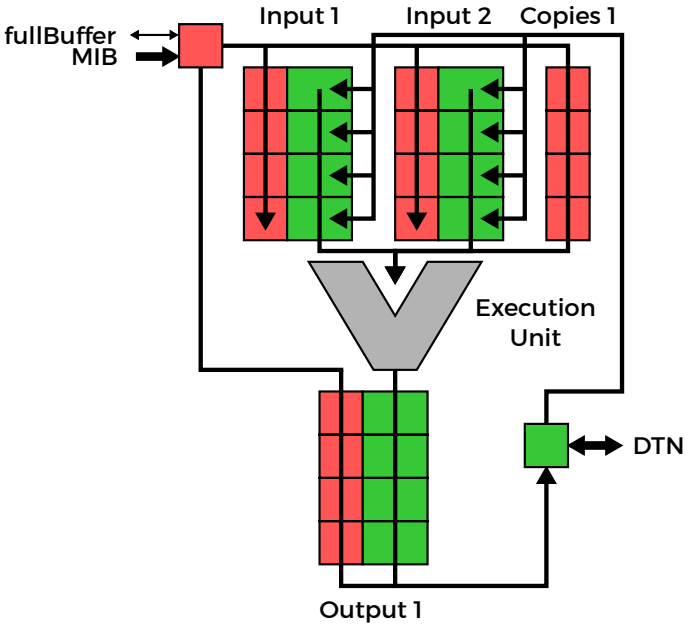


Figure 2.3.: General construction and data-paths of a SCAD functional unit

addresses and values which are not yet available, are denoted by $\perp$. Those queues behave like FIFO buffers for address writes.

SCAD programs are executed in cycles as follows: Within each cycle, the control unit fetches the next instruction, which is always a move instruction consisting of a source and target address (src and tgt). Instructions are broadcast via the *move instruction bus (MIB)* to all functional units of the processor. Both addressed buffers now try to write a new entry at their tail. The input buffer, addressed by tgt, adds an entry $(src, \perp)$, the output buffer src adds $(tgt, \perp)$ or completes an entry to (tgt, val), if there is already $(\perp, val)$ at the tail of the buffer. If one of the buffers src or tgt is full, it signals *fullBuffer* back to the control unit and both writes are aborted. The control unit then tries to write the same move instruction again in the next cycle.

This way, move instructions are effectively scheduled to a later point in time, when the value is available. Synchronous writing of instructions ensures the correct execution of the program by maintaining the correct order of data movements on one hand, and on the other, ensures each transfer through the DTN will be possible later on, since the corresponding space is already allocated in the target buffer.

Further, a move instruction can contain *copy counts*, which are also written by the control unit, and are stored in separate buffers. Copy counts define how often a result will be written to an output buffer. This prevents additional overhead, if a value is needed multiple times, by removing the need for separate copy units or to compute the value multiple times. It also does not collide with the restriction of functions only being al-

lowed to produce one value, since the amount of copies just duplicates a single produced value and is also known at compilation time.

Computing and transferring values between functional units can now happen asynchronously and based on dataflow and is solely done by the functional units themselves. If a functional unit finds values $\neq \perp$ at the head of all of its input buffers it *fires*. This means, it takes those values out of the queue, issues them to the execution unit and writes the results to the output buffers. A value is always written to the entry $(tgt, \perp)$ closest to the head, replacing the $\perp$. If there is no such entry, a new entry $(\perp, val)$ is inserted, the address is then replaced at a later point in time. If there is no space left in one of the functional unit's output buffers, no values are written and the execution unit stalls until there is space again. When a functional units finds a complete entry (tgt, val) at the head of one of its output buffers, it sends a message (src, tgt, val) through the DTN. The input buffer with address tgt now inserts val at the entry $(tgt, \perp)$ closest to its head. Note that there is always such an entry, since all transfers have been scheduled beforehand.

Mind that even if this way of execution looks similar to DPNs, SCAD architectures are only loosely related to them. While functional units fire as soon as data is available at the head of the input buffers, they are not connected point-to-point. Rather, the control flow describes the path of data through the network. Further, execution is not exclusively based on dataflow, since the control flow still exists in form of the address order within functional units.

For memory access, SCAD architecture processors need dedicated functional units, the *load* and *store units*. Those mostly behave like any other functional unit. The store unit has two input buffers for values and the memory addresses values are stored to, the load unit has an input buffer for memory addresses and an output buffer for the loaded values. The difference is, they either have to be synchronized among each other or use a weak memory model in order to guarantee memory consistency.

Branching also makes use of the SCAD architecture's execution scheme. The control unit has an input buffer, which writes its program counter. When writing an instruction to the MIB, the control unit inserts a complete entry with the next program counter into its input queue. However, if the target of an instruction is the control unit's input queue, it just inserts the source address like every other buffer. If such an incomplete entry reaches the head of the buffer, the control unit will stall until the program counter arrives.

The SCAD architecture allows a lot of flexibility for application-specific processors. As mentioned above, the DTN can be chosen freely and different functional units with arbitrary functionality can be combined to optimize the performance for specific calculations.

2.4. Out-of-Order Execution

Out-of-order execution is a method used by many modern processors to improve performance by executing instructions based on their data dependencies rather than the program order. While this is a restricted form of dataflow processing—limited by the window of instructions which can be analyzed at the same time—it allows the efficient use of otherwise idle execution cycles in register-based architectures. This is especially useful for processors with multiple execution units, needing different numbers of cycles to complete their computation, since instructions are dynamically scheduled to those units as soon a they become idle again. This happens without any additional effort from the programmer or the compiler, optimizing parallel processing within sequential programs.

One of the earliest but still most the common way of achieving out-of-order execution on which many modern processors still base their execution scheme, is Tomasulo's dynamic scheduling algorithm [33]. Main advantages of this algorithm which are useful for this work are the high locality, since the building blocks can work mostly autonomously while communicating over a common bus, and the similarity of its components to parts of the SCAD architecture. Even if many improvements have been made to this algorithm over the years, the basic ideas and building blocks are still the same [13].

The following describes a variant of Tomasulo's algorithm, which uses an extension called *reorder buffer* [32] that allows precise interrupts and branch prediction. Although the out-of-order algorithm described later in this work does not make use of those two particular features, the reorder buffer is important in the context of SCAD architectures.

The algorithm extends the processor by the following units: The *reservation station*, a buffer for *forward references*, the *reorder buffer* and the *common data bus*.

Execution now works as follows: An instruction is fetched normally in the instruction decode (ID) stage and a new entry is inserted into the reservation station. If there is no unused space in the reservation station, the ID stage stalls. Each entry has a unique index $st_i > 0$ and stores the op-code, a potential direct operand and, for each other operand, either its value or the index of the entry which produces the value. To determine the operands, the ID stage looks at the buffer containing the forward references. If it finds the entry *zero* as reference to the register the operand should come from, it reads the value from this register directly and writes it to the reservation station. Otherwise, it writes the forward reference stored in the buffer. Further, the forward reference buffer is updated with the entry's index for the register the instruction will write its result to. This way it always contains the index of the newest instruction that will produce a result for this register and, together with fetching operands in-order during ID stage, prevents RAW, WAR and WAW[1] conflicts.

[1] read after write, write after read and write after write

The ID stage also creates an entry in the reorder buffer which contains the associated reservation station index, the target register for the result, and space for the result values. It also contains further information needed for branch prediction. The reorder buffer works like a FIFO queue: New entries are inserted at its tail and, if a entry with a valid value reaches its head, this value is written to the given register and the forward reference for this register is removed. This way register writes are done in-order, leaving the register file in a state that is consistent with the program order if an interrupt happens. Also, no data is written to the register before it is clear whether a branch prediction was correct.

If an entry contains values for all its operands and a suitable execution unit is idle, it is issued to this unit. The execution unit computes the result and sends a tuple $(st_i, value)$ through the common data bus. All other units snoop this bus. The reservation station fills all forward references with the value, allowing new entries to become executable. The reorder buffer also fills all forward references and handles possibly mispredicted branches. No actions are taken by the forward reference buffer and register file, since all register updates are handled by the reorder buffer.

3. Algorithm

3.1. Problem Analysis

The SCAD architecture improves ILP globally by eliminating explicit usage of registers and by dataflow-based scheduling of the firings of individual functional units units. These units themselves however still suffer from problems similar to those of classical register based architectures, having limited buffers and the execution unit as a shared resource. Unlike execution in the DPN model, functional units are not connected by single point-to-point connections using FIFO buffers. Rather, the path data takes through the processor is predefined by the control flow, i.e. source and target addresses in the buffers of functional units. Because of the predefined control flow, values cannot always be consumed as soon as they are available, like in a DPN node.

Considering a *data stream* being an ordered flow of values from a specific output to a specific input buffer, having only one stream arriving or going out of each buffer, execution would look much like that of a DPN. However, data streams can be interleaved through control flow, establishing multiple connections between buffers. Each time a value arrives at an input buffer, it is inserted at the location predefined by the control flow and subsequent results are also sent through the DTN in control flow order. This can lead to cases where independent data streams have to wait for values of others to arrive, delaying execution, especially in cases where data streams have very dissimilar data rates. While correct execution of the program can be guaranteed through the predefined execution order, in many cases, it is not necessary to stick to the control flow given in the buffers to keep this correctness. Such an example is given in Figure 3.1.

Here, all values which are required to execute the third entry in the input buffers are available, so they can be processed by the execution unit. Results are then inserted at the correct position in the output buffer. Further, those results can be directly sent through the DTN, since the control flow defines no values to be sent to address y prior to these results.

Whereas this problem can be reduced by more intelligent scheduling through the compiler, it is very hard to predict the actual timing of dataflow within the processor, together with possible side effects, beforehand. Therefore it is desirable to detect and resolve such problematic cases dynamically during execution, moving execution closer to the purely dataflow-based behavior of a DPN in applicable cases. As such, execution should trigger as soon as values are available in the input buffers, reducing influence of the control flow. In the course of this chapter a dynamic scheduling algorithm is devel-

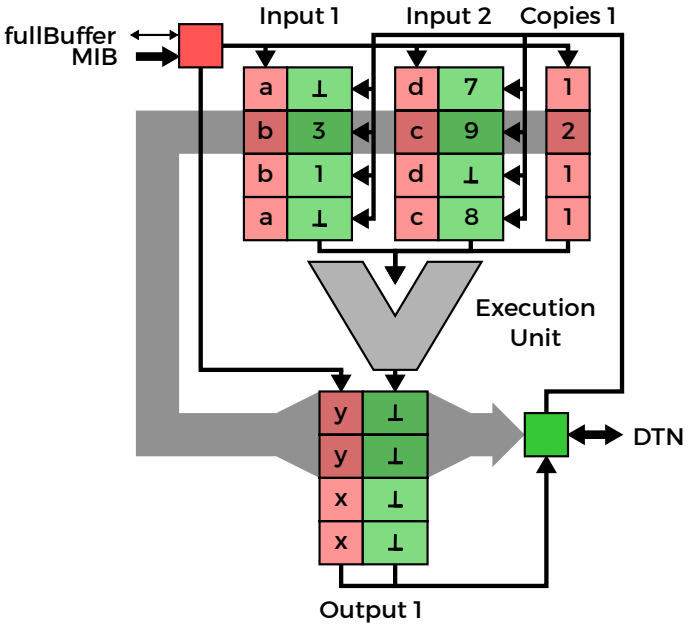


Figure 3.1.: Example for possible out-of-order execution

oped which works local to functional units of a SCAD architecture processor and tries to extract and make use of independent data streams.

3.2. Basic Idea

The main requirement of the proposed algorithm is to restrict visibility of any behavioral changes to single functional units. While it is surely possible to extend out-of-order execution to the scope of the whole processor, predicting the impact of single reorderings needs precise tracking and requires additional synchronization. As correct execution depends on control flow order, confined visibility also ensures correctness of the computation done by the whole processor, as long as functional units still produce the same values.

All data is sent by the output buffers, however, not changing the output order at all is very restrictive to the algorithm and strongly limits its impact to execution performance. But, because only visibility of this order matters, the data stream can be modified. Data always moves from output to input buffers, therefore order is only visible to the input buffers. Streams from different output buffers are, in any case, reordered according to control flow in the input buffers on arrival. In result, only reorderings in data streams coming from one particular output buffer are visible to a specific input buffer. By implication, all values targeting the same input buffer within an output buffer have

to be sent in their control flow defined order, whereas values going to different target addresses can be sent in arbitrary order. As long as this condition is satisfied, execution can happen in any order within a functional unit, without influencing correctness of the overall execution.

To schedule execution in a functional unit, an algorithm based on Tomasulo's dynamic scheduling algorithm is developed in the course of this chapter. Looking at the input buffers, similarities to the reservation station described by Tomasulo become obvious. Although no op-codes and direct operands are stored, because only a single type of instruction exists, essentially a source address together with its position in the input queue defines a reference to the operands, which arrives later in arbitrary order. However, in contrast to the reservation station, in a standard SCAD implementation entries are only taken from the head of the queue.

The next step is to simply allow taking out values from any location in the input queue. There are two enabling factors: First, if an entry with a specific source address contains a valid value, all entries closer to the head, referencing the same address, also contain a valid value, since incomplete entries are filled starting from the head of the queue. Taking out a valid value does, in result, not influence correctness of any later insertion. And second, value positions are always aligned across all input buffers of a functional unit. This is due to the execution unit reading exactly one value from each input buffer on execution. For example, if the values at the third position of all buffers are read, it is the same set of values that would have been read two execution steps later from the head of the buffer.

Values can now be issued for execution early and out-of-order, but the correct order still has to be restored for the results to match up with their predestined target addresses. To solve this, the output buffer is modified to behave similar to the reorder buffer described in Chapter 2.4. When taking a closer look at the reorder buffer, some problems arise. The queue index in the input buffer cannot be used as tag, since entries move further to the front of the queue as values are read, leading to the possibility of the same tag being used multiple times for different value sets erroneously. Further, it is not feasible to write tags to the output buffer at the time instructions arrive at the inputs. Instructions can demand more than one copy for their results, leading to the possibility that the output buffer has not enough space to store all required tags. This is undesirable, since it can unnecessarily stall the control unit.

To address those problems, a reservation mechanism is introduced, which reserves space needed by results and generates a distinct tag right before execution. This process also resolves another problem. When looking at the buffer configuration in Figure 3.2a, the second position can be executed early, but not all copies will fit into the output buffer and the execution unit has to stall until it can write the remaining two copies. Since all entries in the output buffer target the same address, the first entry has to be sent first and is the only way to create space in the buffer again. However, it can never be completed because the execution unit is still waiting for space, resulting in a deadlock. If a reservation ensures enough space in the output buffers before issuing an execution,

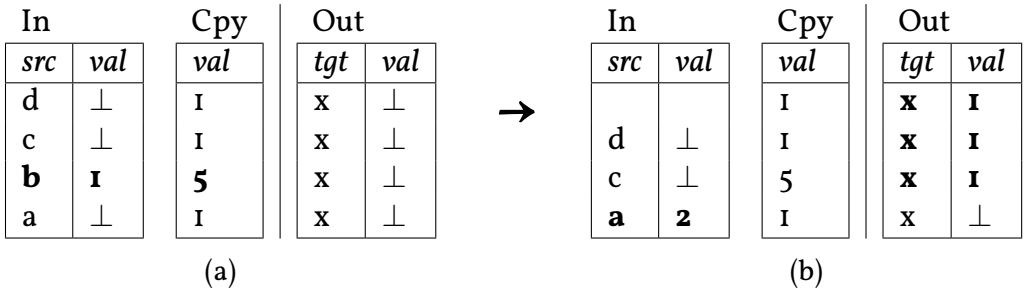


Figure 3.2.: Deadlock scenario for out-of-order execution without reservation

this situation cannot occur. Mind that this is not only a problem for multiple copies, but also for pipelined execution units. Further, there is one exception: Reserving space is not necessary for the head of the input buffer, as there cannot be any incomplete entry closer to the head of the output buffer than its result. This also allows copy counts larger than the output buffer's size. As a last remark on the reservation process, it is left to say that it relies on the property of the execution unit to produce a number of results known before execution starts. Together with the also known copy count, the number of entries to be reserved can be determined.

While the algorithm does not directly interact with other functional units and is agnostic to the computed function, it can not generally be applied to all functional units. If a unit has an internal state or needs to synchronize with other units, e.g., the load and store units, out-of-order execution can change the outcome of program execution. Applicability to such functional units has to be carefully evaluated beforehand.

In the following, additional tasks which have to be performed by each in- and output buffer are explained in more detail.

3.2.1. Input Buffers

Input buffers with out-of-order execution do not differ much from the operation of normal input buffers. Writing addresses and values works the same. The read process, however, is different.

Instead of always reading the head of the buffer, the next value to be read is the one closest to the head, which contains a valid entry in all ports of the functional unit. While other selection criteria are possible, this one is chosen, because on one hand, the result is being inserted closer to the head of the output buffer, yielding a higher chance to be transfered earlier, and on the other hand, it is more likely that there is enough space left at the corresponding position in the output buffer. Further, if there was not enough space for an entry, the search can be stopped, since there also is no space for entries farther along the queue.

To determine if there is enough space in all output queues, a reservation attempt is sent to the output buffer. It contains the current offset from the head of the input queue,

based on the sum of all copy counts of entries closer to the head, along with the copy count of the entry itself. For example, an entry in the *third* position with copy counts of *two* and *three* in the entries beneath, has a final offset of *five*.

On a successful reservation, the output buffer responds with a tag. This tag is passed to the execution unit in addition to values and copy counts and is later used to write the results at the reserved positions. If the reservation is not successful, searching for a set of valid values is stopped for the reasons mentioned above, until a new value arrives at the input buffer.

3.2.2. Output Buffers

While inserting target addresses works the same as in the unmodified buffers, output buffers have two additional tasks considering out-of-order execution. On one hand they must restore the correct order of the results for each target buffer, on the other hand, they have to detect, which results can be sent early through the DTN, i.e. those which are closest to the head of the output buffer for a specific target buffer.

Restoring order works hand in hand with the reservation process. On a reservation request from the input buffer, the output buffer adds the requested offset to an offset between in- and output tracked over time. It is essential to take this additional offset into account, since entries can move independently in buffers on read operations. As reads happen at the same time to all input queues, but at different points in time among output buffers, each output queue has to store its own offset to all input queues as a whole. How this offset tracking works in detail, depends on the concrete implementation of the buffers and is therefore explained in Chapter 4. If the resulting position still lies within the buffer, it is checked if there is enough space for the requested number of copies above this position. This step is omitted if the requested offset is *zero* for the previously explained reasons. The reservation success is then signaled to the input buffer together with a reservation tag. The tag is an arbitrary, currently unused tag from a pool of tags with the size of the buffer. A larger number of tags is not needed, since the number of pending reservations can never exceed the buffer size. Such a tag is also stored in the output buffer along with a reference to the insertion position. Since this position can change during execution, the reference has to be kept up-to-date. How this works in detail is, again, dependent on the actual implementation. As a side note, the reserved space does not need to be explicitly marked as occupied, offset tracking ensures that later reservations will always occupy unused positions.

When a result now arrives at the output buffer, the tag given along with the result by the execution unit is looked up in the list of pending reservations and the value and all following copies are written to the buffer starting at the position referenced by the reservation tag.

In a second, independent process, each output queue on its own, constantly checks, starting at its head, if there is a completed entry available. In a second pass, it is checked

if there is any other entry with the same target address closer the head than the found entry. If this is not the case, the value is sent through the DTN and, depending on implementation, offsets are updated. Otherwise, the next completed entry is checked.

An overview over new data paths and additional extensions for out-of-order execution to functional units is given in Figure 3.3.

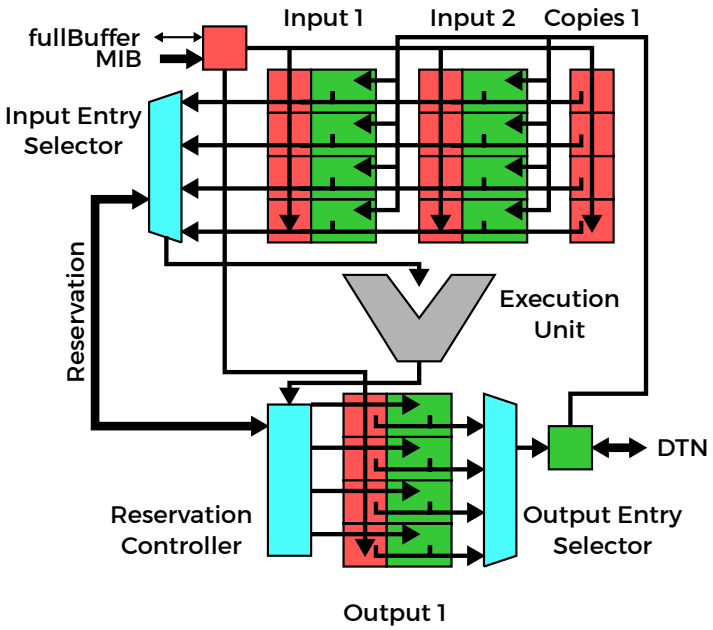


Figure 3.3.: SCAD functional unit with out-of-order extensions

3.3. Complexity

When designing hardware, not only runtime complexity is important, but also required circuit size and especially circuit depth. All those factors depend heavily on the particular hardware design. Ideally they are kept as low as possible, since low circuit depth and size reduce signal delays and therefore allow higher clock rates, while low runtime reduces overall execution cycles needed. Reducing runtime, depth and size at the same time is in many cases not possible, since they are often tied together and lowering one increases the other.

Circuit depth and size are, to a certain degree, optimized by modern synthesis tools. They detect and remove unneeded buffers and signals, and can recognize commonly used patterns, replacing them with more optimal circuits. However, adverse designs can still lead to unnecessarily large circuits, therefore and one has to carefully watch the outcome of synthesis. Runtime, in contrast, completely depends on the design itself.

When looking at the algorithm described in this chapter, it is particularly interesting how these parameters behave according to the buffer size n, because out-of-order execution gets more interesting the larger the buffers get. While most parts of logic are, apart from address widths, independent from the buffer's size, two operations catch the eye: The search for the first valid entry in the input buffer, and the same search together with the check if it the first entry for its address in the output buffer.

Possibilities depend mostly on memory implementation. For memories which allow only one read operation in every cycle, it is obvious that runtime complexity is within $O(n)$, since, in the worst case, all entries have to be read. In the output buffers, where a second pass for the address check needs to be done, runtime complexity is even within $O(n^2)$. Circuit depth on the contrary can be constant because only one comparison is needed in every cycle to check an entry for the property which is searched for.

Having parallel access to all memory entries at once, those operations can be done using a constant number of cycles. In order to find the first entry, for which a specified property holds, in an array needing constant time and using $O(n)$ parallel operations, an algorithm for the PRAM[1] model of computation [16] has been presented in [31]. It uses a technique called *accelerated cascading*, iteratively applying the same algorithm first on a subset of the problem and then again on the result of the first iteration. While the runtime is constant and circuitry can be reused, this method still uses multiple clock cycles. Further problems arise when translating PRAM algorithms to hardware. The PRAM model uses general purpose processing units and implicitly resolves read and write conflicts. Emulating this behavior can add additional circuit complexity. Also, PRAM algorithms reach their peak efficiency for very large problem sizes, the SCAD buffers however are rather small in comparison.

For these reasons, in the case of parallel read access, a tree shaped approach is chosen. This way, search is possible in one cycle using a circuit having a depth within $O(\log n)$, while still keeping overall circuit size comparably small. In the output buffers, the search whether an entry has the first occurrence of an address can be done in parallel for all entries before the final selection of an entry, doubling circuit depth.

The overall circuit size of the output buffers is still expected to be considerably higher than the input buffers, since each port choses the next entries to be sent on its own. Further, using the static runtime approach, each additional entry needs additional search circuitry to find out if it has the first occurrence of its target.

While circuit size appears to be static, for the iterative searches this is not the case. Buffer size still affects address widths and therefore signal widths and size of intermediate storage registers. This also applies to the non-iterative variant.

[1] parallel random-access machine

4. Implementation

To evaluate the algorithm, three different implementations of combinations of in- and output buffers are created. Since, at the time of this work, no complete and also no partial implementation containing the buffers of a SCAD architecture processor exists, a buffer design without out-of-order execution is presented at first. This is the base for later performance and circuit size evaluation. In subsequence, two different approaches, with focus on either circuit size or runtime, are demonstrated.

All implementations were designed using VHDL[1] as hardware description language within the Xilinx Vivado Design Suite[2], targeting generic FPGA devices. The planned evaluation steps need synthesis as well as simulation of the buffers, therefore care is taken to support both. Also, implementations are kept as general as possible, providing parametric adjustment of buffer sizes, data and address widths, port counts and everything else which has to be modified during evaluation. This also makes the buffers easier to adapt to specific SCAD processor implementations.

4.1. Preliminary Considerations

4.1.1. Memory

On modern FPGA devices mainly two different types of memory exist, *block RAM* and *distributed RAM*. Block RAM is implemented using dedicated parts of the FPGA. Each block RAM allows true dual-port access (concurrent reads or writes to two different or equivalent addresses) and handles read and write operations synchronously. They are comparably large in size, with at least 18 KBit on a modern Xilinx Series 7 device, but are usually a rather sparse resource on most devices [20]. The fixed number of ports per unit and the large size make block RAM a fairly inflexible type of memory, mostly suited to handle large amounts of data.

Distributed RAM is implemented using one or more of the FPGA's configurable logic blocks, therefore being distributed across different locations on the FPGA. It allows fine-grained control over the actual size of the RAM, e.g., each block can store only up to 64 bits on a Xilinx series 7 FPGA. Distributed RAM has one write/read port and up to three additional read ports. While write operations are done synchronously, read operations can be asynchronous. One should keep in mind that distributed RAM uses the

[1] http://vhdl.org

[2] http://www.xilinx.com/products/design-tools/vivado.html

same resources as all other logic [21]. This makes distributed RAM more flexible at the cost of a considerable increase of used logic blocks.

Since memories needed for the buffers are expected to be rather small in size and asynchronous read operations make values available sooner, thereby reducing needed clock cycles and storage registers for intermediate results, distributed RAM is chosen as memory for all implementations where memory without parallel access is used.

4.1.2. VHDL-2008

```
architecture Behavioral of reductor is
  signal sin  : std_logic_vector(size-1 downto 0);
  signal sout : std_logic;
begin
  sout <= and sin;
end Behavioral;
```

(a) unary operator

```
architecture Behavioral of reductor is
  signal sin  : std_logic_vector(size-1 downto 0);
  signal sout : std_logic;
  signal help : std_logic_vector(size-1 downto 0);
begin
  help(0) <= sin(0);
  and_gen : for I in 1 to size-1 generate
    help(I) <= help(I-1) and sin(I);
  end generate and_gen;
  sout <= help(size-1);
end Behavioral;
```

(b) generate

```
architecture Behavioral of reductor is
  signal sin  : std_logic_vector(size-1 downto 0);
  signal sout : std_logic;
  constant and_c : std_logic_vector(size-1 downto 0)
                     := (others => '1');
begin
  sout <= '1' when sin = and_c else '0';
end Behavioral;
```

(c) comparison

Figure 4.1.: Different methods for *and* reduction

Tools like Vivado recently began supporting parts [22] of the VHDL-2008 standard [1]. This brings some additions to the language, which especially make development of highly parametric designs, like the SCAD buffers, more straightforward. To be mentioned here, are the improved syntax for `generate` statements, the possibility to reference other generics within the same `generic` list and the support of unary operators for vector reduction.

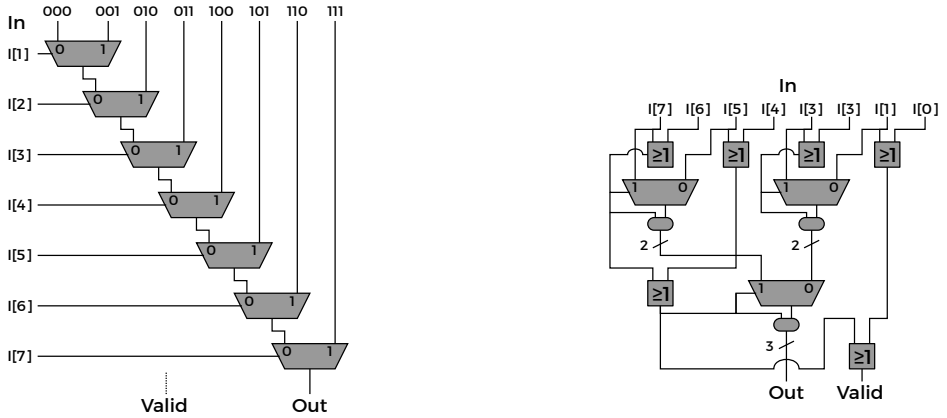
Tests were done to explore the advantages of VHDL-2008. Figure 4.1 shows different methods for an *and* reduction of a logic vector. While using the unary operator provided by VHDL-2008 in Figure 4.1a makes the original intention most clear to the compiler and readers of the code, the other two examples, either generating an iterative structure using a helper signal or simply comparing the input to a constant vector of ones, yield the same hardware circuit after synthesis. Therefore no impact on either performance or circuit size is to be expected by not using VHDL-2008 for such applications.

Still, the more apparent semantics of code using VHDL-2008 constructs for parametric designs make it an ideal candidate for the buffer design. However, VHDL-2008 is not yet fully supported by Vivado. Especially simulation lacks some important features [23]. To avoid mixing both standards and because it is not desirable to have different code bases for synthesis and simulation, the previous VHDL-2002 standard is used for the following implementations.

In the case of vector reduction, the method in Figure 4.1b is used for this work, since it makes the original intention more clear than the one in Figure 4.1c and works for all operators, e.g. *xor*, which is not possible using a simple comparison.

4.1.3. Priority Encoders

As already discussed in Chapter 3.3, the search for the first index in a list fulfilling a certain property can be done using a constant number of computation cycles on a memory allowing parallel access. The circuitry needed is similar to the one of a classical *priority encoder*, which takes a bit vector as input and returns the highest index having a its bit set to high. While circuit depth has already been discussed, [19] shows more advantages of a tree shaped approach over using a sequential chain: Circuit area and compile time are also reduced.



(a) chain (b) tree

Figure 4.2.: Classical priority encoders

Further, standard implementations are given. Those are shown in Figure 4.2. In Figure 4.2a, beginning at the bottom, the highest priority bit switches either its own index or the highest priority index of all other bits to the output. Using the tree shaped circuit in Figure 4.2b, each node adds a new MSB[3] to the index, its value depending on which of its child subtrees contains a set bit at one of its inputs. Note that both variants need an indicator if there is at least one bit set at the input. The chained approach has an additional chain shaped circuit to accomplish this.

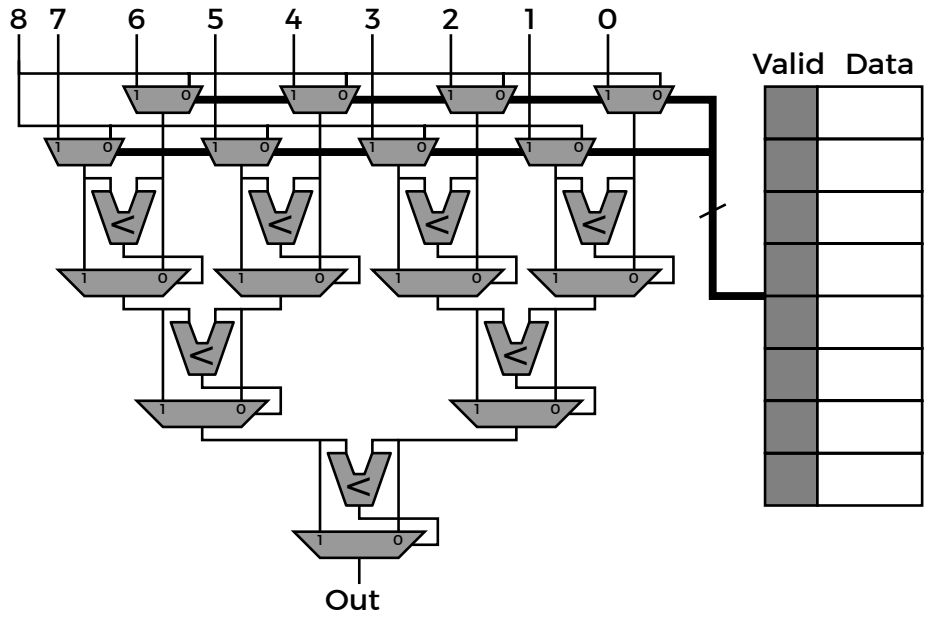


Figure 4.3.: Block diagram of the used search algorithm

In the following, a modified version of the tree shaped approach is used, its block diagram is given in Figure 4.3. Here, the entry having the lowest index has the highest priority. Indices not marked as valid are replaced by a number one higher than the highest existing index. In each node of the tree, the lowest index of both child nodes is routed to the next layer. The resulting output is either the lowest valid index or the number assigned to invalid entries if there is no valid entry available.

This is not an optimal solution considering circuit area, since one additional tree layer is needed and comparators are more complex than multiplexers and *or* gates. Although, this solution is selected for a first implementation, since it makes the intention more clear and inherently solves the problem of checking if there even exists a valid entry. Also its depth complexity is still equal to the one of the tree shaped priority encoder.

[3]most significant bit

```
type tree_connect_array_t is array((2**tree_depth)-3 downto 0) of integer range 0 to size;
signal tree_connect  : tree_connect_array_t;

----

leaves : for J in 0 to size-1 generate
begin
  tree_connect(((2**tree_depth)-3) - J) <= size when entry_valid(J) /= '1' else J;
end generate leaves;

tree_gen : for I in tree_depth-2 downto 1 generate
  level : for J in 0 to (2**I)-1 generate
     tree_connect(((2**(I+1))-3) - J) <= tree_connect((2**(I+2)) - 3 - (2*J)) when
                      tree_connect((2**(I+2)) - 3 - (2*J))
                        < tree_connect((2**(I+2)) - 4 - (2*J)) else
                      tree_connect((2**(I+2)) - 4 - (2*J));
  end generate level;
end generate tree_gen;

current_first <=  tree_connect(0) when tree_connect(0) < tree_connect(1) else
                  tree_connect(1);
```

Figure 4.4.: VHDL implementation of the search circuit

Figure 4.4 shows the corresponding parametric VHDL implementation for the proposed search tree. All intermediate results from the different tree levels are represented in the `tree_connect` signal. The *leaves* generate statement fills the leaves of the tree with indices, inserting `size` for invalid entries. In *tree_gen*, for each level, nodes are compared and intermediate results are written to `tree_connect`. At last, the two values below the root of the tree are compared and the final result is written to the signal `current_first`.

In [3] recursive implementations for such tree shaped circuits are proposed, yet the iterative solution is chosen here because of the low complexity of the single tree nodes, reducing overall code size.

4.2. Architecture and Interface

To allow interchangeability and mixing of different buffer implementations in SCAD architecture processors, the outside interfaces are kept the same across all buffer types. Furthermore, the architecture of the buffers themselves is built to be as modular as possible to keep updates and optimizations of single parts of the buffers simple. Also, buffers share as much of the architecture and modules as possible to achieve better comparability between the different approaches.

All modules are realized as separate VHDL entities. An overview of the fundamental architecture of a functional unit, as well as the buffer units can be seen in Figure 4.5. Grey parts are not directly related to the buffer implementation and are therefore not covered in this chapter.

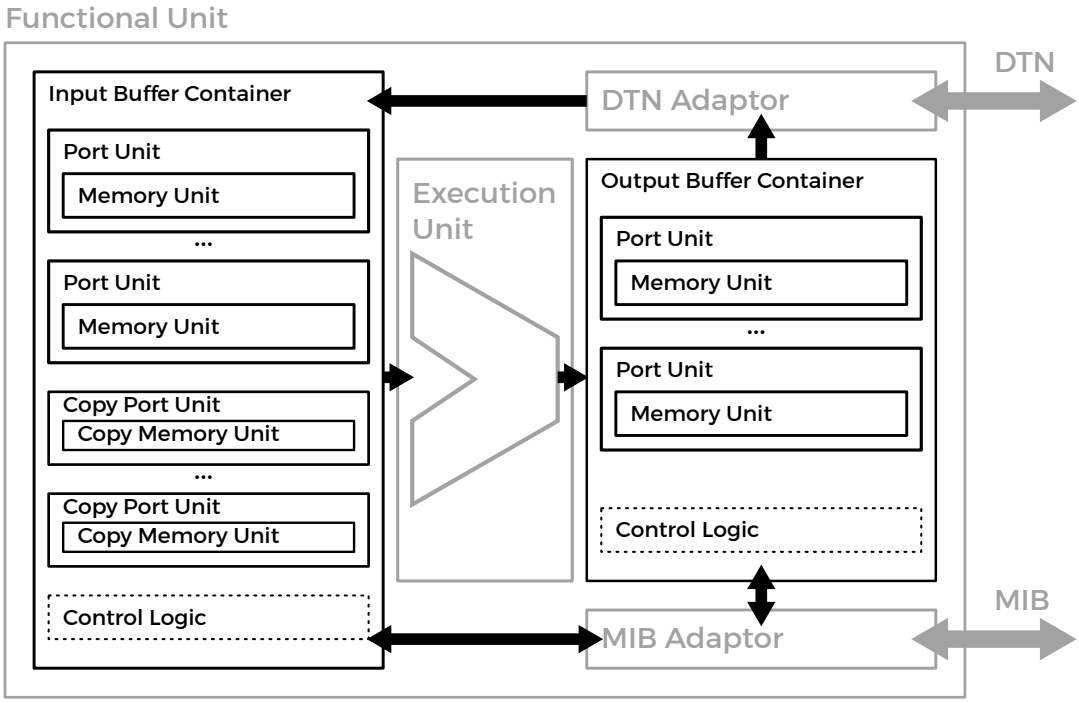


Figure 4.5.: Basic architecture of a functional unit

Buffers are not directly connected to the DTN and MIB, since these can be realized by arbitrary interconnection networks, which are not yet specified. Rather, the connection is abstracted through adaptors using a defined interface and behavior on the buffer side. Buffers are addressed using an arbitrary number of bits, split into base and port address. The base address specifies the buffer container whereas the port address defines the port within the container. These ports are numbered in ascending order. Copy buffers store the number of copies requested for each output buffer. They are supposed to be filled along with writing instructions. However, it is not yet clear how this is happening in detail in a SCAD architecture processor. Therefore, in this first implementation approach, copy counts are written via the DTN, addressing copy buffers like any other port. On the insertion of copy counts, the source address is ignored and the value is written to the tail.

The MIB interface of the buffers has inputs for source and target addresses, as well as a write enable and an apply signal. Also, it can signal a full buffer towards the adaptor. An instruction coming from the MIB is written as follows: The adaptor applies source and target addresses and sets the write enable to high for one clock cycle. After one idle clock cycle, the buffer either signals that it is full or does nothing. The adaptor can then finalize the write attempt after an arbitrary number of cycles if it has seen no other *full* signals from any other buffer on the processor. Otherwise, it can start a new write

attempt. A timing diagram for an instruction write operation needing two attempts is shown in Figure 4.6.

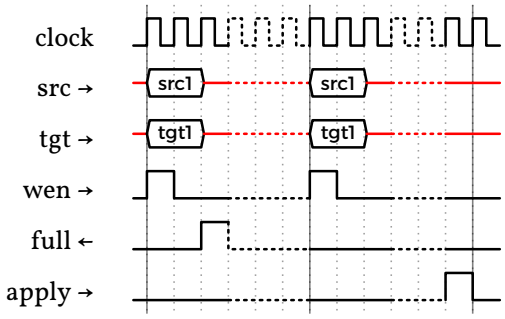
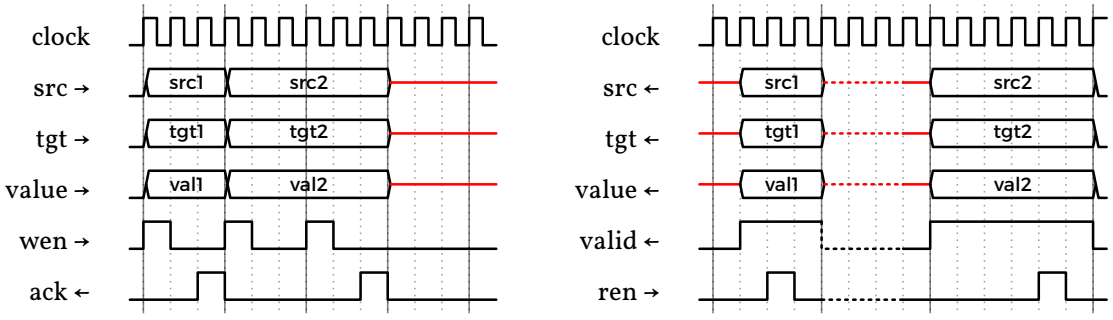


Figure 4.6.: MIB interface timings

Interfaces towards the DTN adaptor look different on the input and output buffers. For the input buffer, after value, source and target addresses are applied by the adaptor, the write enable is set to high for one clock cycle. Two cycles later the input buffer signals an *ack* if the write was successful. If not, the write has to be attempted again. This way, every attempt takes exactly three clock cycles. The diagram in Figure 4.7a shows two successful writes, requiring one and two attempts respectively.

Since the output buffer writes messages to the DTN, it sets addresses and the value itself. It also signals if the currently set message is valid. If the DTN adaptor is ready to send a message, it sets the read enable signal to high for one cycle in order to remove the value from the buffer. After one idle cycle, which is needed to update the valid signal, the next message can be read if available. This can be seen in Figure 4.7b.



(a) input (b) output

Figure 4.7.: DTN interface timings

Out-of-order enabled buffers have an additional interface, directly connecting in- and output containers, to handle the reservation process. As shown in Figure 4.8, to issue a reservation, the input container pulls the request line to high, applying entry positions and sizes for all ports at the same time. After an arbitrary number of clock

cycles, greater than one, the output container either signals an *ack* and a reservation tag if the reservation was successful or a *nack* otherwise. The next reservation attempt can start directly after a response is received.

The interface towards the execution unit is straightforward, providing values, copy counts and a read enable on the input side. On the output, results, a write enable and an *ack* response signal, which is set to high exactly two cycles after a successful write, is available. The out-of-order buffers add a signal indicating the reservation tag and, on the output side, a counter for the current copy.

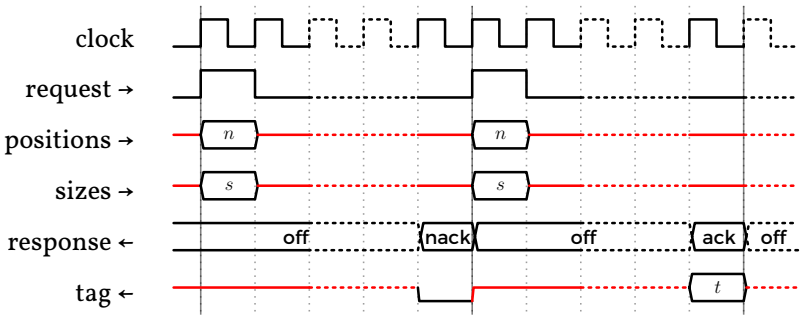


Figure 4.8.: Reservation process timings

Buffers themselves are enclosed by a buffer container, including control logic to handle communication and, if required, further tasks concerning out-of-order execution. Further they contain a number of port units according to amount of ports the buffer provides. Port units contain a memory unit, abstracting access to the physical memory implementation, and organize the entries within memory. Also, the input buffer container has additional copy count port units, one for every output buffer port, each containing its own memory unit. Mind that even if all implementations described in the following are built upon the same architecture, single parts can generally not be interchanged between buffer types, since they have to fulfill special requirements and handle different tasks for each buffer type.

4.3. Buffers Without Out-of-Order Execution

The buffer implementation without out-of-order execution, hereafter called the *simple buffer*, serves as a base for all following implementations and as a reference for evaluation.

All port units use the same memory units. They store addresses and values in two different physical memories, having one read and one write port each. As mentioned before, distributed RAM is used to facilitate asynchronous read operations. Size and memory address width are parametrized and set upon instantiation. Copy count mem-

ory looks similar, except that it only has one physical memory for storing copy counts, therefore providing only one read and write port.

4.3.1. Input Buffers

Input buffers have a port unit for every input port of the functional unit. Port units manage memory and handle value insertion as well as read access. They can be instantiated with arbitrary size, defining the number of contained entries. As an interface, they provide separate inputs for addresses and values, but only an output for values, since addresses are only needed to determine the insertion position for a value. Further status signals for *full*, *empty* and *busy* are provided, *busy* being active while the buffer is handling a value insertion. Lastly, the unit maintains a register of the same width as the buffer size, marking memory entries containing valid values.

Addresses are managed like a FIFO queue, keeping a read and a write pointer and two status bits indicating if the buffer is empty or full. Addresses are written to memory using the write pointer, increasing it, overflowing to zero if necessary. For reading, the read pointer is simply increased. If it reaches the write pointer, the buffer is empty. Respectively, if the write pointer reaches the read pointer during a write operation, the buffer is full. Both pointers start at zero.

Value writes need random access and must be handled differently. The port unit buffers the written value and its source address and sets a third pointer, the search pointer, to the current read pointer. The unit is now busy and, every following cycle, reads the address and the valid register at the current search pointer. If the address matches the buffered address and the entry contains no valid value, the buffered value is written to this position and the busy flag is cleared. By starting at the read pointer, values are always inserted at the matching entry closest to the head of the address queue. Mind that there is always a matching insertion position available, since all value transfers are scheduled beforehand. If there is a valid entry a the read pointer, it is signaled to the container unit. On a read, the valid flag is cleared.

Copy count buffer units completely behave like a FIFO queue, inserting values at the tail and reading from the head, using one read and one write pointer.

Multiple of these buffer units are combined in the input buffer container, which is connected to the DTN and MIB adaptors as well as the execution unit. Container units take buffer size, port count and a base address as parameters.

Messages from the MIB are buffered on a write enable and are forwarded to the port units on an apply signal if the base address matches the container and the port address is within the port range. The *full* signal is generated by outputting the *full* signal of the currently addressed port or, if this port is not within the container, by outputting false.

On a write enable at the DTN interface, it is checked if the target address is valid within the container. Value, source address and write enable are forwarded to the cor-

responding port unit if it is not currently busy. If it is busy, nothing happens and write has to be attempted again.

All of the port's value outputs are connected through to the execution unit. Valid signals from the outputs of the port units are reduced using a logical *and*, since the execution unit needs a complete set of values to compute a result.

4.3.2. Output Buffers

Output buffers do not need to store copy counts and therefore only have normal port units. They have the same address input interface as the input units and handle write operations of target addresses the same way source addresses are handled in the input buffers.

Value storage in the port units on the contrary also works like a FIFO queue. There are separate write pointers, as well as distinct signals for the *full* and *empty* status of value and address memory. The read pointer is shared, since target address and value are always read at the same time. Because of the FIFO behavior, a busy signal and a register to indicate valid values are not needed.

Beside the MIB input, which works the same as in the input container, the output container has to manage messages to the DTN. Since only one message can be sent at a time, value reads have to be distributed over all ports. Therefore, the container cycles through all ports until it finds a valid entry, i.e. value and address buffers are not empty. This entry is then applied to the DTN output, setting its valid signal. When the DTN adaptor reads, the read enable is forwarded to the current port unit and the search is continued at the subsequent port unit having the next higher address or address zero in case of an overflow. This way, a port will be read after a maximum of $port\ count - 1$ DTN transfers and entries are consumed evenly from all ports.

4.4. Buffers With Out-of-Order Execution

In the following the simple buffers described before are extended to support out-of-order execution. These buffers are be referred to as *out-of-order buffers*. The focus for this implementation is to keep the circuit size close to the simple buffers, therefore adding as few extensions as possible. This is expected to keep the performance increase comparably low, while still being applicable to larger buffers. Since out-of-order execution only affects dataflow, everything related to the MIB and address management is left unmodified.

4.4.1. Input Buffers

Input buffer port units use the same memory units as port units in the simple buffers, because no additional read or write operations are needed. Value output is not based on the read pointer anymore, but rather on a search pointer provided from the outside by the container. This pointer is relative to the read pointer: Having value n it is therefore referring the n-th position from the head of the queue. Such a pointer is needed, because search for a valid entry needs to be synchronized across all ports. In addition to the register marking valid values, a second register is added marking entries that have been read. Instead of moving the read pointer, read operations now simply mark entries using this register. However, the read pointer is still needed to manage the buffer's fill status. If the entry at the read pointer is marked as read, the pointer is moved, just as it is on a read in the simple buffer.

The copy count port units also add the external search pointer. Additionally, they signal the number of copies at the read pointer when it is moved. This results in two potential read operations per cycle, one at the read pointer and one at the search pointer. Therefore, the memory unit is modified to expose a second read port.

The container adds an interface for reservations, as well as an output signaling copy counts when the read pointer if moved. The read pointer always moves in all ports at the same time, since entries are aligned and read simultaneously in all ports as mentioned in Chapter 3.

While the data insertion process stays unchanged, the container now searches for a complete set of values in all ports and issues a reservation before handing it to the execution unit. On this account, the search pointer is applied to all port units, starting at zero. It is increased in every cycle until the container sees a *valid* signal from all port units. During this process, copy counts from all copy count units are summed up, resetting to zero if the search pointer overflows. A reservation request is sent to the output buffer, using the sums of copies as positions—because these are the positions the insertion of results has to start in output ports—and the copy counts at the current search pointer. The container now waits for an answer from the output buffer. If a *nack* is replied, search is reset and started again at zero, because if there is no space for the current entry, any entries further up the queue do also not fit. On an *ack*, the set of values is applied to the execution unit interface. When the execution unit reads, the search is started over at zero and the read is propagated to all ports.

4.4.2. Output Buffers

In the output buffers, all tasks concerning out-of-order execution are handled within the port units, since no synchronization between the single ports is needed. While the memory unit of the simple buffers can be reused and writing of addresses stays untouched, writing and reading of values works different.

First of all, the port units keep an offset to the input buffer. This offset describes how many results, which have not yet been read, the entries that have already been removed from the head of the input buffer, have generated in the corresponding output ports. The offset together with the position given during reservation is needed to calculate the correct insertion position in an output port for a result (cf. example in Figure 4.9). Every time the input buffers remove an entry from their tail, the offset is increased by the number of copies in the corresponding copy port. The offset is decreased by one, when an entry is removed from the output port's tail. Mind that the offset is always positive, since the entry which has generated a result at the head of an output queue already must have been removed from the input buffer. The maximum value is either the buffer size or the highest possible copy count, whichever is larger, since the head entry of an input buffer can be issued for execution even if there is not enough space for all results in the outputs.

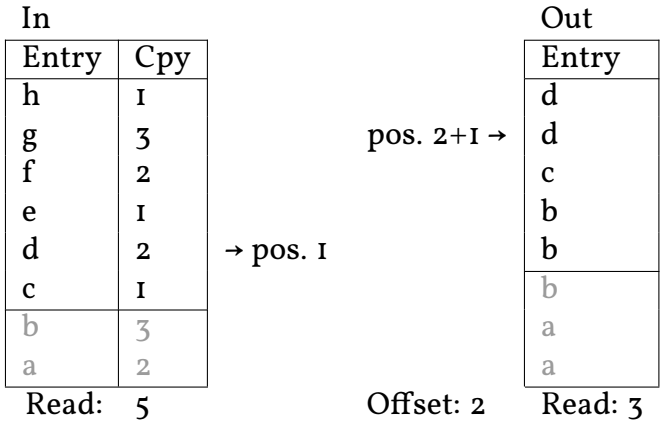


Figure 4.9.: Example on offset calculation and usage

Using the offset, reservations become possible. On a reservation request, an output buffer adds offset, as well as the requested position and copy count, and checks if this value is smaller than the buffer size. If so, there is enough space in the buffer to write the results and the reservation is answered with an *ack*. If the requested position is zero, it is only checked if the offset is smaller than the buffer size for reasons explained in Chapter 3. The reservation cannot be avoided completely in this case, since a reservation tag is needed to insert the result later on.

The responded reservation tag is given by the memory address the result has to be written to. This is possible, because read and write operations only move the pointers marking head and tail of the queue, while the relation between entry position and memory address stays intact. Further, the reservation does not change anything within the output buffer, since offset tracking is sufficient to avoid multiple reservations to the same queue position and the tag uniquely describes the insertion position. In result,

output ports do not need any synchronization to communicate if a reservation was successful in all ports.

Results arriving from the execution unit are simply written to the addresses defined by the reservation tag, adding the copy counter given by the execution unit.

To find an entry ready to be sent through the DTN, the port unit iterates over all entries, starting at the current read pointer, until if sees a valid value which is not yet marked as read—the output unit tracks read values just like the input unit. A pointer to this entry, as well as the address it contains, is buffered and a new search is started at the read pointer, this time comparing the target address to the currently buffered address. If a matching address in a non-read entry is found, address iteration is canceled and the search for a valid value continues at the buffered pointer. If the address search reaches the buffered pointer without such an occurrence, the buffered entry is the first one in the queue for its target address and can be transfered early. The according target address and value are applied to the data output along with a *valid* indicator. When the container reads, the entry is marked read and a new search is started. Read entries leave the queue the same way as entries leave the input ports: By reaching the head.

The only modifications needed in the container unit are the pass-through of the additional signals, as well as a reduction of all reservation responses to an *ack* if an *ack* was responded by all ports or a *nack* otherwise.

4.5. Buffers With Partial Out-of-Order Execution

In this last implementation approach, a more different buffer design with focus on a high performance increase is presented. Most operations are reduced to a constant number of clock cycles at the cost of a much larger circuit size. Since this is problematic for larger buffers, *partial buffers*, as they are called from here on, can be extended by simple buffer units, restricting out-of-order execution to only parts of the overall buffer size. Architecture is changed in a way that in- and output containers now include a *reorder unit*, which handles all out-of-order related processes for all ports at once.

4.5.1. Input Buffers

The input reorder unit, apart from address and data inputs, value output and reservation interface, has additional inputs for addresses and values from an external extension buffer. Further, it has *full* and *empty* status signals for the source address queues of each port and an indicator if the currently applied source address matches an address within the unit. It is instantiated with a size smaller or equal to the containing buffer, as well as its port count.

To allow concurrent access to all entries contained in the unit, no memory unit is used for storage. Rather, addresses and values are kept in registers directly contained in the

reorder unit. For each port, source addresses, values, and valid flags for the values are stored in a register file. Copy counts are stored in a separate register file. Further, the current address fill for each port is stored and used to indicate the *full* or *empty* status.

Because of the concurrent access, many operations can be solved combinatorially. The insertion position for a value coming from the DTN is continuously determined by using the tree shaped circuit described in Section 4.1.3. Valid candidates have the same source address as the DTN message and no valid data in the port specified by the target address. The result of the search is also used to indicate a matching address to the container, since all indices smaller than the buffer size signal a match. The same method is used to find the next entry to issue to the execution unit. Here, all entries containing valid values in all ports are possible candidates. Further, for each entry in the copy count ports, the sum of all counts beneath the entry is calculated using the same method as for the reduction of logic vectors shown in Section 4.1.2.

Source addresses coming from the MIB are inserted at the position indicated by the current fill of a port, increasing it. Value writes are done directly at the continuously determined position, additionally setting the valid status. An entry written by the external buffer is also inserted at the fill position, only writing the value if the external buffer already contained a valid value for the written entry.

When an set of entries is signaled as ready to be executed through positions unequal to the buffer size, a reservation request is sent to the output buffer. Since the positions of currently valid entries can change any time due to writes from the DTN or external buffer, the positions are buffered at the time of the reservation request. On a negative reply, the reservation process is started from the beginning, possibly using other entries which became available in the meantime. If the reply is positive, the entries at the buffered positions, along with the reservation tag and copy counts, are applied to the execution unit's interface. If these entries are read, all entries in all register files above those entries are shifted down and the corresponding fills are reduced by one, also the reservation process is started again from the beginning.

The extension buffer port units work very similar to the simple port units, the main difference being that the source address is also available at the output. Therefore a modified memory unit is needed, supporting two concurrent read operations on the address memory, since the other is still needed for address insertion. To extend the copy count ports however, the copy count port units of the simple buffer are reused.

The container unit of the partial input buffer has an additional parameter specifying the size of the reorder area used to instantiate the reorder unit and, if needed, extension buffers accordingly. A reorder area the size of whole buffer, leads to a buffer only containing a reorder unit where all write operations are redirected to this unit only.

MIB messages are written to the reorder unit on an apply signal if it is not full and the extension buffer is empty, otherwise the write is applied to the extension buffer. It is important that the extension buffer is empty when writing to the reorder unit, since

otherwise, entries can still be present in the extension buffer, waiting to be moved to the reorder unit. The *full* status towards the MIB is directly connected to the extension buffers, because those represent the tail of the queue. Value writes are directly acknowledged and applied to the reorder unit if it signals a matching entry, otherwise the write procedure from the other buffer types is used. Further, if the reorder unit is not full and the corresponding extension buffer contains an entry at its head, with or without a valid value, this entry is moved to the reorder unit.

4.5.2. Output Buffers

In the output reorder unit, in addition to target addresses and values, reservation tags together with a valid flag are stored in the register file for all ports. Further, for each port, a separate offset for each entry, as well as a mapping from reservation tag to entry position, is stored. Besides the reservation and execution unit interface, an output for completed entries, an address input and *full* and *empty* status signals are available.

The reorder unit concurrently searches entries ready to be sent through the DTN. In the first part of the circuit an intermediate signal is generated, which indicates entries having the first occurrence of their target address relative to the head of the queue of their port. While priority encoders can be used for this operation, they are not necessary, since only existence of an entry with the same address is needed but not its position. Therefore, for each entry, a set of comparators checks all entries below in parallel. The results are reduced via *or*, and the inverted result is fed to the intermediate signal. Mind that the reduction still has a depth in $O(\log n)$. Also, the impact on circuit size is quite large, since each entry needs a number of comparators equal to its position. Entries marked through the intermediate signal, which also have a valid value, are possible candidates for a DTN transfer. The selection is done using the tree shaped priority encoder from Chapter 4.1.3. The found entry is available at the data output.

On a reservation request, availability of space is checked by adding the offsets stored for the requested positions to the positions themselves and the requested result sizes for all ports. If one of the sums exceed the buffer size, reservation is not possible. For requests to position zero only the offsets and positions are considered. When a reservation is possible in all ports, the next unused tag is assigned to the entries at the calculated insertion positions. This positions are also inserted into the the tag map for the corresponding tag and port. Tags are a number between zero and the buffer size minus one, since this represents maximum number of possible concurrent reservations. The offsets describe the state in the input buffer and are therefore not directly related to the entries in the output. For this reason, all offsets above and including a requested position, are shifted down by the copy count of the result, instead of shifting at the final insertion position. Offset slots getting empty at the tail are initialized with the previous topmost offset. Updates to offsets happen at reservation, since entries are instantly removed from the input buffer if a reservation is successful.

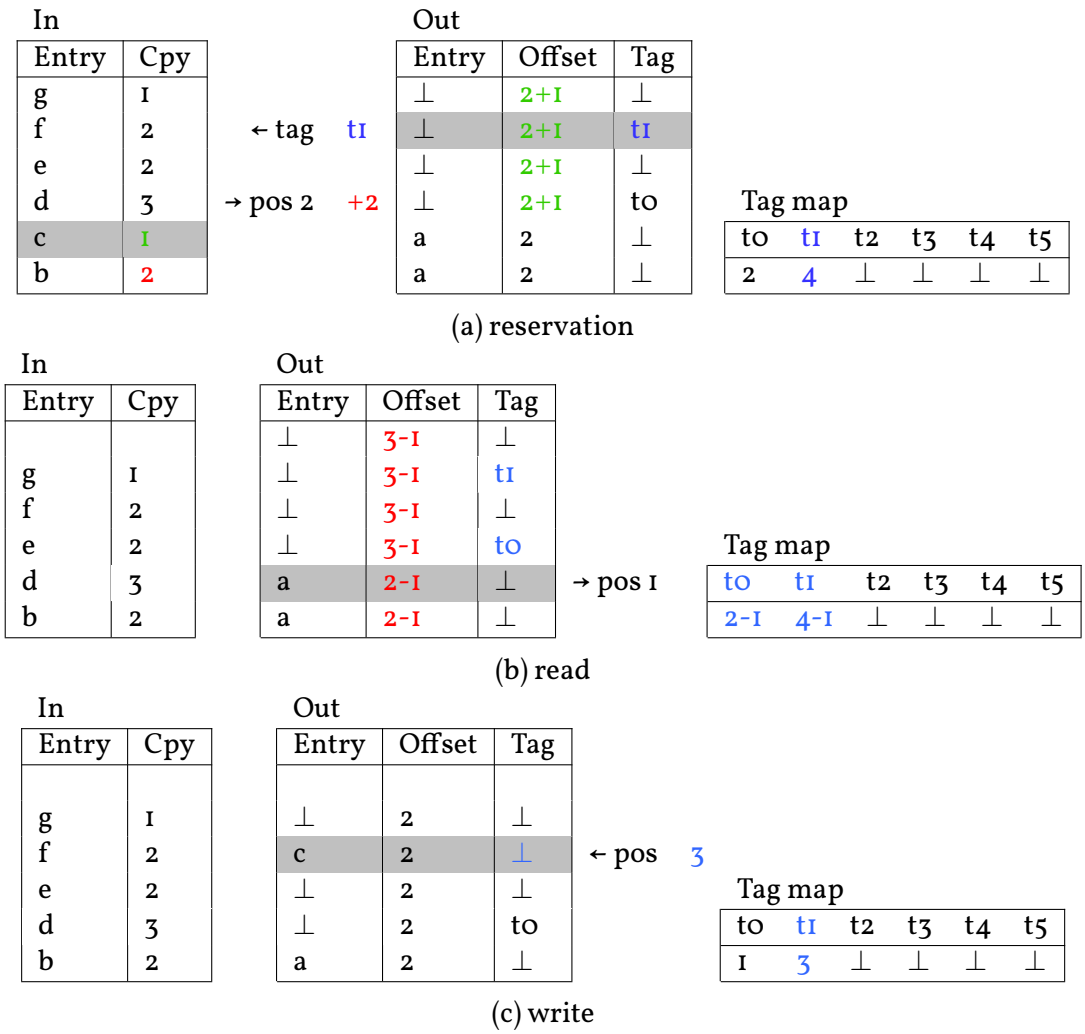


Figure 4.10.: Example for reservation, writing and reading

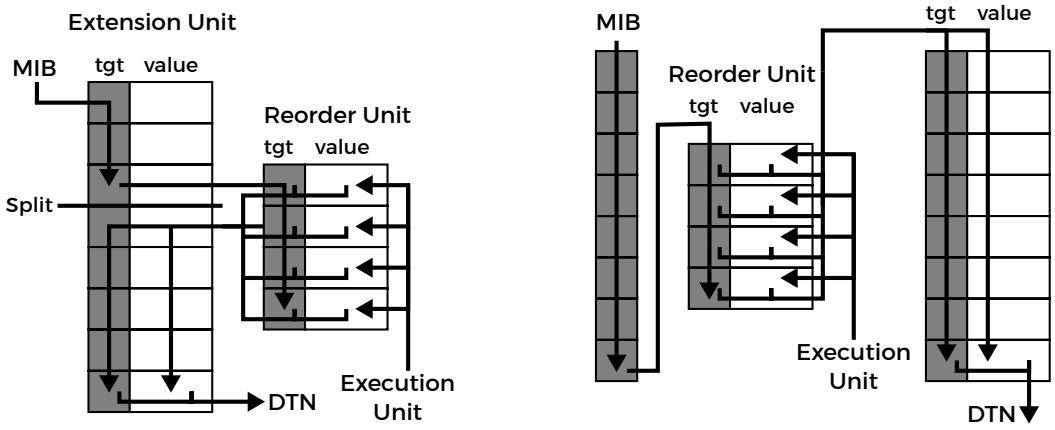
When results arrive at the reorder unit, the insertion position for each port is looked up in the tag map. Values are inserted at this position and marked valid, while the reservation tag is marked invalid.

When an entry is read, to represent changes in contrast to the input buffer, all offsets which would redirect an insertion to or above the read entry, i.e. $position + offset >= read\ position$, are decreased by one. Further, all entries in the read port, which are above the read entry, are shifted down by one position. If a shifted entry contains a valid reservation, the tag map is also updated to reference the new position.

Figure 4.10 shows examples for the different processes in the output reorder unit. At reservation, entry *c* is at position *two*, having *two* copies of *b* underneath, and is redirected by an additional offset of *two*, caused by entry *a* already having been read from the input buffer. The result fits into the output unit, and the next free tag *t1* is returned,

while offsets starting at position *two* are increased by the copy count *one*. The tag is referenced at the reserved entry and the tag map now stores the reserved position *four*. During the read operation in the next step, all offsets redirect above the read position, therefore being decreased. Both reservations are also above the read position and the tag map is updated accordingly. When the result is finally being written, its current position is looked up in the tag map, now being *three*. The value is inserted at this position and the reservation reference is cleared.

Extension buffers in the output container need special units. On one hand, target addresses need to be buffered if the reorder unit is full, on the other hand, completed entries have to be stored if the DTN is busy.



(a) Split extension buffer

(b) Dual address extension buffer

Figure 4.11.: Different extension buffer approaches

A first approach, providing exactly the size specified for the output buffer, is shown in Figure 4.11a. Here the extension unit is split dynamically into two areas, one only containing target addresses, the other only completed entries. While this solution at first sight looks rather compact, it yields some problems. The value spaces in the address part stay unused, limiting the value capacity of the buffer in cases where not much out-of-order execution is possible. Further, two write operations on the address part are needed, while the used distributed RAM only supports one write per clock cycle. This can be solved by additional resolution logic and buffers, at the cost of making the extension unit more complex. Therefore, for the final implementation a more simple solution is chosen (cf. Figure 4.11b). A port extension unit is used, which has two separate address memories for incoming addresses and completed entries, both having the full size of the unit. Since addresses are comparably small and the FIFO type memory management does not need much logic, the impact on overall circuit size is still reasonable. Furthermore, this added address storage can decrease stalls of the control unit caused by full buffers.

In result, the extension units use a memory module providing an additional address memory. They have separate in- and outputs for addresses and completed entries and organize both parts as simple FIFO buffers.

The container unit instantiates a reorder unit and extension port units for each port. The Execution unit and reservation interfaces are directly connected to the reorder unit. Address writes from the MIB are handled the same way as in the partial input buffer. Equally, addresses are moved from the extension unit to the reorder unit, this time without values. If the reorder unit signals a valid entry at its output and the corresponding part of the extension buffer is not full, this entry is moved from the reorder unit to the extension unit. DTN reads are distributed across the extension units like in the simple and partial buffers.

5. Evaluation

At the time of this work, no complete SCAD-architecture-based processor implementation was available. Therefore, evaluation is done on a local scale, based on single functional units in combination with dummy execution units. Because of this, no program examples are used. These are also problematic, since execution depends a lot on the available functional units of the processor, the DTN and how the compiler schedules instructions to the execution units, therefore introducing a vast amount of additional parameters. The goal of this first evaluation of the presented out-of-order algorithm, is to get a general overview over the possibilities and limits in execution performance and to explore worthwhile application scenarios. Therefore, the focus in the following is the dataflow-based value throughput of functional units, as well as its relation to the circuit size.

Evaluation of the algorithm is done in two parts: First, some measurements are done to estimate improvements in execution performance. In a second step, the units are synthesized to FPGA hardware and the circuit size is correlated with the execution performance measured before.

5.1. Execution Performance

To measure execution performance, test benches written in VHDL are used, which instantiate an in- and output buffer with arbitrary buffer sizes and connect them using a dummy execution unit. The simulation is done using the simulator delivered with the Vivado suite.

Although only single functional units are evaluated, a lot of parameters that can be varied are available. Those include buffer size, port count, execution delay and pipeline depth, DTN delay, order of incoming data, copy counts and MIB delay. Additionally, for the partial buffer the size of the out-of-order area can be varied. In the following some of these variables are ruled out, in order to simplify evaluation and concentrate the focus on the impact of the out-of-order algorithm.

All buffer units in this series of tests have exactly one input and one output port. This has several reasons. First, the port count does not matter for the output buffers, as each port can execute its part of the algorithm independently. Secondly, the delay that is caused by entries being ready in one input port but not all others, can be simulated by delaying insertions into a single input port accordingly. And lastly, while variational

behavior can come from different input delays in buffers, it is a very sublime and hardly assessable factor, and therefore not considered for this first performance evaluation.

Partial buffers are always tested with a reorder area of 100%, 50% and 25% of the buffer's size, the 25% case is skipped for buffers of size *four*, since the size of reorder units needs to be a multiple of *two* for technical reasons. Further, buffers are generally instantiated with sizes of 4, 16, 64 and 256.

The connection to the MIB and DTN is simulated by the test bench. In a first step, all move instructions described by a testing scenario are written through the MIB ports of both buffer units. The time needed for this operation is not included in the measurement, since on one hand, it is the same for all buffer types because all test sets completely fit into the buffer, and on the other hand, move instructions describe control flow and cannot directly be influenced by dynamic scheduling. Nevertheless, one should keep in mind that indirectly the control flow can be influenced by freeing entries in the buffers earlier and reducing the stalling probability of the control unit.

Then data is inserted into the input buffers through the DTN input, using the order and delays described by a given testing scenario. Since the DTN is spread across the whole processor, its clock rate is expected to be lower than the clock rate of the locally concentrated functional units. With this in mind, the length of one DTN clock cycle for evaluation is always based on $buffer\,size + 2$ functional unit clock cycles, which is the maximum delay needed to insert a value at the correct position into an input buffer. This number is chosen because it represents the borderline between the DTN being able to insert a value into the input buffer in every cycle and additional delays being introduced by buffers still being busy with inserting values in some cycles. In addition, this is particularly interesting for the partial buffers, where the insertion process is faster in some cases.

As soon as the input buffer signals executable data, the dummy execution unit begins simulating execution. It can be configured to either wait for a specified number of clock cycles and then attempt to write a result to the output buffer, or to simulate a pipeline of arbitrary length, stalling if the output buffer is not ready to accept new data. The result is always the same value as read from the input.

Copy counts are constantly set to *one* during this evaluation, since their only impact is a higher delay caused by the execution unit writing the additional copies. Such a delay is already covered by changing the number of computation cycles the dummy execution unit simulates.

The test bench reads a result from the output buffer in every DTN cycle if a valid value is available.

For measurement, the number of elapsed DTN clock cycles is counted from the first value being written to the input buffer to the last expected result being read from the output buffer. The performance increase is then expressed as speedup

$$S_{type} = \frac{T_{simple}}{T_{type}}.$$

T_{simple} is the number of DTN cycles a functional unit using simple buffers needs to complete a testing scenario and T_{type} the number of cycles needed by the different extended buffers, *type* being *ooo* for the out-of-order buffer and *p25*, *p50* or *p100* for the partial buffers with reoder areas of 25%, 50% or 100% respectively.

5.1.1. Testing Scenarios

This section describes the different testing sets used for performance evaluation. Remember that the input order is not defined by the addresses in the input buffer, but rather the given timing—addresses only define the position a value is inserted into.

The first set of scenarios, shown in Figure 5.1, describes combinations of worst- and best-cases for out-of-order execution in the in- and output buffers respectively. These tests are supposed to limit the range of the possible speedup to be expected from out-of-order execution. In the worst-case (*wc*), all values arrive at the input buffer in-order and no out-of-order execution is necessary, so the partial and out-of-order buffers are expected to perform worse than simple buffers, since the reservation and search processes generate additional delays. To simulate the best-case (*bc*), all values arrive in the reverse order of the control flow at the input buffer. Additionally, the output buffer contains only distinct target addresses. Out-of-order execution should noticeably increase performance in this case, since each incoming value can directly be executed and each result can instantly be forwarded to the DTN. The scenario *bwc* is a combination of the best-case input and worst-case output buffer situation. Here, results arrive faster at the output buffer through out-of-order execution, but all values have to be reordered before sending. This can be problematic for the out-of-order algorithm, since it puts additional effort into reordering values, whereas results cannot be sent early in the end. The remaining combination can be ignored, since results arriving at the output buffer are calculated using the head of the input buffers and can by directly sent through the DTN in any case. All of these scenarios are extended to larger buffer sizes by continuing each series of addresses or delays.

In		Out
src	*t*	*tgt*
d	4	i
c	2	i
b	3	i
a	1	i

(a) wc

In		Out
src	*t*	*tgt*
a	1	i
b	2	j
c	3	k
d	4	l

(b) bc

In		Out
src	*t*	*tgt*
a	1	i
b	2	i
c	3	i
d	4	i

(c) bwc

Figure 5.1.: Best- and worst case testing scenarios

In addition to buffer and reorder area size, the DTN clock is varied between its base clock, as well as half and one fourth of the base clock for this scenario. This is to explore

the influence of the insertion delay in the input buffers. The execution unit has a static delay of one functional unit clock cycle and a pipeline depth of one.

The next set of testing scenarios describes different interleaved data streams, these are shown in Figure 5.2. Here, the goal is to evaluate the performance in more realistic situations which the algorithm is designed for. The first one (*even*) describes two independent data streams passing through the functional unit, both evenly interleaved. However, this is not respected in the control flow, since both streams are expected to be processed sequentially by the functional unit. Scenario *uneven* describes a similar situation: This time stream *b* has three times as many values as stream *a* and values for *b* arrive only every fourth cycle. This test relates to the example of a data stream generated by a slow functional unit interfering with other streams having higher data rates. Results in those two scenarios have a distinct target buffer for each stream. In the last test, the input buffer is filled like in *uneven*, *b* spreads its results over different target buffers. This is done to see how execution behaves if not all values can be instantly forwarded by the algorithm.

These patterns are expanded in such a way that all ratios remain constant. In the input buffer of size 16 for example, the lower four entries are occupied by *a*, with values arriving in cycles 1,5,9 and 13.

All these scenarios are executed only at the half the base DTN clock to include effects of insertion delays, whereas the dummy execution unit is instantiated as one cycle, eight cycle and pipelined eight cycle unit.

In		Out
src	*t*	*tgt*
b	8	j
b	6	j
b	4	j
b	2	j
a	7	i
a	5	i
a	3	i
a	1	i

(a) even

In		Out
src	*t*	*tgt*
b	8	j
b	7	j
b	6	j
b	4	j
b	3	j
b	2	j
a	5	i
a	1	i

(b) uneven

In		Out
src	*t*	*tgt*
b	8	j
b	7	j
b	6	j
b	4	j
b	3	i
b	2	i
a	5	i
a	1	i

(c) uneven spread

Figure 5.2.: Interleaved data stream testing scenarios

5.1.2. Results

The results of the worst-case scenario are as expected. The partial and out-of-order buffers, perform worse than the simple buffer. However, with increasing buffer size,

size	$\frac{DTN}{1}$	$\frac{DTN}{2}$	$\frac{DTN}{4}$
4	0.64	0.77	0.71
16	0.90	0.90	0.48
64	0.99	0.96	0.94
256	1.0	0.99	0.98

Figure 5.3.: *Wc* results for buffer type *ooo* at different DTN speeds

the performance decrease becomes smaller until nearly the same performance as with the simple buffer is reached. The DTN speed only has a minor influence on buffers of type *ooo* as shown in table 5.3. The anomaly in the buffer of size 16 at fourth the DTN speed can be explained through unfavorable timings of search resets in the presented implementation. Partial buffers generally have less negative impact in the worst-case situation, the *p100* buffer showing no decrease at all. In the buffers of type *p50* and *p25*, the overhead only shows for smaller sizes of 4 and 16, having a slightly larger impact at higher DTN speeds.

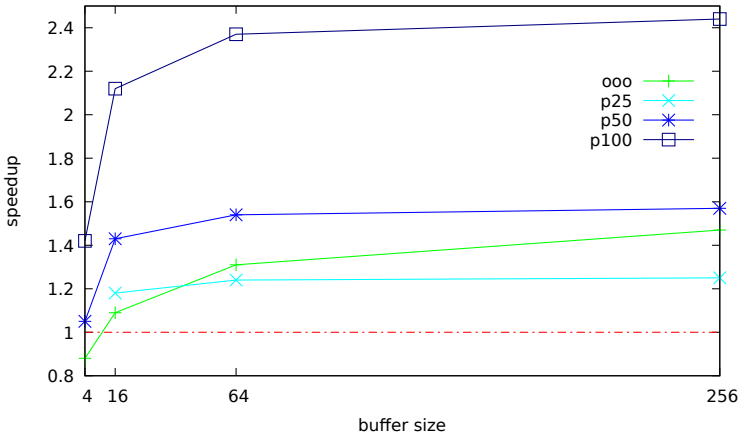


Figure 5.4.: Average speedup in *bc* for different buffer sizes

The graph in Figure 5.4 shows the average speedup behavior for all DTN speeds in the best-case scenario for different buffer sizes. The size of the reorder area in the partial buffers has a huge impact on overall speedup. This is due to the fact that buffers are filled in reverse and out-of-order execution cannot start until values are inserted into the reorder unit at the bottom of the buffer. Only buffers of size four are impacted by the additional overhead, all others show an increase in performance. While the partial buffers start at rather high speedup factors, they quickly saturate with increasing buffer size. Out-of-order buffers on the other hand, further increase performance with growing buffer size.

Figure 5.5 shows the average speedup over all tested buffer sizes for the different DTN speeds. Here it becomes obvious that the partial buffers—at least for the covered buffer

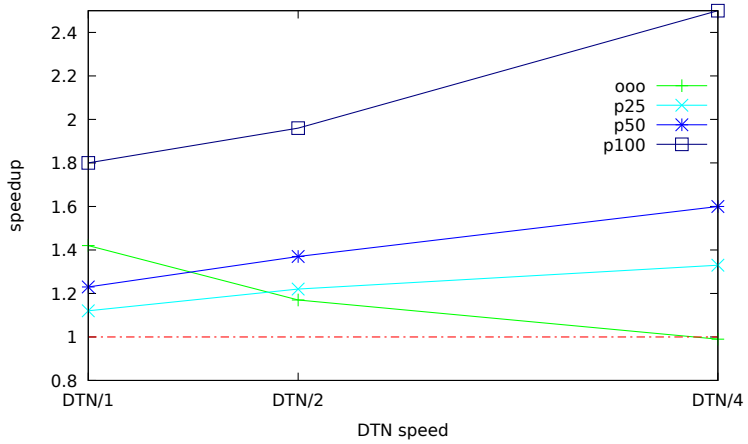


Figure 5.5.: Average speedup in *bc* for different DTN speeds

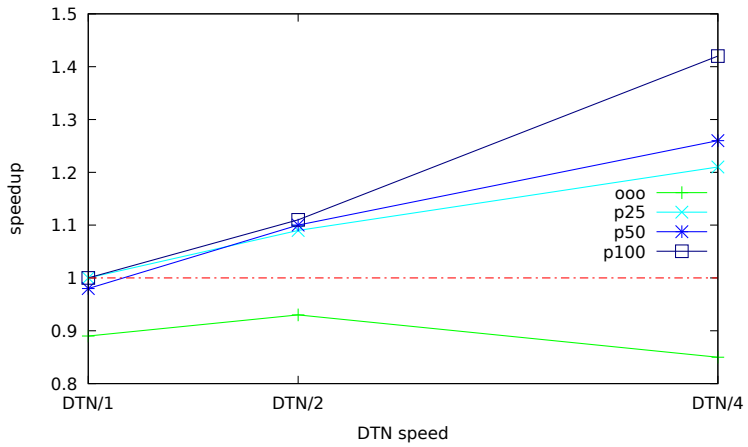


Figure 5.6.: Average speedup for *bwc* at different DTN speeds

sizes—generally benefit from higher DTN speeds. Especially the *p100* buffer, since it has no insertion delay. For the out-of-order buffers, the opposite is the case. Also, with higher DTN speed, delays caused by the multi-cycle search operations become more obvious.

The *bwc* scenario shows a similar trend as the *bc* scenario for partial buffers, while yielding a considerably lower overall speedup (cf. Figure 5.6). No performance increase happens at the base DTN speed, since speedup mainly comes from avoiding idle cycles caused by insertion delays, as well as some cases where consecutive results are available earlier in the output buffer and sending profits from the higher DTN transfer speed. For out-of-order buffers, the long search delays and the missing improvement of the insertion process causes an overall slowdown, not being reduced by increasing buffer size.

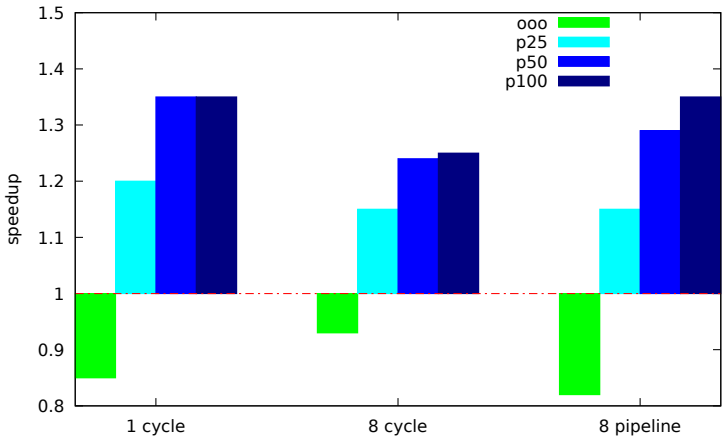


Figure 5.7.: Average speedup for different execution delays for buffers of size 16

The second set of tests generally shows that a delay induced by the execution unit, pipelined or not, has no impact at all on tested buffer sizes larger than 16. This can be explained by the DTN delay being half the buffer size, while execution delays are statically chosen to be eight cycles. Therefore, the rate of incoming values is much lower than the rate at which results are calculated in the tested buffers starting at size 64. For this reason, Figure 5.7 uses the average speedup factor for all scenarios of the second set of tests, only considering buffers of size 16. Buffers of size four are left out, since buffers of type *p25* have no support for this size and would therefore stand out, because all other buffers show a performance decrease or no increase at size four. The chart demonstrates that for the partial buffers a higher data throughput in the execution unit by means of a lower execution delay or pipelining yields better performance through out-of-order execution. For the buffers of type *ooo* in contrast, a higher delay is generally more favorable, leaving more time for the algorithm to do its work.

In general, the out-of-order buffers perform badly in the interleaved data stream scenarios. As it can be seen in Figure 5.8, the *even* scenario shows no performance increase for all sizes, while still better bad with increasing size, possibly yielding a speedup for larger sizes. In the scenarios with unevenly interleaved streams, where execution in the simple buffers is blocked by single values for a longer period, execution is sped up earlier, starting at sizes 64 and 256 respectively.

Partial buffers are more suitable for interleaved data streams, only decreasing performance for small buffers of size four. They also perform best in the *interleaved* scenario, but handle *even* better than *spread*. This shows that they suffer more from a higher delay needed for sending results than from a shorter time period to handle out-of-order execution.

An overview over the average speedup in all scenarios using a one cycle execution unit for all tested buffer sizes of partial buffers is given in Figure 5.9. Here it can be seen that, while the *p25* buffer still brings a reasonable performance increase, *p50* and *p100*

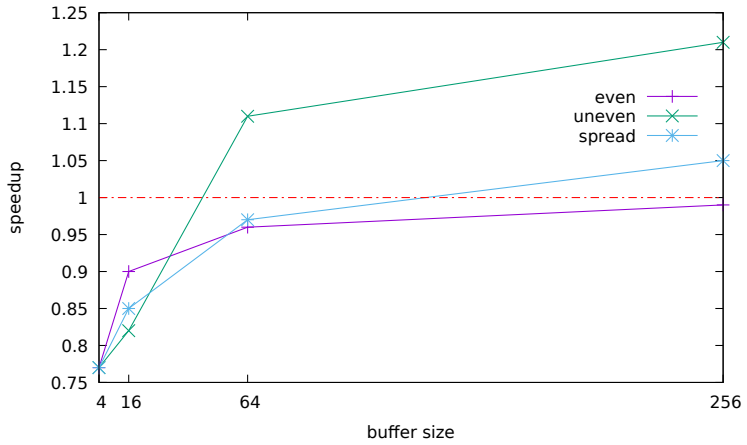


Figure 5.8.: Speedup of *ooo* buffers for interleaved streams with 1-cycle execution delay

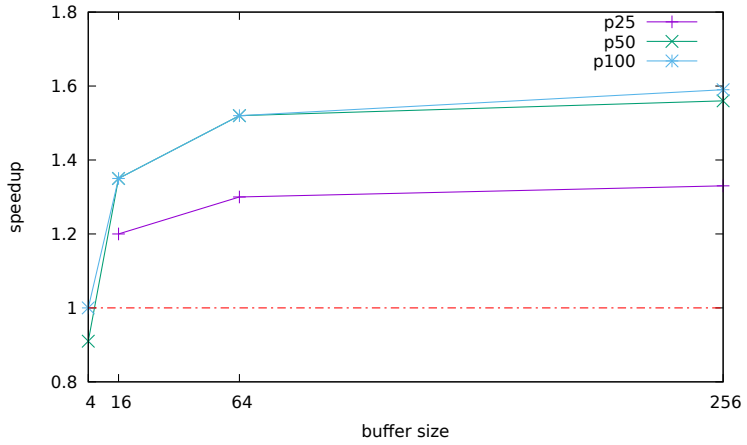


Figure 5.9.: Average speedup of partial buffers with 1-cycle execution

type buffers are even faster. However, both, *p50* and *p100*, reach nearly the same speed. This is due to the delayed entries accumulating in the lower part of the buffers, filling up the reorder area in case of the *p25* buffer. In buffers with larger reorder areas, space is left for other entries to pass.

As a general trend, this shows that partial buffers yield better performance for processors with high data transfer speeds, small buffers and overall high clock rates. Out-of-order buffers in contrast do profit from exactly the opposite, even reducing performance in the more realistic interleaved data stream scenarios if buffers are too small and the DTN speed is too high.

5.2. Circuit Size

Now a look is taken at the circuit size needed by different types and sizes of buffers. Therefore, single instances of input and output buffers are synthesized using the synthesis tool included in Vivado. Not only the sizes used for performance evaluation are synthesized, but all multiples of two from 4 to 512. This is because circuit size is not expected to grow linearly in all cases, due to address widths and characteristics of the FPGA hardware, and more fine grained results are desirable. Parameters are chosen with a reasonable processor design in mind. Input buffers have two and output buffers have one port. Values have a width of 32 bits, copy counts a width of 4 bits and buffers are addressed using 8 bit addresses.

Circuit size is measured by using the amount of LUTs[1] used for logic in combination with the number registers needed by the synthesized design, totaling to the number of slices needed on the FPGA. LUTs used for distributed RAM are excluded from the measurement, since the amount is mostly the same for all buffer types, except for the partial buffers where the particular interest lies in how not having entries in memory affects circuit size. Further, the memory units are designed to be exchangeable and contain little to no logic related to the out-of-order algorithm, therefore they are not considered when looking at the logic size.

For correlation with the performance measurements, the results are again converted to a factor I_{type} describing the increase in circuit size in contrast to the simple buffers:

$$I_{type} = \frac{C_{simple}}{C_{type}}$$

Here C_{simple} and C_{type} express the sum of used LUTs and registers for the input as well as the output buffers of the according buffer types.

5.2.1. Results

A first look at only the size of the required logic for the different buffer implementations in Figure 5.10 shows that the simple buffers have a small circuit growing linearly with increasing buffer size. Adding the out-of-order logic increases the growth factor, but size is still increasing linearly for the tested buffer sizes. Mind that the added circuitry does not simply add a static overhead, because it is also dependent on the larger address sizes needed by larger buffers, since out-of-order execution needs more intermediate address buffers.

Storing entries in registers heavily impacts the size of the partial buffers. All tested instances show exponential growth where the exponent increases with the size of the reorder area, quickly generating circuit sizes magnitudes larger than the *simple* or *ooo* type buffers. This is also the reason why some of the larger partial buffers are absent in

[1] Lookup Table: Basic building block used for logic on FPGAs.

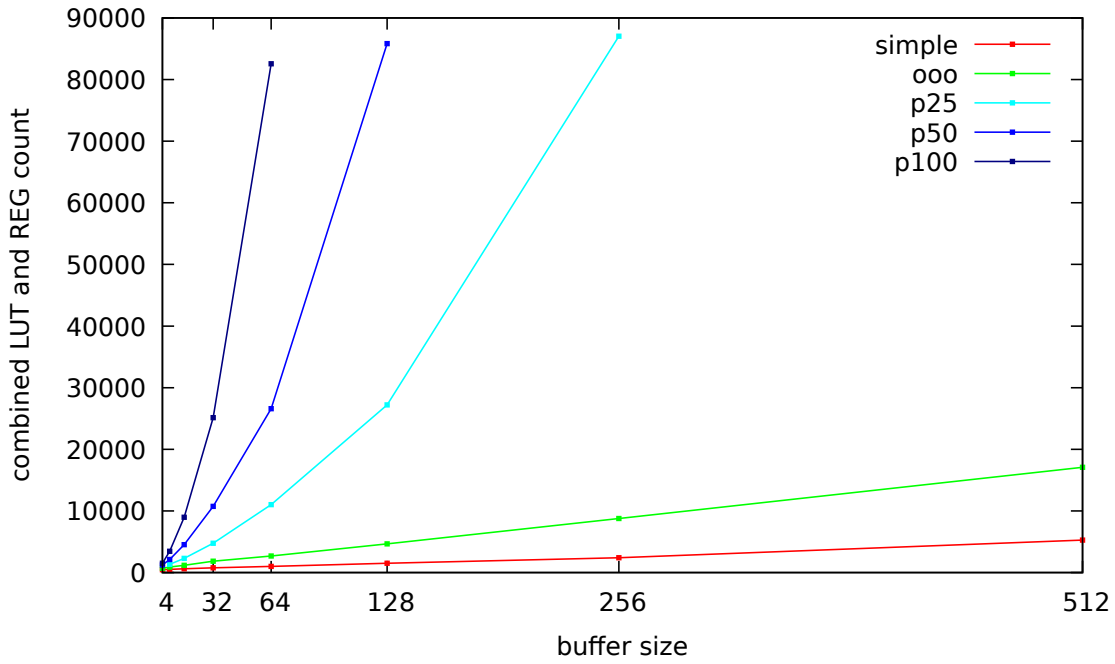


Figure 5.10.: Combined LUT and register count after synthesis

the results, since the synthesis tool was not able to synthesize *p100* larger than 256, *p50* larger than 128 and *p25* larger than 64.

Figure 5.11 puts the average speedup factor for the best-case scenario for all tested buffer sizes in relation to the increase in circuit size towards the simple buffer implementation. Here the *ooo* type buffers show a reasonable performance increase while keeping the circuit size low. The *p25* buffers only have a slight advantage at size 16, needing little more logic at a higher performance. Using the *p50* and *p100* buffers yields a good performance to size ratio at buffer size four. At size 16 both give a high performance boost at a considerably larger circuit size. Especially the partial buffer with 100 percent reorder area has a really good speedup factor, but can only be used if physical space is not an issue. Partial buffers with larger numbers of entries, while still increasing performance, yield a very high circuit size, making them only suitable for very specific application scenarios.

Looking at the average results for the interleaved data stream tests in Figure 5.12, the situation looks different. Out-of-order buffers only bring improvements for sizes larger than 64, buffers of size four do not yield any speedup in general. Partial buffers *p25* with size 16 and 64 have a good speedup factor by still showing a comparably low circuit size increase. At size 16, *p50* and *100* both have the same good performance increase in contrast to the *p25* buffer, while the *p100* buffer has a notably larger size. The higher speedup from any of the larger buffers can unlikely be utilized, because of the high increase in circuit size.

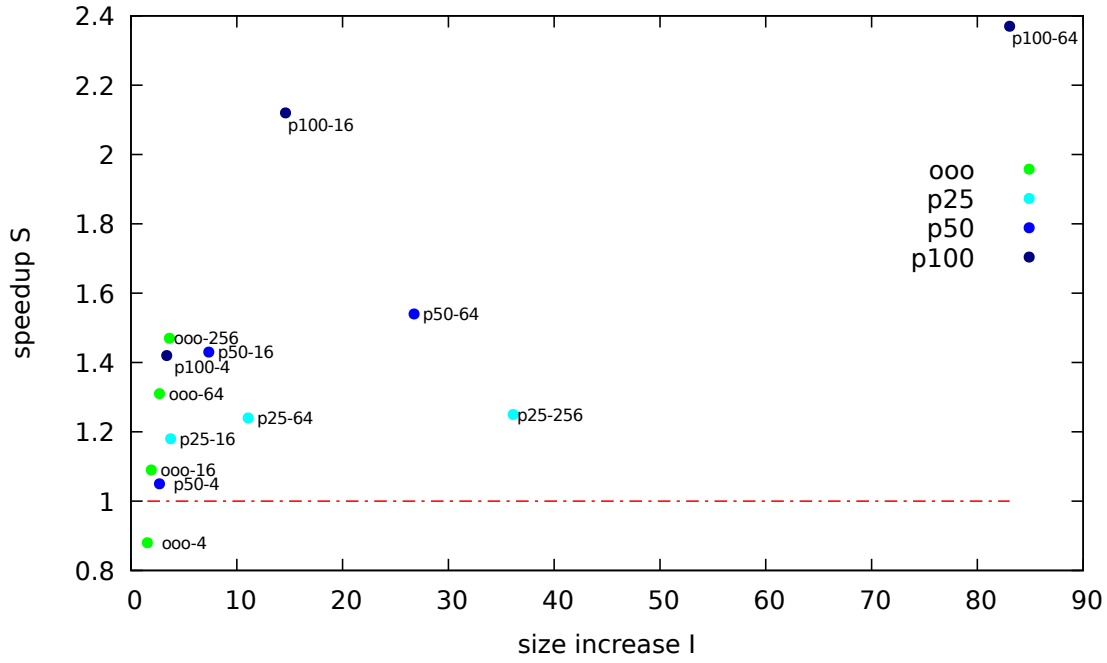


Figure 5.11.: Circuit size in comparison to speedup in *bc*

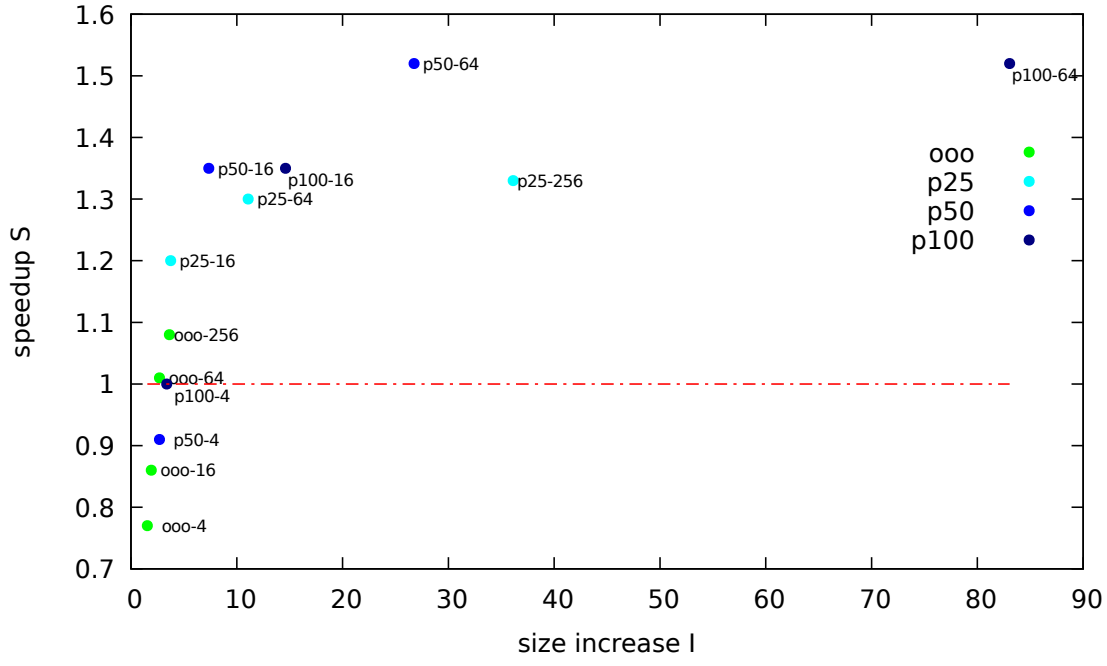


Figure 5.12.: Circuit size in comparison to speedup for interleaved streams

All in all, this shows that performance increases, even if small, mostly lead to considerably larger circuits, especially in the case of the partial buffers. However, partial buffers demonstrate to have a reasonable circuit size to performance ratio for smaller buffers whereas buffers of type *ooo* still can be used for large buffers without impacting physical area too much. This coincides with the results from the pure performance evaluation and should make the selection of the right buffer type for a specific application more straightforward.

6. Conclusion

This work shows that the execution performance of functional units of the SCAD architecture can be improved in many cases by enabling out-of-order execution using the proposed algorithm. Modified buffers, needed by this algorithm, are also realizable on FPGA hardware, even for large buffers. This has been demonstrated by means of different implementations.

However, evaluation made clear that none of the buffer implementations is optimal in every situation. Choosing the wrong type of buffer can even lead to performance losses. Which buffer is most useful, depends on many parameters: Clock speeds, desired buffer size, available chip area and, most importantly, the expected patterns of incoming data. Buffers have to be carefully chosen and parametrized for each application. Nevertheless, since SCAD architectures are already very suitable for application-specific processors, this gives additional flexibility to further optimize a processor for particular tasks. On the contrary, out-of-order execution can also help to make a highly specialized processor behave better at executing tasks it is not optimized for.

The test results acquired in this thesis should in no way considered to be final. First of all, tests were only done local to the functional units. It is possible that out-of-order execution behaves differently when looking at the performance of a processor globally. Having a working processor also allows to explore further ideas. Incoming and outgoing data streams can be profiled for single functional units, helping to choose the right out-of-order buffer type or even to decide if out-of-order execution is needed at all for a specific unit. What is also yet to explore, is the maximum clock frequency possible. Although the circuit depth was kept in mind during the design phase, it would be interesting to see at which clock speeds such buffers can be realized

Secondly, the implementations are not yet optimal. The given set of approaches is intended give a first impression on the effects of out-of-order execution and to transport a set of ideas on how such an algorithm can be realized. Circuit area, mainly in the case of partial buffers, can certainly be reduced. Also, the optimization of timings, mainly when to restart searches in the out-of-order buffer, should be looked further into. Search can be reset, for example, on new values arriving at the buffer or if space is being freed in the output buffers.

Other possible optimizations are also interesting to explore in the future: Value selection can not only depend on its position in the queue, but also on the number of produced copies or on how soon it can be transferred by the output buffer. Further, out-of-order input buffers could start to shift entries to use already read locations when the address queue is full, avoiding stalls of the control unit. Also the role of the compiler

can be looked into. It can take the availability of out-of-order execution into account when determining the instruction order. Prioritizing instructions to be preferred during value selection by the out-of-order algorithm is also thinkable.

One could also consider designing a SCAD processor only consisting of very few, maybe even just one functional unit with multiple functionalities. Giving those units really large buffers with out-of-order execution could also lead to a good level of ILP. In this context it is also left to be evaluated how different memory types, like block RAM, can be utilized.

Lastly, the presented buffer types should not be considered fixed as well. Combinations of reorder units with out-of-order buffers are possible, as well as combining different types for input and output buffers.

All in all this proves out-of-order execution to be a beneficial addition to the SCAD architecture, with a lot of potential left to be explored in the future.

Bibliography

[1] IEEE standard VHDL language reference manual - redline. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline*, pages 1–620, January 2009.

[2] K. Aasaraai and A. Moshovos. Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming. In *2009 International Conference on Field Programmable Logic and Applications*, pages 79–85, August 2009.

[3] P. J. Ashenden. Recursive and repetitive hardware models in VHDL. Technical report, TR 160/12/93/ECE, University of Cincinnati, and TR 93-19, University of Adelaide, 1993.

[4] D. Baudisch. *Synthesis of Synchronous Programs to Parallel Software Architectures*. PhD thesis, Department of Computer Science, University of Kaiserslautern, November 2013.

[5] D. Baudisch, J. Brandt, and K. Schneider. Out-of-order execution of synchronous data-flow networks. In J. McAllister and S. Bhattacharyya, editors, *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS)*, pages 168–175, Samos, Greece, 2012. IEEE Computer Society.

[6] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous data-flow networks. *International Journal of Parallel Programming (IJPP)*, 43(1):1–44, February 2015.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598, February 2008.

[8] V. E. Beneš. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43(4):1641–1656, 1964.

[9] A. Bhagyanath, T. Jain, and K. Schneider. Towards code generation for the synchronous control-flow/asynchronous dataflow (SCAD) architecture. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, Freiburg, Germany, 2016.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, pages 3255–3258 vol.5, May 1995.

[11] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, July 2004.

[12] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture*, 45(12):949–973, 1999.

[13] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual – volume 1: Basic architecture, April 2016.
`https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf`.

[14] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[15] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 140–150, New York, NY, USA, 1983. ACM.

[16] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[17] L. R. Goke and G. J. Lipovski. Banyan networks for partitioning multiprocessor systems. In *Proceedings of the 1st Annual Symposium on Computer Architecture*, ISCA '73, pages 21–28, New York, NY, USA, 1973. ACM.

[18] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, January 1985.

[19] Synopsys Inc. Guide to HDL coding styles for synthesis, June 2002.

[20] Xilinx Inc. UG473: 7 series FPGAs memory resources – user guide (v1.11), November 2014.
`http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.

[21] Xilinx Inc. UG474: 7 series FPGAs configurable logic block – user guide (v1.7), November 2014.

http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[22] Xilinx Inc. UG901: Vivado design suite user guide – synthesis (v2015.3), March 2015.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_3/ug901-vivado-synthesis.pdf.

[23] Xilinx Inc. UG900: Vivado design suite user guide – logic simulation (v2016.1), April 2016.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug900-vivado-logic-simulation.pdf.

[24] T. Jain and K. Schneider. Designing and implementing a synchronous control asynchronous dataflow machine, 2016. Hardware-Software Synthesis Project Specification.

[25] G. Kahn. The semantics of a simple language for parallel programming. *In Information Processing*, 74:471–475, 1974.

[26] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[27] F. J. Mesa-Martinez et al. SCOORE santa cruz out-of-order RISC engine, FPGA design issues. In *Workshop on Architectural Research Prototyping (WARP), held in conjunction with ISCA-33*, pages 61–70, 2006.

[28] R. B. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1):9–50.

[29] M. Rosière, J. l. Desbarbieux, N. Drach, and F. Wajsbürt. An out-of-order superscalar processor on FPGA: The ReOrder buffer design. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1549–1554, March 2012.

[30] K. Schneider. Hardware-software systems. Lecture Notes, Technical University of Kaiserslautern, October 2013.

[31] K. Schneider. Parallel computing. Lecture Notes, Technical University of Kaiserslautern, April 2013.

[32] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Computers*, 37(5):562–573, 1988.

[33] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, January 1967.

[34] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. 1986.

[35] J. von Neumann. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993.

[36] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

A. Raw Evaluation Results

A.1. Performance

Timing results T are given in DTN clock cycles.

Simple buffer

size	wc			bc			bwc		
	DTN/1	DTN/2	DTN/4	DTN/1	DTN/2	DTN/4	DTN/1	DTN/2	DTN/4
4	7	10	12	10	14	17	10	14	17
16	18	19	19	33	38	48	33	38	48
64	66	66	66	129	145	195	129	145	195
256	258	258	258	513	577	795	513	577	795

size	even			uneven			uneven spread		
	1 cycle	8 cycle	8 cycle pipeline	1 cycle	8 cycle	8 cycle pipeline	1 cycle	8 cycle	8 cycle pipeline
4	10	17	10	10	17	10	10	17	10
16	26	29	26	28	32	28	28	32	28
64	97	97	97	113	113	113	113	113	113
256	385	385	385	457	457	457	457	457	457

Out-of-order Buffer

size	wc						even					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	11	0.64	13	0.77	17	0.71	13	0.77	17	0.89	10	0.71
16	20	0.90	21	0.90	40	0.48	29	0.90	29	1.0	26	0.84
64	67	0.90	69	0.96	70	0.94	101	0.96	97	0.96	97	0.96
256	259	1.0	261	0.99	264	0.98	389	0.99	385	0.99	385	0.99

size	bc						uneven					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	12	0.83	15	0.93	19	0.89	13	0.77	19	0.89	14	0.71
16	25	1.32	35	1.09	56	0.86	34	0.82	34	0.94	34	0.82
64	77	1.68	116	1.25	194	1.01	102	1.11	103	1.11	102	1.11
256	279	1.84	416	1.39	671	1.18	379	1.21	379	1.21	379	1.21

size	bwc						uneven spread					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	15	0.67	18	0.78	23	0.74	13	0.77	19	0.89	14	0.71
16	36	0.92	40	0.95	70	0.69	33	0.85	37	0.86	35	0.8
64	132	0.98	150	0.97	197	0.99	116	0.97	116	0.97	116	0.97
256	516	0.99	579	1.0	802	0.99	436	1.05	435	1.05	436	1.05

Partial Buffer 25

size	wc						even					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	-	-	-	-	-	-	-	-	-	-	-	-
16	18	1.0	19	1.0	20	0.95	23	1.13	25	1.16	24	1.08
64	66	1.0	66	1.0	66	1.0	82	1.18	82	1.18	82	1.18
256	258	1.0	258	1.0	258	1.0	322	1.2	322	1.2	322	1.20

size	bc						uneven					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	-	-	-	-	-	-	-	-	-	-	-	-
16	30	1.1	32	1.19	38	1.26	19	1.47	25	1.28	20	1.4
64	144	1.13	118	1.23	144	1.35	66	1.71	66	1.71	66	1.71
256	450	1.14	466	1.24	576	1.38	258	1.77	258	1.77	258	1.77

size	bwc						uneven spread					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	-	-	-	-	-	-	-	-	-	-	-	-
16	33	1.0	35	1.09	41	1.17	28	1.0	29	1.1	29	0.97
64	129	1.0	133	1.09	159	1.23	111	1.02	111	1.02	111	1.02
256	513	1.0	529	1.09	639	1.24	447	1.02	447	1.02	447	1.02

Partial Buffer 50

size	wc						even					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	8	0.88	11	0.91	14	0.88	11	0.91	18	0.94	14	0.71
16	18	1.0	19	1.0	20	0.95	19	1.37	24	1.2	20	1.3
64	66	1.0	66	1.0	66	1.0	66	1.47	66	1.47	66	1.47
256	258	1.0	258	1.0	258	1.0	258	1.49	258	1.49	258	1.49

size	bc						uneven					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	10	1.0	13	1.08	16	1.06	11	0.91	18	0.94	13	0.77
16	26	1.27	27	1.41	30	1.6	19	1.47	24	1.33	20	1.4
64	98	1.32	98	1.48	106	1.83	66	1.71	66	1.71	66	1.71
256	386	1.33	386	1.49	418	1.9	258	1.77	258	1.77	258	1.77

size	bwc						uneven spread					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	11	0.91	13	1.08	17	1.0	11	0.91	18	0.94	13	0.77
16	33	1.0	35	1.09	41	1.17	23	1.21	27	1.19	24	1.17
64	129	1.0	129	1.12	137	1.42	82	1.38	82	1.38	82	1.38
256	513	1.0	513	1.12	545	1.46	322	1.42	322	1.42	322	1.42

Partial Buffer 100

size	wc						even					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	7	1.0	10	1.0	12	1.0	10	1.0	17	1.0	12	0.83
16	18	1.0	19	1.0	19	1.0	19	1.37	24	1.2	19	1.37
64	66	1.0	66	1.0	66	1.0	66	1.47	66	1.47	66	1.47
256	258	1.0	258	1.0	258	1.0	258	1.49	258	1.49	258	1.49

size	bc						uneven					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	7	1.43	10	1.4	12	1.42	10	1.0	17	1.0	12	0.83
16	18	1.83	19	2.0	19	2.53	19	1.47	24	1.33	19	1.47
64	66	1.95	66	2.2	66	2.95	66	1.71	66	1.71	66	1.71
256	258	1.99	258	2.24	258	3.08	258	1.77	258	1.77	258	1.77

size	bwc						uneven spread					
	DTN/1		DTN/2		DTN/4		1 cycle		8 cycle		8 cycle pipeline	
	T	S	T	S	T	S	T	S	T	S	T	S
4	10	1.0	13	1.08	14	1.21	10	1.0	17	1.0	12	0.83
16	33	1.0	34	1.12	34	1.41	23	1.22	26	1.23	23	1.22
64	129	1.0	129	1.12	129	1.51	82	1.38	82	1.38	82	1.38
256	513	1.0	513	1.12	513	1.55	322	1.42	322	1.42	322	1.42

A.2. Circuit Size

size	slice	simple		ooo		I_{ooo}	p25		I_{p25}	p50		I_{p50}	p100		I_{p100}
		in	out	in	out		in	out		in	out		in	out	
4	LUT	84	30	149	123	1.55	-	-	-	396	153	2.70	357	483	3.38
	REG	253	82	308	114		-	-		431	238		400	278	
8	LUT	115	36	232	177	1.76	445	163	2.58	679	469	4.16	806	1423	6.77
	REG	274	87	359	133		465	250		627	356		719	512	
16	LUT	178	42	350	226	1.92	750	522	3.75	1361	1569	7.36	2038	4495	14.61
	REG	303	92	442	160		668	368		1001	596		1450	1002	
32	LUT	282	51	670	379	2.37	1499	1585	6.11	3463	4454	13.80	4350	15903	32.29
	REG	348	97	590	203		1058	608		1724	1092		2840	2032	
64	LUT	413	54	879	661	2.70	3621	4479	11.08	5365	15951	26.76	8488	64225	83.05
	REG	425	102	864	278		1814	1104		3159	2128		5616	4226	
128	LUT	741	89	1866	963	3.01	5753	16007	18.11	11194	64292	57.11	18893	-	-
	REG	566	107	1396	419		3314	2140		6008	4348		11152	-	
256	LUT	1334	118	3722	1884	3.64	12031	64360	36.13	24467	-	-	41974	-	-
	REG	845	112	2465	700		6306	4340		11693	-		22191	-	
512	LUT	3528	165	7713	3579	3.25	26488	-	-	65323	-	-	85758	-	-
	REG	1440	129	4571	1224		12285	-		22996	-		44296	-	