TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

**ALEKSI TERVO**

# OPTIMIZING TRANSPORT-TRIGGERED ARCHITECTURES FOR FIELD-PROGRAMMABLE GATE ARRAYS

Master's thesis

Examiner: D.Sc. Pekka Jääskeläinen

The examiner and topic of the thesis were approved on 15 August 2018

# ABSTRACT

**ALEKSI TERVO**: Optimizing Transport-Triggered Architectures for Field-Programmable Gate Arrays
Tampere University of Technology
Master of Science Thesis, 53 pages
Major: Embedded Systems
Examiner: D.Sc. Pekka Jääskeläinen

Keywords: transport triggered architectures, field programmable gate arrays, parallelism

With the growing importance of energy efficiency, heterogeneous computing has become more popular in recent years. Field-programmable gate array (FPGA) devices are no exception: offering highly parallel execution at low power, they are an option worth considering for many tasks, and increasingly more available for users through cloud computing services.

While FPGA devices offer a lower barrier to entry to logic design than integrated circuit design, they are still difficult to design for compared with instruction set processors. While tools exist for translating a high-level language description of an algorithm into an FPGA design, they still require expertise most software designers do not have.

One way around this problem is building soft processors onto the programmable logic as a programmability layer for sofware designers. Transport-triggered architectures (TTAs) are a promising avenue of research in this area for their simple implementation and inherently parallel programming model.

This thesis presents FPGA-centric optimizations for transport-triggered architectures and evaluation of these optimizations through synthesis. Together, these optimizations yielded between 20 and 30 percent reduction in logic utilization in the tested architectures while having little effect on the clock frequency. Additionally, the scalability of TTAs for more parallel workloads is evaluated with various configurations of a TTA vector processor as well as a convolutional neural network processor case study.

# TIIVISTELMÄ

**ALEKSI TERVO**: Siirtoliipaistujen prosessorien optimointi ohjelmoitavalle logiikalle
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: TkT Pekka Jääskeläinen

Avainsanat: siirtoliipaistut prosessoriarkkitehtuurit, ohjelmoitava logiikka, rinnakkaisuus

Energiatehokkuuden tärkeyden kasvaessa heterogeeniset laskenta-alustat ovat kasvattaneet suosiotaan. Ohjelmoitavat logiikkapiirit eivät ole tästä poikkeus: erittäin rinnakkainen suoritus matalalla energiakulutuksella tekee ohjelmoitavista logiikkapiireistä varteenotettavan vaihtoehdon moniin käyttötarkoituksiin, ja käyttäjille koko ajan helpommin saatavissa pilvipalveluiden kautta.

Vaikka näille laitteille suunnittelu on helpompaa kuin perinteisille sulautetuille logiikkapiireille, suunnitteluprosessi vaatii erikoistietämystä verratuuna käskykantasuorittimien ohjelmointiin. Työkalut, joilla voidaan kääntää korkean tason ohjelmointikielellä kuvattu algoritmi ohjelmoitavalle logiikkapiirille, käyttävät samoja kieliä kuin käskykantasuorittimille ohjelmointi, mutta nämäkin vaativat suunnittelijalta syvällistä ymmärrystä siitä, miten algoritmi kuvautuu logiikkapiiritoteutukseksi.

Yksi tapa kiertää tämä ongelma on rakentaa ohjelmoitavalle logiikalle käskykantasuoritin, jota ohjelmoijat voivat käyttää toteuttaakseen halutun algoritmin. Siirtoliipaisut suoritinarkkitehtuurit ovat lupaava tutkimussuunta tällä alalla niiden yksinkertaisen toteutuksen ja rinnakkaisen ohjelmointimallin takia.

Tässä diplomityössä esitellään siirtoliipaistujen suorittimien toteutusta ohjelmoitaville logiikkapiireille parantavia muutoksia, sekä näiden muutosten arviointeja synteesitulosten kautta. Näiden muutosten yhteisvaikutus näkyy pääasiallisesti prosessorien koossa: eri arkkitehtuurien koko pieneni 20-30 prosenttia, ilman suurta muutosta kellotaajuudessa. Tämän lisäksi siirtoliipaistujen suorittimien skaalautuvuutta rinnakkaisille algoritmeille tutkitaan siirtoliipaistujen vektorisuorittimien sekä syväoppimiseen suunnitellun suorittimen näkökulmasta.

# PREFACE

The work for this thesis was done at the laboratory of Pervasive Computing at Tampere University of Technolgy with funding from the HSA Foundation, as well as the ALMARVI and FitOptiVis projects.

I would like to thank Dr. Pekka Jääskeläinen for giving me a chance to work on interesting projects and the valuable learning opportunities that have come with that work, as well as his guidance and support regarding this thesis. I would also like to thank my coworkers, present and former, for a positive work environment and a helpful attitude. Finally, I would like to thank my friends and family for supporting me throughout my life and studies and during the writing of this thesis.

In Tampere, Finland, on 15 November 2018

Aleksi Tervo

# CONTENTS

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ALU | Arithmetic-Logic Unit |
| ASIC | Application-Specific Integrated Circuit |
| CNN | Convolutional Neural Network |
| FF | Flip-Flop |
| FPGA | Field Programmable Gate Array |
| FU | Function Unit |
| CU | Control Unit |
| HLS | High-Level Synthesis |
| IC | Interconnection Network |
| IP | Intellectual Property |
| LSU | Load-Store Unit |
| LUT | Look-Up Table |
| NRE | Non-Recurring Engineering |
| PC | Program Counter |
| PE | Processing Element |
| RAM | Random Access Memory |
| RF | Register File |
| RISC | Reduced Instruction Set Computer |
| RTL | Register-Transfer Level |
| SIMD | Single Instruction Multiple Data |
| SRAM | Static Random Access Memory |
| SVTL | Semi Virtual-Time Latching |
| TTA | Transport Triggered Architecture |
| VLIW | Very Long Instruction Word |

# 1. INTRODUCTION

Field-programmable gate array (FPGA) devices offer a low barrier to entry to logic design compared with application-specific integrated circuit (ASIC) design. They are more suitable for low volume production and prototyping, since they have lower non-recurring engineering (NRE) costs and can be updated in order to fix erroneous behavior in the design [81]. Cloud service providers, such as Amazon Web Services [3] and Huawei [23], have recently added servers with FPGA devices to their offering, further aiding the wider adoption of FPGAs.

FPGAs, however, do not remove the engineering difficulty of developing the logic circuit in the first place. While register-transfer level (RTL) design entry is done with a programming language, the paradigm of traditional hardware description languages is likely to be unfamiliar to software developers. Furthermore, the RTL approach is low-level and requires the designer to understand what kind of logic is inferred from the code they have written, and how this affects the performance and resource consumption of the synthesis result [74, pp. 5-11]. This is even more important in FPGA designs, as the routing resources and logic primitives are less flexible than their ASIC counterparts. The required skills are a rarity: According to a 2017 survey, software engineers outnumber computer hardware engineers 17 to 1 in the United States [69].

High-level synthesis (HLS) is one approach to improve the accessibility of hardware design for software engineers. Here, design entry is done in a high-level language such as C or C++ that is familiar to software programmers and operates at a higher level of abstraction than RTL design. However, HLS tools do not eliminate the need for hardware expertise: Better results can be achieved with an understanding how the algorithms map to the low-level implementation. [44]

Processors are another way to shift design effort from hardware engineers to software designers. They can be used either to manage the control logic for fixed-function hardware, or as primary processing elements in and of themselves. However, a programmability layer introduces inefficiencies in power, performance and utilization compared with fixed-function logic. This overhead can be reduced by tailoring the processor for the application at hand [25], but the high NRE costs of ASIC production mean processors implemented with these technologies have to be more flexible. Soft processors, i.e., processors implemented on FPGA fabric, can be customized more aggressively, as the device can be reprogrammed with a different processor if the application changes. Programming these processors will be more similar to the software programmer's existing experience, and they will not only use familiar languages, but use them in a familiar manner.

Transport-triggered architectures (TTA) have shown promise in the field of soft processors. Compared with a traditional operation-triggered architecture, TTA has a simpler implementation, leading to lower logic requirements and higher frequency. Furthermore, the instruction encoding describes explicit parallelism without requiring complex decoding logic. [37]

This thesis describes FPGA-specific optimizations for TTAs and extends previous work by exploring the TTA design space, focusing on different parallel programming models and their scalability on FPGAs. Chapter 2 presents an overview of common processor architectures and their approaches for high performance are reviewed. This is followed by an in-depth examination of the TTA paradigm in Chapter 3 followed by an overview of the toolset used in this thesis to aid the architectural exploration of TTAs. Chapter 4 begins with a description of the structure of modern FPGA devices, followed by an overview of the design methods targeting FPGAs and a comparison to ASIC technologies. In Chapter 5, the details of TTA soft processor implementations are discussed. Different implementations of the components of TTAs and guidelines for high-performance FPGA designs are presented here. These implementations are evaluated in Chapter 6. In addition to a comparison of the individual component implementations, parallel function unit configurations are examined, and a case study involving a TTA optimized for convolutional neural network execution is presented. Chapter 7 presents related work and compares the existing approaches to the work presented in this thesis. The thesis finishes with a discussion of what has been achieved in it and an examination of possible future research avenues in Chapter 9.

# 2. PROCESSOR ARCHITECTURES

There are a wide variety of processor architectures, with different use cases and different approaches on fulfilling their design goals. Understanding popular processor design ideas is clearly beneficial when designing similar processors, and while the processor architecture approaches presented here have originated from and seen most use in traditional operation-triggered architectures, many of these approaches can be applied to TTA design as well.

This chapter presents the basics of processor architectures and common ways to to achieving high performance. This is divided into instruction scheduling, ways to parallelize computation at a processor architecture level, and memory hierarchies. Finally, we cover the principles of application-specific processor design.

## 2.1 Instruction Scheduling

Dynamic instruction scheduling can be used to mask long instruction latencies, and increase the performance particularly with heavily branching code. It is used extensively in modern general-purpose processors, and allows for speculative execution of conditional branches, executing multiple instructions from one thread in parallel, and managing high-latency instructions by executing the following instructions while it completes, as long as the data dependencies between operations are satisfied. However, the required scheduling logic can be very costly. Indeed, this is the main disadvantage of a hardware-heavy scheduling approach [26, p. 222].

Even without dynamic scheduling, pipelined processor architectures require hazard detection and resolution logic to detect and avoid pipeline hazards, such as an instruction reading a value from the register file (RF) before the correct value has been written there by the previous instruction. Exposed datapath architectures alleviate the logic requirements by moving the work of resolving pipeline hazards to the compiler. By allowing direct control of the datapath and requiring the compiler to resolve some or all of the control logic for it, the hardware implementation can be simplified.

In addition to TTAs, which will be covered more thoroughly in this chapter, various exposed datapath architectures have been proposed, with a large variance in organization and scale. Some architectures use one-dimensional [82] or two-dimensional [7, 9, 54, 79, 80] arrays of small processing elements (PEs) with a software-controlled interconnect between them, while others have an organization resembling very long instruction word (VLIW) processors, with multiple PEs controlled by one instruction [80].

There are drawbacks to exposing the datapath at the architectural level. Most directly, since more of the implementation is described by the architecture, it cannot be changed radically to accommodate e.g. out-of-order execution. New versions of exposed datapath processors are more likely to break compatibility with their predecessors. This is undesirable in systems where a longer lifespan is a key concern, especially those with closed-source software developed for them.

Relying on static scheduling by the compiler limits the optimizations to the opportunities the compiler can guarantee. For example, if a function accesses two pointers simultaneously, reading from one and writing to the other, the compiler cannot necessarily resolve if the two pointers are equivalent. In this case, the two memory accesses have to be kept in the same order as they are specified in the program. It is possible for the compiler to produce code that checks during execution whether the accesses overlap, and choose between two versions of the code based on that, but this increases the resulting code size. A hardware-based dynamic scheduler can look at the two accesses when they are ready to be scheduled, and issue them accordingly based on whether or not they conflict.

Dynamic latencies can also be an issue. In order to resolve data dependencies between operations, the compiler needs to make some assumptions on how many clock cycles it takes for an operation to produce a result. However, certain operations, most commonly cached memory accesses, may take a nondeterministic amount of time. Some of this can be hidden by the architectural description of the processor, by assigning these operations a latency equal to the maximum latency of the operation. This negates the performance gain of a potentially shorter operation, such as a cache hit, and may introduce no-operation instructions into the instruction stream, thus raising the instruction memory size requirements.

Alternatively, the execution of other PEs can be halted while the operation result is being produced, either when the PE should produce the result, or when the result would be read by another instruction. This approach can improve performance when the longer latency executions happen less often and does not require inserting nonfunctional operations into the instruction stream, but requires more communication between PEs.

## 2.2 Parallel Organizations

A simple approach to improving performance is to connect multiple processors together, usually by sharing some or all of the memory space between individual processors. Each processor executes independently and communicates through the shared memory space. While multiprocessor systems give a theoretically linear increase in performance and logic utilization, communication between processors limits their performance in practice.

Another way to parallelize an application is to run multiple threads simultaneously on one processor. Even without additional resources for the execution of operations, switching between threads can mask long operation latencies. *Barrel threading* is a specific multithreading approach where the active thread is switched every cycle in a constant order. This is simple to implement in hardware, and can be used to reduce data forwarding between processor pipeline stages. For example, in a four-thread barrel threaded processor, each thread is only active every four cycles, so there is no need to pass data from one pipeline stage to the three pipeline stages preceding it, as they are executing different threads. Furthermore, the RF can be split into four parts, one for each thread. These RFs can be further simplified, as it might be possible to use the same port for two or more accesses in different pipeline stages through time multiplexing.

Exposing parallelism explicitly in the instruction encoding can be a more resource-efficient way to increase performance. This is especially popular for application-specific processors, where binary compatibility between processors is not as important. Like traditional superscalar processors, VLIW processors have multiple PEs in parallel, capable of executing different sets of instructions. Instead of the instruction scheduler extracting parallelism from an inherently serial instruction stream, VLIW machines have an instruction that specifies operations separately for each PE. In order to simplify the control logic between many PEs even further, VLIW processors are statically scheduled, with architecturally visible fixed latencies for all operations. As a result, the burden of resolving the hazard falls on the compiler instead of the hardware, simplifying the logic implementation. [26, p. 194]

A VLIW processor usually has a very complex RF. Traditionally, PEs have two operand ports and one result port. In order to execute, for example, a four-lane VLIW instruction in parallel, the PEs will read eight operands from the RF and store four results to it. Any memory structure with eight read ports and four write ports is very complex, but there are approaches to reduce the port count of the RF blocks. Most commonly, a complex RF can be partitioned into multiple simpler ones, e.g. an RF with four write and eight read ports can be split into two RFs with two write and four read ports each. While the register files still have the same number of total read and write ports and can therefore be used with four PEs with full utilization, this creates additional constraints for the compiler. For example, it cannot schedule an instruction that tells all four PEs to write to the same RF partition. The performance impact can be reduced with a register allocation scheme that takes the partitioning into account.

Single Instruction Multiple Data (SIMD) is another way to expose parallelism to the programmer. They processors execute the same instruction across multiple elements, similar to vector arithmetic. This execution model allows the PEs to perform work largely decoupled from each other. Communication between PEs is still necessary, but this is often done through memory rather than the register file, or by specialized instructions that transfer data between vector elements. However, programming SIMD processors can be more difficult: Even when the algorithm can be executed in vector operations, assembly level programming may be needed to utilize it [91]. On the other hand, compilers attempt to vectorize programs automatically when the processor can use those operations, and libraries with vectorized implementations of common algorithms are available.

Another way to increase the flexibility of parallel processing is the Single Instruction

Multiple Thread (SIMT) execution model, most often seen in graphics processing units [26, p. 288]. Here, multiple threads are executed simultaneously as a *warp*, usually in parallel on multiple PEs. The execution is more restricted than a multicore processor, however, as traditionally only one instruction can be issued at once. This means that when threads in a warp take different paths from a branch instruction, i.e., *diverge*, the SIMT processor will execute both paths serially, allowing execution on some of the threads while keeping the other threads idle. Lower divergence means more active threads on average and a greater performance.

The two main challenges with SIMT architectures are memory accesses and warp scheduling. When executing a memory operation during completely convergent execution, all threads issue one load. Handling many memory accesses serially is very slow, so parallelizing the accesses is preferred. However, multiport memory structures are expensive to implement, so this is usually done by a series of wide memory accesses.

Because the processing elements run different threads, simultaneous memory accesses do not necessarily have any relation to each other. In the worst case scenario, all active threads issue a load to addresses so far apart from each other that any wide memory access can only hit at most one of the requested memory locations. Some of these difficulties can be alleviated by a combination of caches and thread- or warp-local scratchpad memory, and by switching execution to another set of active threads or to another warp entirely while the memory accesses are done in the background. However, the best performance can only be reached by using an algorithm where simultaneous memory accesses to land next to each other.

The traditional approach to keeping track of warp execution is stack-based. Whenever warp execution diverges, the scheduler pushes a divergence token onto the stack. This contains information about which threads did not take the branch, and the program counter (PC) from where to continue executing those threads. This information will be used to return active status to those threads once threads that took the branch reach an instruction with a bit that signifies the end of the branch, i.e., the point where the originally diverged execution should resynchronize. Once the newly active threads reach the same instruction, the warp resynchronizes and continues execution. [18].

This method of warp scheduling is incompatible with the way mutexes are used to synchronize between threads in general purpose processors. If a thread tries to acquire a lock that is held by a different thread, it cannot progress forward until the thread unlock the mutex. If the thread holding the lock is in the same warp, it will not progress until the active threads reach a synchronization point. This results in a deadlock.

One way to handle warp scheduling that avoids this problem is to have each thread hold its own state, i.e., program counter and stack [68, p. 27]. The scheduler chooses one of the program counters to issue an instruction, and threads that have a program counter that matches what has been issued activate. Some logic is still needed to choose a PC such that divergent threads converge as soon as possible in order to reach a good performance. At its simplest, this method can be the thread that has the smallest stack pointer, or in the case of a tie, the smallest program counter [17]. This approach can also be used in a system where the scheduler can issue more than one instruction at a time [16].

It should be noted that the different approaches to parallelization do not necessarily exclude one another, and many combinations are complementary to the strengths of individual approaches. For example, it is possible to design a processor with a VLIW programming model that also has SIMD instructions, or a SIMT processor that uses a combination of parallel PEs and barrel threading to execute a warp.

## 2.3 Memory Organization

Memory organization is an important part of a processor system. In terms of performance, larger memories are slower to access than smaller ones. The main memory is often on a separate integrated circuit from the processor, introducing additional latency in the driving circuitry, and through the increased capacitance of the longer wires. In terms of power consumption, faster memories are more power-hungry, which encourages reducing the size of fast memories places an upper limit on how fast you can make the main memory.

Cache hierarchies are often used in general purpose processors in order to have tiers of small but fast memories close to the processors, backed up by a large but slow memory off-chip. When a cache receives a memory access request for the first time, it will forward it to the next level in the hierarchy. The response can be stored in the cache, possibly replacing an old value, so that an access to the same memory address can be fulfilled without accessing the higher levels in the hierarchy.

Another option is to have a non-uniform memory structure, where a portion of the memory is faster to access from a particular processor, or by having a separate address space for a local scratchpad. Unlike caches, this requires explicit software control and may as a result increase programmer effort.

## 2.4 Application-Specific Tailoring

When a processor is designed with a particular application in mind, rather than as a general-purpose processor, the design should take that into account in all of its parts. The approaches presented above may suit one application better than another, and it is important to understand the application, for example, in terms of how well it can utilize a SIMD organization, or how much memory is needed to contain the working data set at any time. For example, many image processing algorithms can be parallelized easily, as they essentially perform one operation on every pixel of an image, while some encryption algorithms may depend on the value of the previous iteration, leading to serial execution.

The instruction set provides another way of tailoring the processor. Most simply, the designer can start with a general-purpose instruction set and remove the operations that are not being used by the application. Alternatively, instructions can be added depending on the needs of the application. These can be constrained versions of general-purpose instructions, a common combination of instructions, or a custom instruction that is simple to implement in hardware but does not map well to software, such as specific bit-level transformations that may require many bitwise shift and logic operations in software, but only need a handful of logic gates and wires for the hardware implementation.

# 3. TRANSPORT-TRIGGERED ARCHITECTURES

Transport-triggered architectures are a mostly unexplored avenue in soft processor architectures. However, they have many features that make for a simple and efficient implementation on FPGAs, and initial work [37] has shown them to be competitive with vendor-supplied soft processor templates.

This chapter first presents the TTA paradigm, along with transformations from two common processor architectures into an equivalent TTA and comparisons between the approaches. Lastly, this chapter will cover the TTA-Based Co-Design Environment (TCE), specifically the configuration parameters of its TTA template and the default implementation of the TTAs that can be described by this template.

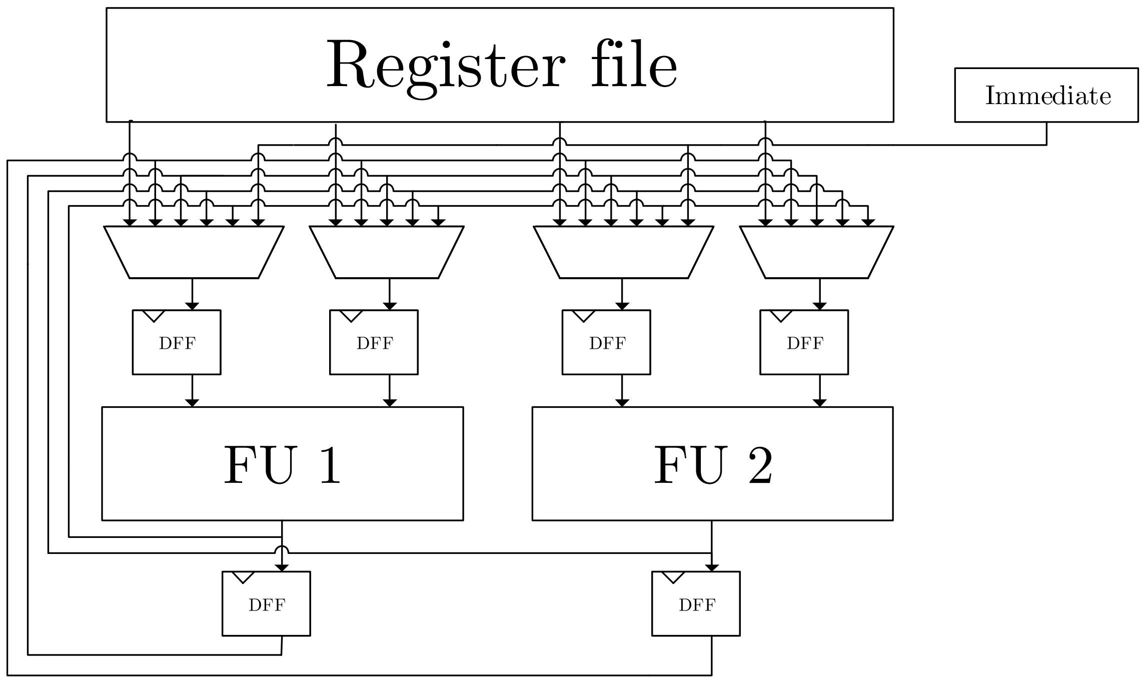
## 3.1 From VLIW to TTA

The TTA paradigm is often seen as an extension of VLIW architectures, and there are a number of similarities: Both are statically-scheduled processors which typically feature explicit parallelism by dividing the instruction word into independent partitions. Indeed, Corporaal [19, pp. 81-102] demonstrated how to transform a generic VLIW machine into a TTA, a process which will be revisited here.

The VLIW machine in Figure 3.1 exemplifies VLIW design. The instruction word is divided into multiple smaller instructions which execute independently on separate function units (FU). Operations have architecturally-visible latencies and some of the work of resolving hazards is moved to the compiler. The main downside of VLIW implementations, especially on FPGAs, is the register file, which is needed for the FUs to communicate with each other. Assuming a typical FU with two inputs and one output, the RF requires three ports for each FU.

There is a large degree of redundancy in the RF ports. Reaching a state where the FUs, and therefore also RF ports, are fully utilized requires a degree of instruction-level parallelism rarely seen in any program. Even then, there are many opportunities to eliminate RF accesses. Value bypassing is one form of this: If a value can be read directly from the result port producing it, a read access can be eliminated. While the bypassing itself is traditionally done in hardware, the programming model rarely restricts the programmer to a specific number of individual reads. The implementation must then either provide enough RF ports to satisfy worst case access patterns, or add costly dynamic scheduling logic for them.

It is possible to expose the bypass network to the compiler. Since the program can now directly specify which values are bypassed and which are read from the RF, the architecture



| FU 1 | | | | FU 2 | | | |
|---|---|---|---|---|---|---|---|
| opcode | src 1 | src 2 | dst | opcode | src 1 | src 2 | dst |

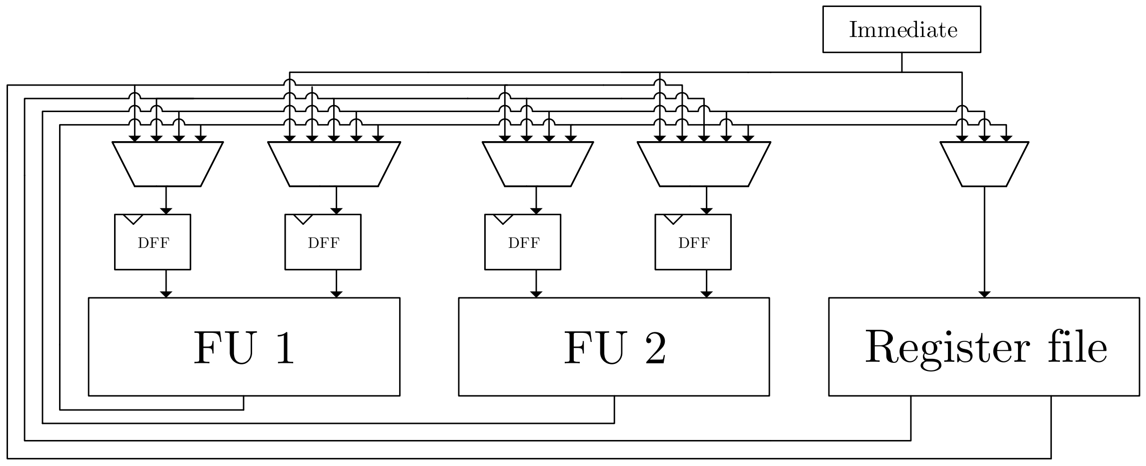
***Figure 3.1***. *A simplified VLIW architecture and possible operation encoding.*

can limit the number of direct RF accessess, enabling the removal of the RF port. A representation of this intermediate stage can be seen in Figure 3.2. The source field of each suboperation specifies explicitly where the operand should be sourced from, and the RF is given register indices separately from the FU suboperations. This gives rise to the main drawback of TTAs: The resulting operation is very verbose, and as such, the instruction word is even longer than with a similar VLIW architecture.

As can be seen by comparing Figure 3.1 and Figure 3.2, the exposed datapath version of the same VLIW machine has one extra multiplexer, located before the register file. This multiplexer is due to the smaller number of RF ports, and the resulting need to feed one RF write port with values from both function units. In the common case where read ports are more numerous than write ports, the multiplexer and RF together are strictly cheaper than the equivalent multiport RF, if it is built out of single write port memory primitives using the live value table method [42]. Such a RF would contain an equivalent multiplexer at each of the read ports to select the most recent value from the register banks, and would use more memory primitives than the register file with fewer ports.

While this architecture supports almost all of the cases presented by Corporaal where RF accesses can be eliminated, its scheduling is very limited. As the operands of an operation are specified all at once, they are constrained to be read simultaneously, reducing the opportunities to bypass values from the results of previous cycles.

For further scheduling freedom, the data transports to operand ports can be presented separately, and operations are started when a transport is issued to a predefined trigger port.



| FU 1 | | | FU 2 | | | RU |
|---|---|---|---|---|---|---|
| opcode | src 1 | src 2 | opcode | src 1 | src 2 | src |

***Figure 3.2**. Exposed-datapath VLIW.*

This is where the TTA paradigm gets its name, and its primary difference from traditional operation-triggered architectures (OTA). Since the values of the operands have to be stored while waiting for the operations to start, the architecture needs registers in the operand as well as the result ports. By decoupling the data transfers of an operations operands and result values, this design allows for the final optimization presented by Corporaal, where successive operations in the same FU can reuse one of the operands. The resulting TTA with VLIW-like connectivity can be seen in Figure 3.3.

This interconnection network (IC) design is impractical for large designs, as the number of buses and thus multiplexer inputs and instruction width grows linearly with the number of FUs. While adding buses allows the programmer to schedule more instructions in parallel, there is a limit to the amount of instruction-level parallelism in any given program. Optimizing the IC by merging buses that are rarely used together and removing rarely used connections both simplifies the hardware implementation and reduces the width of the instruction word.

Figure 3.4 shows an example of a practical TTA design. While most of the output ports are connected to all of the buses, the connectivity of the IC has been significantly reduced from a fully-connected or a VLIW-like IC, and e.g. moves loading an operand to the load-store unit (LSU) and triggering a jump cannot be issued at the same time, as they would use the same bus. However, the operand and trigger ports of the LSU and the ALU can be used independently of each other.

The above TTA has two general-purpose register files, as an example of RF partitioning, used for the same purposes as with VLIW architectures. In addition to the general-purpose RFs, the processor has a boolean register file. The values are used as predicates for conditional execution, used either for masking unwanted branch moves or for a series of conditionally-executed moves, e.g. representing both sides of an if-else structure. As

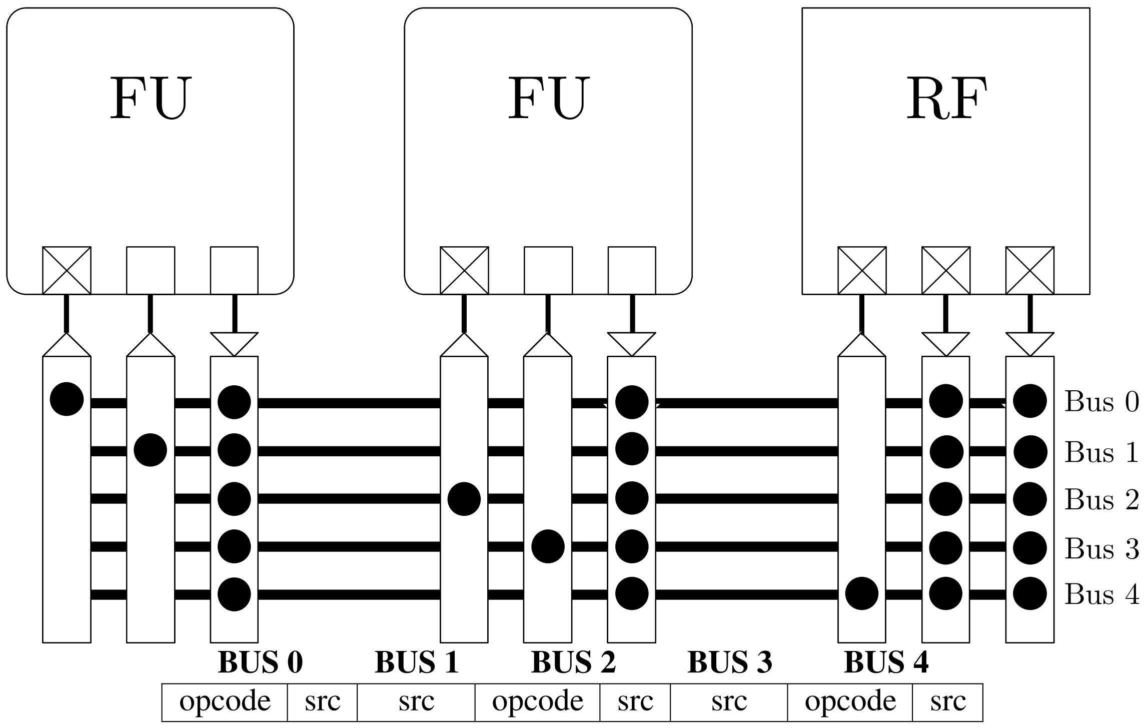


***Figure 3.3**. VLIW-connected TTA and possible instruction encoding.*

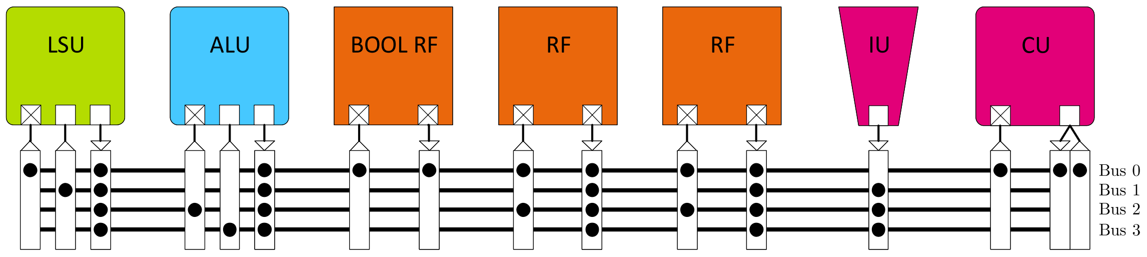


***Figure 3.4**. An example of a TTA design.*

branching is expensive on TTAs due to being statically scheduled architectures, the latter can be significantly faster.

While the primitives are less flexible, the lower NRE costs of FPGAs allow for a more targeted design than ASIC. This enables more aggressive tailoring of a soft processor to be more aggressively tailored for the application at hand. TTAs benefit from this, as they have a larger design space than e.g. VLIW machines, due to the highly customizable interconnect network. In particular, pruning the IC of less frequently used connections is of particular benefit, as this has a direct impact on the inferred multiplexer size.

## 3.2 From Scalar OTA to TTA

While this example started from a VLIW architecture, it should be noted that the TTA paradigm is sufficiently flexible to transform e.g. RISC or SIMD processors similarly, and VLIW was chosen for the similarities in its programming model, and to highlight the scalability advantages of TTAs in a multi-issue case. For example, the above transformation

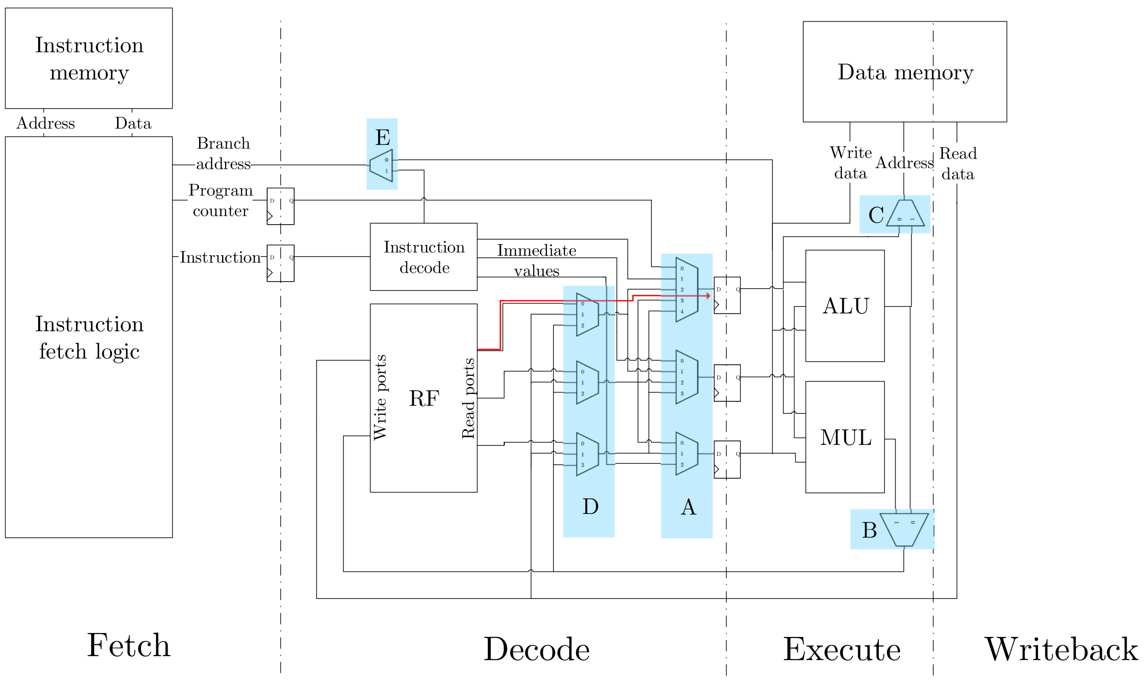


***Figure 3.5**. A scalar OTA design with a four-stage pipeline and data forwarding [73], with the critical path through multiplexers' datapath marked with a red arrow. Most control signals have been omitted for clarity.*

to TTA can be repeated starting with the real-world 4-stage single-issue scalar RISC design developed for the PULP platform [73]. The pipeline and its forwarding paths can be seen in Figure 3.5.

There are a couple of reductions we can make to the connectivity of this pipeline when translating it to a TTA processor without losing any functionality. First, we can eliminate the return path from the operand C pipeline register. It is only used for conditional branches as a delayed version of the immediate branch address signal, which is the other signal at the multiplexer E input. In a statically scheduled processor, we can simply load the address when the ALU is ready to provide the guard value for the branch.

Furthermore, bypass multiplexers (marked as D) can be merged with the operand multiplexers (marked as A) by connecting the signals from the execution and writeback stages to the decode stage to each of the operand multiplexers (A). The transformation to TTA also removes the bypass control logic, as the compiler will decide between a bypass or a register read statically. Especially for architectures with many RF ports, this can save a significant amount of logic: Even at its simplest, bypassing requires a comparison between each read port address and each write port address in order to select between the value from the RF and the values of the writeback connections. Lastly, moving multiplexer B to the decode stage and connecting the ALU and multiplier outputs independently to the multiplexers will make the transformation to a TTA interconnection network simpler.

The resulting pipeline can be seen in Figure 3.6. From here, the transformation to a TTA is straightforward. The multiplexers in the decode stage can easily be made to correspond

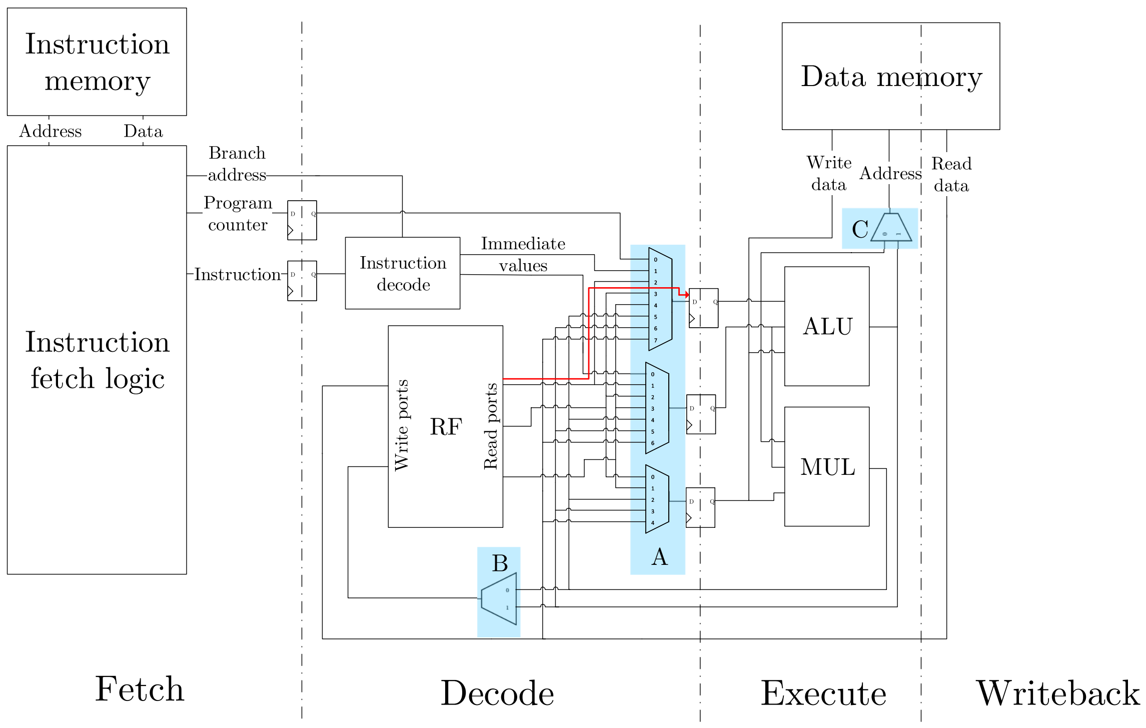


***Figure 3.6**. Modified four-stage scalar OTA pipeline, with the critical path through multiplexers marked with a red arrow.*

to four buses, whereas the multiplexer in the execute stage can be described by an input socket connected to one of the buses and another bus with one connection to the ALU. The other LSU operands, memory data input, and the ALU and multiplier operands can be sourced directly from the busses describing the decode stage multiplexers. The two direct connections, from the LSU to the RF and from the instruction decoder to the instruction fetch logic, can be simple two-connection buses. The TTA with equivalent connectivity to the modified OTA pipeline can be seen in Figure 3.7.

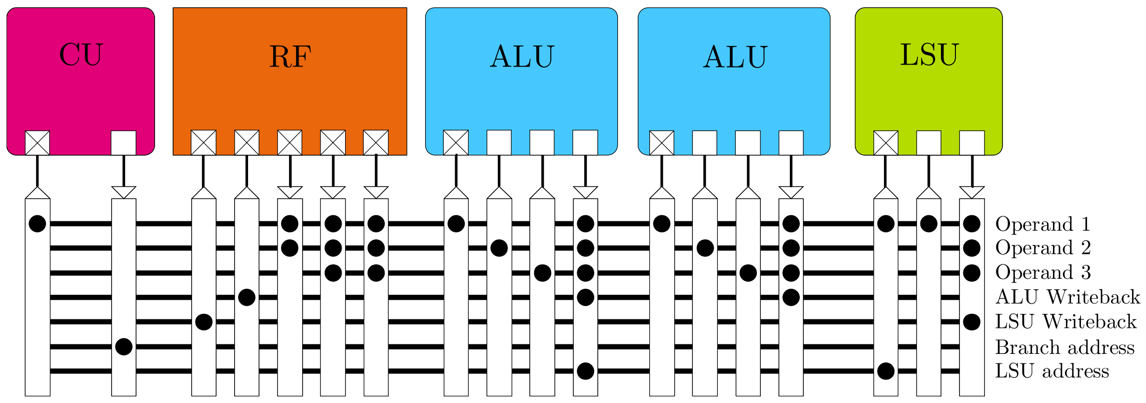


***Figure 3.7**. A TTA architecture with the connectivity of a four-stage scalar OTA pipeline with bypassing.*

Modeling each multiplexer as being built out of tiers of two-input multiplexers, we can get an approximation of the logic utilization of the multiplexers in each processor design. For example, a four-input multiplexer has two tiers of simple, two-input multiplexers, one with two multiplexers cascaded with the next layer with a single simple multiplexer, for three multiplexers total. Every time a signal is added, one of the input signals can be replaced by the output of a multiplexer, connected to the original input and the added signal. Therefore, a multiplexer with $N$ inputs requires $N-1$ simple multiplexers to build, and has $\lceil \log_2 N \rceil$ levels of multiplexers along its critical path.

The original pipeline datapath has nine multiplexers: three with two inputs, four with three inputs, one four-input and another five-input multiplexer. These multiplexers can be built out of 18 simple multiplexers. If only the multiplexers are considered, the critical path goes through one bypassing multiplexer and the widest operand multiplexer, with a total of five levels of simple multiplexers. The TTA conversion has five multiplexers, two of which have two inputs, and the three remaining ones have five, seven and eight inputs. This corresponds to 19 simple multiplexers. Because they are not cascaded, the critical path of the multiplexers is simply the critical path of the largest multiplexer, with three tiers. Both of these critical paths have been marked in Figures 3.5 and 3.6, respectively.

From this model, we can assume that the translation from an OTA to a TTA does not increase the logic utilization for the multiplexer components. However, in the absence of dynamic operation latencies, or when the dynamic latencies can be treated as static, the TTA implementation does not require additional logic to resolve pipeline hazards, as these can be resolved by the compiler. Furthermore, the instruction decoding is simpler for TTAs, as the instruction word maps more directly to the multiplexer control signals. The eliminated logic contributes alongside the removed multiplexer cascading to a higher maximum frequency, and better performance.

This suggests that TTAs are also suitable as small soft processors, a hypothesis backed by previous work [37], where a small TTA architecture improved the performance of minimally configured Microblaze soft processors by over 100 % on average while only increasing LUT utilization by under 25 %. These small, simple processors could be used on their own as faster control-oriented cores or as processing elements in a SIMT architecture.

## 3.3 TTA-Based Co-Design Environment

TTA-Based Co-Design Environment [38] is a toolset for hardware-software co-design of TTA processors. While it also features a graphical interface for processor design, a retargetable compiler based on the LLVM project [52] and an instruction set simulator, among other tools, this section will only cover the tools relevant to this thesis. Specifically, it will cover the TTA template used by TCE and the RTL implementation produced from architectural descriptions using the template. In the following chapters, this thesis will propose modifications to this implementation. These modifications have been integrated in TCE and TCE is used in the experimental portion of this thesis.

### 3.3.1 TTA Template

As the TTA architecture represented in Figure 3.4 was designed with TCE tools, it is also an example of the TTA template [15] used by TCE. In this template, a TTA processor consists of one or more FUs and RFs connected together by an IC. A FU has one or more input port and can have any number of output ports. One of the input ports acts as a trigger port for all operations of the FU. The operands and results can be bound to any of the available input and output ports, respectively. While the operand writes and result reads can be assigned at any point of the operation execution, most operations read all operands on the first cycle and produce all results on the last cycle, which defines an unambiguous static latency for the operation.

Each machine may have one control unit (CU), which controls the instruction fetch logic through branching and subroutine call instructions. While the pipelining details of the instruction fetch and decode stages are implementation-defined, the CU defines branching latency, and therefore the number of delay slots.

The IC is composed of one or more buses, each of which corresponds to one move in the instruction word. On any bus, a move can be issued between any two source and destination sockets connected to that bus. Any socket can be connected to any combination of buses and to any number of ports although most designs have each socket connected to only one port.

Each bus may have a short immediate value of arbitrary width with either zero or sign extension to the width of the bus. For conditional transfers, each bus may have a set of guard registers they can use to conditionally execute any move, with optional support for inverting the value of the guard register.

### 3.3.2 Hardware Generation

There are a number of parameters that can be given to the processor generator that would change the RTL description. Likewise, the CU has a variety of optional features, such as a loop buffer and support for variable-length instructions. These configurations are outside the scope of this thesis; instead, the following description will focus on the result of the default parameters.

The instruction fetch stage is straightforward. The next instruction address, either incremented from the current one or sourced from the operand of an active control operation, is issued to the instruction memory. Unless the busy signal of the memory interface is asserted, the instruction word is available in the next instruction cycle, and is forwarded to the decode stage through a register, which can be bypassed according to a CU parameter. The instruction decode stage drives the control signals of the interconnect and the load and opcode signals of the function units through an optional register. The instruction fetch

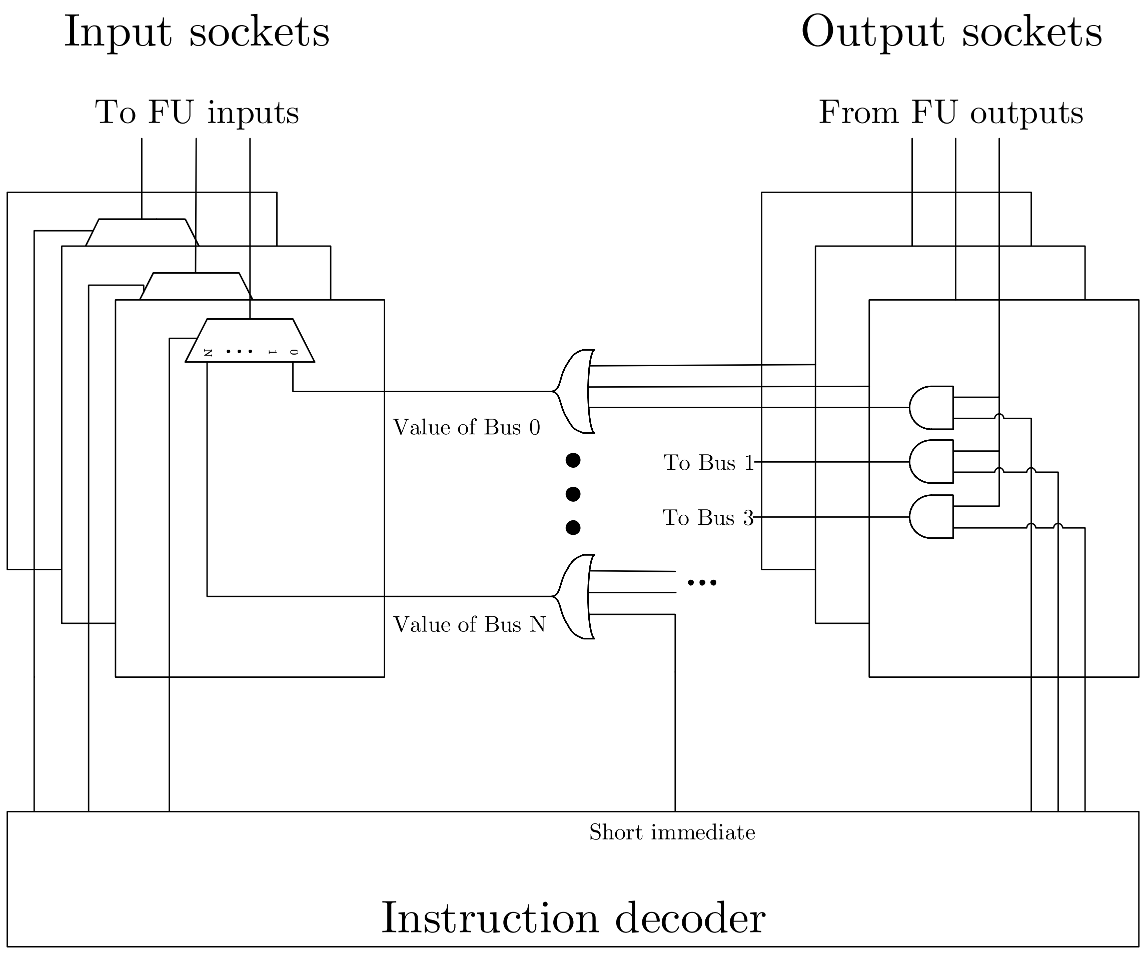


***Figure 3.8**. And-or interconnect network.*

and decode registers as well as the instruction memory latency determine the architectural latency of the branch instructions and, by extension, the number of delay slots.

Prior to the work described in this thesis, the IC has been implemented as an and-or network, shown in Figure 3.8. The instruction decoder drives a control signal for every input socket decoded from the source field of each transport in the instruction word. This selects which outputs the input socket data is forwarded to, and which are kept as an all-zero signal. For each bus, the input socket outputs for that bus and the short immediate from the instruction decoder are combined with an OR port. As only one of these signals may be active at a time, that signal is driven to the output of the OR port. These signals are connected to the output sockets of each bus, which selects the appropriate signal with a multiplexer, controlled by the instruction decoder.

Function unit and register file implementations are picked from a hardware database. Function units can also be constructed from VHDL or Verilog snippets describing each operation. In order to handle dynamic latencies, e.g., when connecting an LSU to a cache, function units can raise a global lock, stalling execution until the value is available.

The TTA template requires the function units to implement semi virtual-time latching

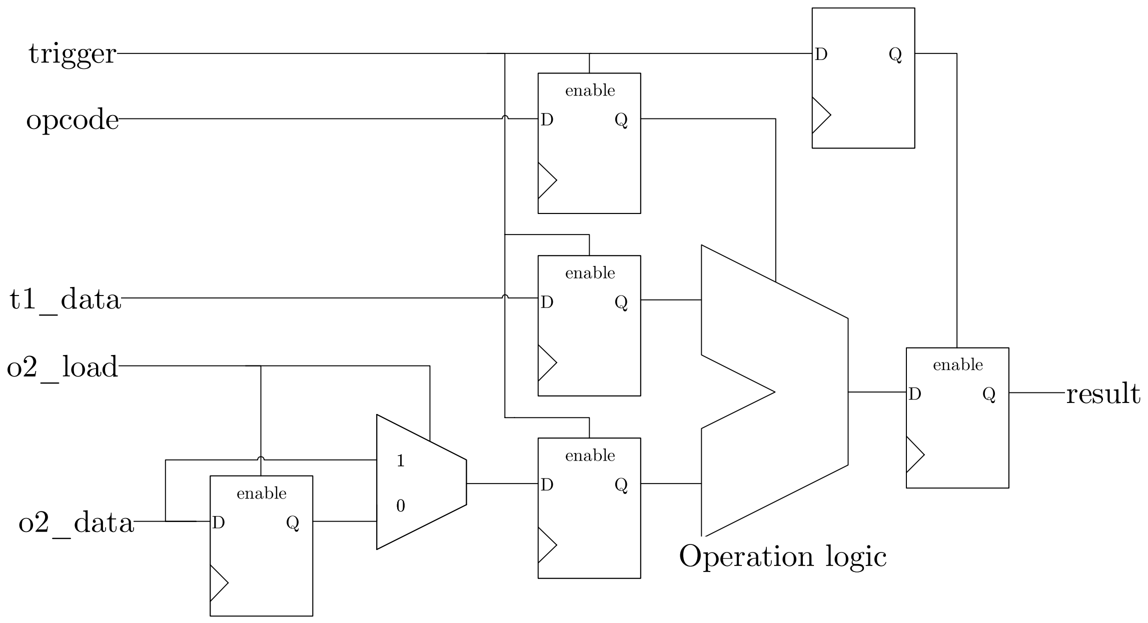


***Figure 3.9**. A possible implementation of semi virtual-time latching.*

(SVTL), which is one of the approaches presented by Corporaal [19, 335-345]. While the implementation details are left to the designer of the FU, one possible implementation can be seen in Figure 3.9. The operand ports can store a value for any number of cycles, while only exposing it to the operation logic once an operation is triggered. As such, the result of the operation will not change due to a changed operand and the value of the previous result can be used for longer.

# 4. FIELD PROGRAMMABLE GATE ARRAYS

Field-programmable gate arrays are a widely-used component in high performance application-specific processing. The reconfigurability of FPGA devices allows for thorough customization of the executed logic, which can reduce the power consumption dramatically. However, compared with traditional processors, they are more difficult to design for, and usually require experience in hardware design that a software engineer does not have.

This chapter first presents an examination of modern FPGA architectures with a focus on the components most important for a processing system: fine-grained programmable logic and coarse-grained random access memory (RAM) and arithmetic blocks. This is followed by an overview of the methods used to design for FPGAs, and a comparison to ASIC implementations, focusing on soft processors.

## 4.1 Architecture

Soft processor design requires a detailed understanding of FPGA technology. Modern FPGA devices offer a variety of resources to implement the desired functionality, an overview of which can be seen in Table 4.1. The four presented vendors make up the overwhelming majority of the FPGA market [20].

### 4.1.1 Logic and Arithmetic

In the presented device families, look-up tables (LUTs) comprise the numerical majority of the logic components. These are the main components used to implement arbitrary logic functions. All device families also offer some amount of fixed logic connected to the LUTs; at a minimum, the carry logic for a ripple-carry adder is present to speed up addition.

In most device families, LUTs feature four inputs and one output. As Intel and Xilinx have moved on to larger LUT sizes, they have adopted LUT architectures that can be used as multiple smaller LUTs, in order to avoid spending all of a large LUT on a smaller logic function. For example, modern Xilinx devices feature LUTs capable of implementing a 6-input, single output logic function, or any 5-input, 2-output logic function, which allows the LUT to implement two smaller, e.g. 3-input and 2-input independent logic functions.

While LUTs are the most versatile component in FPGAs, they are an inefficient way of implementing complex arithmetic. Multiplication followed by addition or accumulation is very common in digital signal processing (DSP) algorithms, such as convolutional

*Table 4.1*. *Memory and arithmetic resources by FPGA family. R = read, W = write, and RW = read-write.*

| Vendor | Device family | LUT | | Hard multiplier | | Dual-port | Other RAM | |
|---|---|---|---|---|---|---|---|---|
| | | inputs | Outputs | Integer | Floating point | SRAM size | Ports | Size |
| Intel | MAX 10 [32] | 4 | 1 | 18b x 18b | None | 1024x9b | None | |
| | Cyclone 10 LP [30] | | | | | | | |
| | Cyclone 10 GX [29] | 8[1] | 4[1] | 27b x 27b + 64b | Yes[2] | 1024x20b | 1 R + 1 W | 32x20b |
| | Arria 10 [28] | | | | | | | |
| | Stratix 10 [35] | | | | | | | |
| Xilinx | 7 Series [84–86] | 5[3] | 2[3] | 25b x 18b + 48b | None | 1024x36b | 3 R + 1 RW[4] | 256x1b[4] |
| | Ultrascale [88–90] | | | 27b x 18b + 48b | | 1024x36b or 4096x72b | 7 R + 1 RW[4] | 512x1b[4] |
| Microsemi | Polarfire [60] | 4 | 1 | 18b x 18b + 48b | None | 1024x20b | 1 R + 1 W | 64x12b |
| | IGLOO2 [62] | | | 18b x 18b + 44b | | 1024x18b | 2 R + 1 W | 64x18b |
| | RTG4 [61] | | | 18b x 18b + 44b | | 2048x12b | 2 R + 1 W | 128x12b |
| Lattice | iCE40 UltraLite [48] | 4 | 1 | None | None | None | 1 R + 1 W | 256x16b |
| | iCE40 Ultra [47] | | | 16b x 16b + 32b | | None | 1 R + 1 W | 256x16b |
| | iCE40 UltraPlus [47] | | | 16b x 16b + 32b | | 16384x16b[5] | 1 R + 1 W | 256x16b |
| | CrossLink [45] | | | None | | 1024x9b | None | |
| | MachXO3 [50] | | | None | | 1024x9b | 1 R + 1 W | 16x4b |
| | ECP5 [46] | | | 36b x 18b + 54b | | 1024x18b | 1 R + 1 W | 16x4b |

[1] Not a general 8-input LUT, as any one output cannot depend on all 8 inputs
[2] Single-precision floating point arithmetic can be implemented entirely by hard multipliers, double precision arithmetic requires an external adder
[3] Can be combined into a 6-input LUT with a single output
[4] Port count depends on implemented memory size, e.g. 3 R + 1 RW is only available in configurations up to 64x1b in Xilinx 7 Series FPGAs
[5] Only one RW port

and finite impulse response filters. For these tasks, multipliers implemented as fixed-function logic or with reduced configurability, i.e., hard multipliers, are very common in FPGA devices. Only the very low end iCE40 UltraLite family and the bridging-focused CrossLink and MachXO3 families from Lattice completely omit hard integer multipliers. The multipliers usually feature built-in adders for accumulation of the multiplication result. High-end Intel devices additionally offer floating point arithmetic support in their hard multipliers.

### 4.1.2 Routing Resources

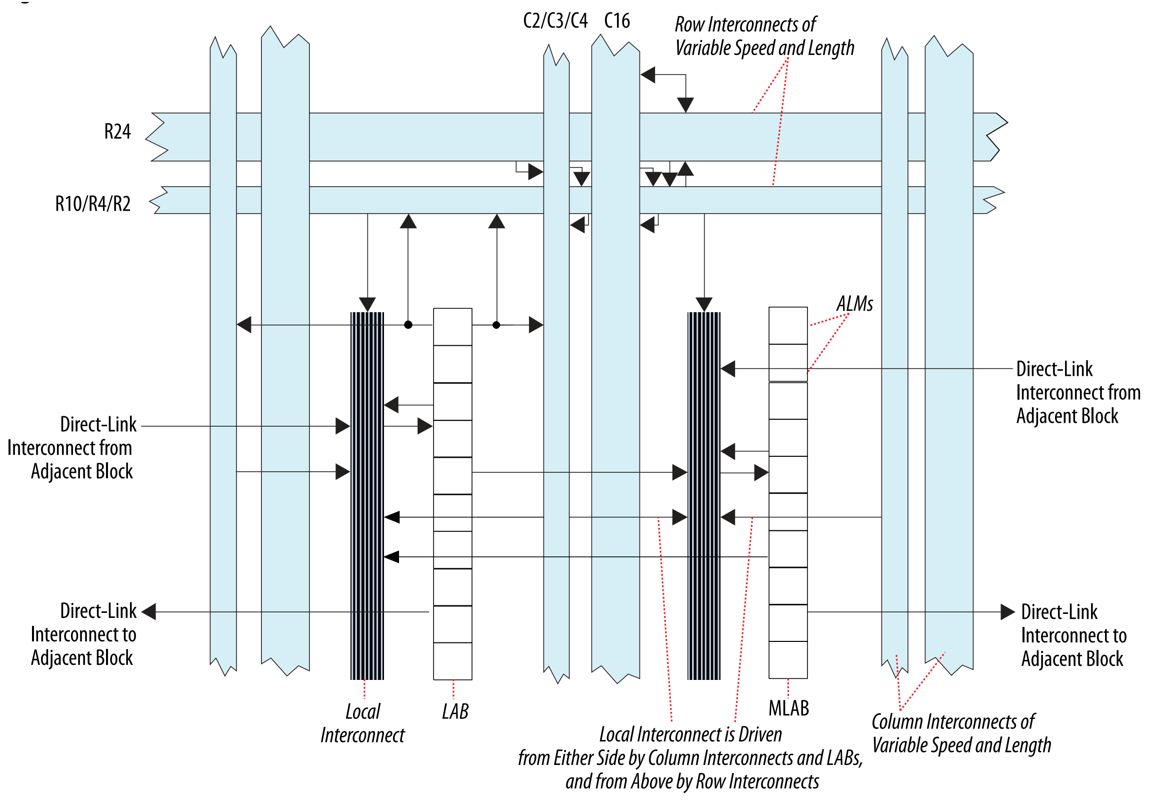

C2/C3/C4
C16
Row Interconnects of Variable Speed and Length
R24
R10/R4/R2
ALMs
Direct-Link Interconnect from Adjacent Block
Direct-Link Interconnect from Adjacent Block
Direct-Link Interconnect to Adjacent Block
Direct-Link Interconnect to Adjacent Block
Local Interconnect
LAB
MLAB
Local Interconnect is Driven from Either Side by Column Interconnects and LABs, and from Above by Row Interconnects
Column Interconnects of Variable Speed and Length

***Figure 4.1**. A high-level description of the routing resources in Intel Stratix 10 devices [36]*

In order to connect these elements together, FPGAs use a routing network composed of wires connected by switches, which can be configured to the desired state when the FPGA device is programmed. An example of this can be seen in Figure 4.1. While the exact details of the implementation of the routing networks, such as the number of wires and the connectivity of the switch circuitry, differ between vendors and device families and are not always available to the public, there are some general rules that most devices seem to follow.

Much like the logic resources, the most numerous of the routing resources are the general-purpose ones, with a high degree of connectivity through the switches. However, there are also fixed connections between the logic elements. One example of this is the carry path that can be used to implement wide addition. Similar signals for cascading can often be found in the hardened arithmetic blocks.

### 4.1.3 Storage Elements

The smallest and most flexible storage components are one-bit sequential components connected to LUT outputs. Usually, these elements are flip-flops (FF) although they can also be configured as latches in Xilinx devices and the Lattice MachXO3 family. The hardened arithmetic blocks also have registers that can be used for pipelining the arithmetic operations or bypassed, available at a few fixed positions in the arithmetic operation pipeline.

On-chip memory is an important part of a processor-based system. A memory hierarchy with multiple levels is an integral part of modern computers, and the gap between computational resources and memory bandwidth keeps growing [26, pp. 72-74]. While these memories can be built out of individual registers in the FPGA fabric, this is inefficient except for the smallest register files (RF).

Most device families have memory primitives to build small memories, suitable for RFs. These smaller memories are less flexible than individual registers, but are faster and more resource-efficient while still providing multiple read ports, especially in Xilinx devices. Their greatest limitation is that even the high-end UltraScale LUT-based RAM is restricted to a single write port.

For larger memories, such as caches or local scratchpad memories, most device families supply SRAM (static RAM) blocks with two bidirectional ports. The depths and word sizes of these memory blocks vary between vendors and device families. The word widths of the memory blocks are rarely a power of two, as would be conventional in computer systems, but tend to have widths slightly over the nearest power of two for storing e.g. checksums.

Since the placement of the logic is more restricted than ASIC, especially for hardened blocks, and the routing is less flexible, the path of a signal through the routing network can contribute a significant portion of its delay between registers. In addition to the registers next to the LUTs, Intel Stratix 10 devices also contain registers in the routing network, which gives the synthesis tool more freedom in register placement [31].

### 4.1.4 Reset

Reset is an important part of a synchronous system. FPGAs offer a wider variety of reset options compared with ASIC technologies. In particular, registers and memory can be set to a known state when the device is programmed [11, 34, 51]. This is a resource-light approach, as it does not require general routing or logic resources, but is only applied once. Similarly, a global reset driven by internal logic or an external signal uses dedicated routing, but it is inflexible and as such unsuitable on its own for designs which require a partial reset.

Reset functionality for LUT FFs and hardened component pipeline registers varies between device families. Most commonly, both asynchronous and synchronous resets are offered. However, while Xilinx devices have LUT FFs with a configuration option supporting both reset methods for their LUT FFs, memory and arithmetic block pipeline registers can only be reset synchronously. Intel FPGA devices only offer asynchronous reset, emulating synchronous reset with additional logic in the data path. In this case, the reset signal would essentially AND-gate the datapath signal to zero. This increases the number of inputs to each transfer function by one, increasing the LUTs required to implement them.

## 4.2 Designing for FPGAs

The traditional tool for implementing an algorithm on an FPGA is RTL design. However, this is a very low-level design method, and as discussed, RTL design languages have a very different programming paradigm from most software programming languages despite the surface-level similarities. This makes it inaccessible to software-focused programmers.

High-level synthesis tools attempt to bridge this gap. They can transform an algorithm described in a software programming language, such as C, into a description the FPGA synthesis tools can use for an FPGA design. However, in addition to the usual program description, these tools often introduce tool-specific directives the programmer can use to direct the tool in terms of how it implements the operations or unrolls and pipelines the logic on the FPGA. Using these is essentially mandatory for high-performance designs, and require knowledge of the underlying architecture and how to implement logic most efficiently for it.

Overlay architectures are another way to map algorithms on FPGAs. They are a collection of coarse-grained components that can be implemented on top of the FPGA fabric, to allow for a higher level of abstraction for the programmer. The components can range in complexity from blocks resembling the FPGA fabric itself [8, 10] to ones incorporating arithmetic units [39] to soft processors, i.e., conventional processors implemented on the FPGA fabric.

Soft processors are an important tool in an FPGA designer's toolkit. The flexibility and relative ease of programming a soft processor has attracted a wide variety of approaches to processors, from conventional RISC designs to architectures designed around run-time reconfigurability. However, processor design is almost always done at a low level of abstraction, and the underlying FPGA architecture and its differences with ASIC technologies have to be taken into account when designing and implementing a soft processor. This means that while programming the processors is simple, designing them falls outside the skill set of a software programmer.

## 4.3 Comparison to ASIC

Wong et al. [83] performed a quantitative comparison between ASIC and FPGA-based processor components, carried out with the Intel Stratix III device. This is an older high-end device family with features between the two tiers of Intel devices presented in Table 4.1, with 6-input LUTs and additional memory block sizes in addition to the 9216-bit memory blocks of the MAX 10 and Cyclone 10 LP families.

Wong et al. found that, in general, the arithmetic which has specialized logic for them in the FPGA fabric area penalties compared with their ASIC implementations than other components. In particular, multipliers and adders fared well in the comparison. On the other hand, multiplexers had very high area and delay ratios. Newer FPGA architectures may perform better in this respect: Series 7 FPGAs from Xilinx incorporate multiplexer logic alongside the LUTs, and the larger LUTs of the higher-end Intel devices should be able to implement multiplexing logic more efficiently.

Simple, single-ported memories were another highly area-efficient component. However, complex memory structures, such as multi-ported memories for register files and content-addressable memories for processor caches required more resources than the simpler memories, as they could not be implemented directly with the FPGA memory primitives. As such, Wong et al. recommend building large caches with low associativity so that content-addressable memories are not required. Off-chip bandwidth was observed to be worse than for fixed-function circuits by 30 to 50 percent, which can be alleviated with caching the required data.

As expected, the hardened memory and arithmetic blocks feature heavily in efficient FPGA designs. While the synthesis tools are capable of mapping generic RTL code to hardened components, particularly multipliers, automatically, i.e. *inferring* them, this gives the synthesis tool significant latitude in choosing the appropriate organization and whether to optimize for fewer utilized LUTs, fewer utilized hardware multipliers or greater performance.

The synthesis tool has a large amount of information at its disposal and the implementations of common logic circuits have most likely been optimized to the underlying resources device by the FPGA vendors. While this could be expected to lead to better outcomes than a human designer, the implementation of the synthesis tool often requires adjustment for better results in the author's experience. This may be due to the synthesis tool prioritizing the optimization metrics differently from the designer.

The alternative approach, explicitly *instantiating* the necessary hardware resources and configuring them to the designer's preferences, requires more effort, but can lead to better results overall. In addition to hardened arithmetic resources, efficient implementation of, for instance, bit-shift operations can require the explicit use of the logic intended for multiplexing and addition. [66]

# 5. IMPLEMENTED OPTIMIZATIONS

The default RTL description generated by the TCE toolset has been primarily optimized for ASIC designs. In the process of optimizing TTAs for FPGA designs, some structures were found to be inefficient. This chapter details the changes made to each component of the processor and lays out guidelines for FU design for a more efficient FPGA implementation.

## 5.1 Instruction Fetch and Decode

The instruction fetch and decode stages of the examined TTA template are very simple: by default, only absolute jumps and calls are supported, and the instruction word is simple to map to the control signals. The primary change for these stages is the reset organization.

As discussed, there are more options for reset in FPGAs than there are for ASIC. However, the high performance target limits the choices considerably. In order to scale up a TTA design to large FPGA devices, a multiprocessor organization is likely needed. In such a design, the ability to reset processors independently is desirable, e.g. if different processors are executing different workloads. This eliminates global reset structures and configuration-time reset from the list of possible choices.

For local reset, there are still two options, namely, asynchronous and synchronous. The choice is straightforward for Intel FPGA devices: because the registers only support asynchronous reset, synchronous reset would introduce extra logic in the data path and affect clock rates and utilization negatively. In Xilinx devices, LUT FFs can be reset either asynchronously or synchronously, but the hardened arithmetic and memory components only support synchronous reset, and synchronous reset is recommended so that the synthesis tool has the option to map some of the processor functions to the hardened blocks.

## 5.2 Interconnection Network

A complex interconnection network can be the largest individual component in a TTA processor, and it may affect the critical path within any function unit as FU logic is moved across the registers to the IC or vice versa. Therefore, its efficient implementation is paramount to a high-performance TTA design. The default implementation, described in Chapter 3, did not map efficiently on to FPGA hardware.

For the FPGA implementations, the input socket side of the IC, with an AND-OR network performing what is essentially a multiplexing operation, was replaced with a switch-case structure in the RTL code. This describes a single multiplexer per bus, as shown in Figure 5.1. In addition to mapping better to the dedicated multiplexing logic of the LUTs

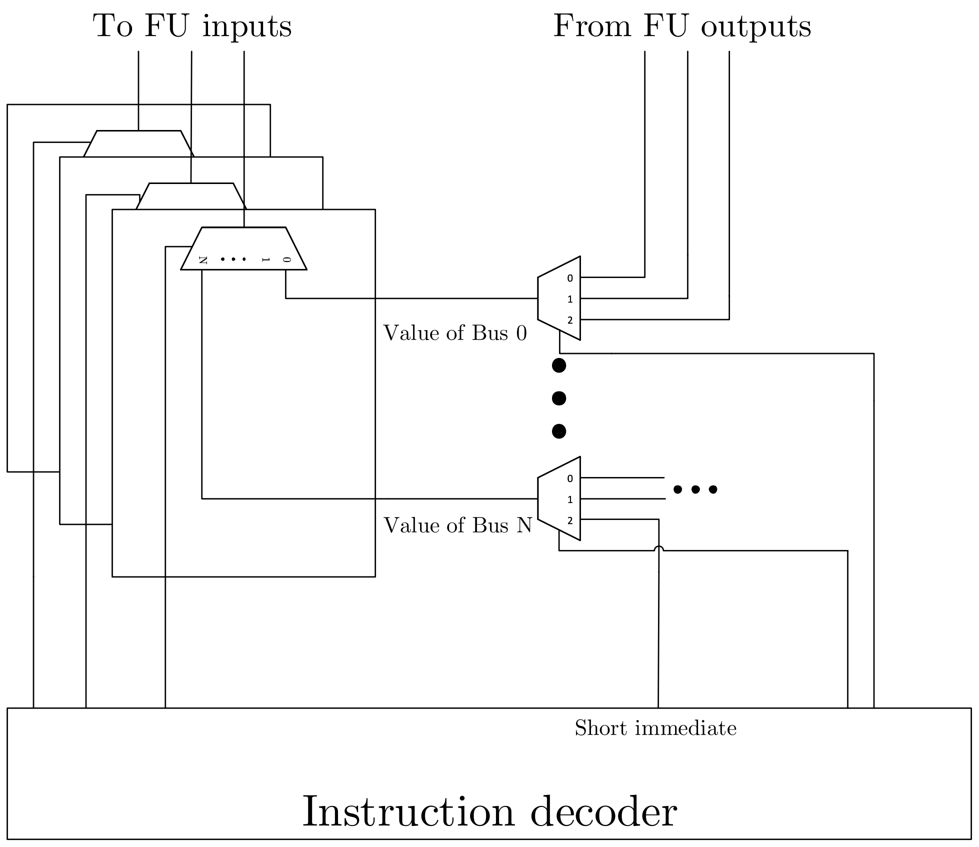


***Figure 5.1**. Multiplexer-based interconnection network.*

of the FPGA device, the decode process needs to examine the source fields of a single bus, rather than the source fields of every bus a given input socket is connected to. This reduces the number of inputs to the logic function required to determine the control signals and, subsequently, the number of logic elements required to implement it.

## 5.3 Memory Organization

The memory organization of a processor is an important factor contributing to its overall performance. Traditional approaches, such as cache hierarchies, also apply to TTAs. However, cache latency is inherently dynamic, depending on whether or not a cache miss occurs. This requires the processor to handle more lock signals. The LSUs are often the only source of the lock signal outside instruction fetch, as arithmetic operations generally have static latencies.

In addition to caches and other sources of variable latency memory interfaces, high latency interfaces pose an issue: extreme instruction latencies cannot be covered with other operations, so every memory load introduces many no-operation instructions, inflating the required instruction memory size. While they can be described as static latency memories, it is usually beneficial to introduce a lock signal. In these LSUs, the lock signal should be designed so that the logic within the LSU contributes little to the critical path originating from the lock signal. In other words, the lock signal should originate directly from a register, wherever possible.

Omitting the lock signal altogether simplifies the locking circuitry. This can be done by using smaller, static latency memories local to the processor. The input data and the results have to be moved between main memory and the processor through software control in this approach, and requires an external interface to the memory model. Hardened SRAM blocks in FPGAs usually have two bidirectional ports, one of which can be used for the LSU, while the other enables external access.

The LSUs have little control over the instantiated memory: in the interest of portability, memory models are kept outside the processor and given external signals to connect to. The LSU only has to provide an interface compatible with the memory model, which is fairly trivial in itself. However, like the reset, the lock signal is another high-fanout signal, connected to most of the registers of the TTA.

Since LSUs are associated with an address space and address spaces usually correspond to their own memory interface, the straightforward configuration is one LSU per address space. This avoids the need for arbitration logic between two LSUs attempting to access one address space. However, multiple LSUs accessing one address space allows for parallel accesses. FPGA memory primitives are commonly limited to two ports, and building memories with more ports is very expensive for large memories. Banking memory, i.e., dividing it into multiple independent blocks with e.g. one read and write port, can increase parallel performance when different memory banks are accessed at once, but this introduces variable latencies to the LSUs.

Register files are more self-contained, and have full control over how they are mapped to the FPGA primitives. While many-ported memories can be constructed from primitives with fewer ports, this can lead to a very large resource utilization [42]. Due to the port restrictions of FPGA memories, write port reduction is especially important, as each write port requires its own instance of the memory primitive as well as increases the required bookkeeping logic. This can be further multiplied if the RF read port count exceeds that of the memory primitive. The port thresholds vary, but for the Xilinx 7 Series devices, each set of three read ports requires its own set of memory primitives. Partitioning the RF into multiple less complex ones is one way to reduce the port count of the instanced memories.

## 5.4 Arithmetic Logic Units

Like LSUs, the challenge for FPGA optimization of ALUs is roughly the same: to efficiently map the arithmetic functions and memory, respectively, to the hardened blocks of the FPGA device. As a general guideline for all FUs, registers should not be reset unnecessarily, as that requires more routing resources and adds fanout to the reset signal.

Compared to LSUs, ALUs have a wider design space, as the operations of one ALU can be easily divided between multiple FUs. There are still reasons to keep certain operations together. For example, addition, substraction and comparison operations can share a most of their implementation logic, and keeping these operations together can save on resources. However, replicating an adder may be a sufficiently small price to pay for the possibility of executing two operations in parallel, on two different FUs. Furthermore, increasing the number of FUs increases the number of ports, and therefore also port registers, which can act as storage in lieu of an RF register.

When the application is suited to it, further parallel execution can be achieved by SIMD function units. As these apply an operation independently to multiple vector elements, without necessarily any communication between elements, the ALUs themselves can be scaled up almost without limit. However, a comparatively massive ALU and the similarly-sized RFs for the vector buses can essentially stretch the interconnect to cover more area, and therefore negatively impact the timing of the processor. Another bottleneck may be the communication between the scalar and vector buses. Whether it is done through memory or by FUs, picking one vector element to feed to a scalar bus can require a large multiplexer for wide vectors.

The ALUs have more control over the internal pipelining of the hardened blocks than LSUs, as they usually reside fully within their ALU. A direct implementation of the operation logic may not be enough, as there are many ways to organize even a simple multiplication, and the synthesis tool may decide to use the wrong organization for the optimization target set by the designer. As discussed, the direct instantiation of the hardened arithmetic components is sometimes required to use the ideal organization.

For example, the *2x32b* design presented in Section 6.4 only reached maximum clock frequencies between 93 and 154 MHz with various organizations of the SIMD ALU, when the multipliers were inferred from 32-bit wide multiplication operations in the VHDL code, while the same design with hand-instantiated multipliers performing the same task reached 195 MHz.

# 6. EVALUATION

In this chapter, different implementations of TTA architectures are evaluated experimentally. First, different configurations and optimizations of the subcomponents of a TTA architecture are evaluated and compared. Following that, high-performance TTA processor systems are discussed, and the scalability of TTA soft processors with SIMD capabilities is examined. Lastly, an FPGA re-design of an architecture designed for an ASIC implementation is presented, along with the individual changes made to the architecture.

## 6.1 Multiplexers on FPGA

***Table 6.1***. *Multiplexer synthesis results.*

| Inputs | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| LUTs | 32 | 32 | 32 | 64 | 64 |
| F7 Muxes | 0 | 0 | 0 | 0 | 0 |
| F8 Muxes | 0 | 0 | 0 | 0 | 0 |
| Delay (ns) | 1.903 | 1.961 | 2.093 | 2.371 | 2.54 |
| Inputs | 7 | 8 | 9 | 10 | 11 |
| LUTs | 64 | 64 | 97 | 128 | 128 |
| F7 Muxes | 32 | 32 | 0 | 0 | 0 |
| F8 Muxes | 0 | 0 | 0 | 0 | 0 |
| Delay (ns) | 2.385 | 2.571 | 2.888 | 2.808 | 2.745 |
| Inputs | 12 | 13 | 14 | 15 | 16 |
| LUTs | 128 | 130 | 128 | 128 | 128 |
| F7 Muxes | 0 | 32 | 64 | 64 | 64 |
| F8 Muxes | 0 | 0 | 32 | 32 | 32 |
| Delay (ns) | 2.59 | 3.046 | 2.632 | 2.943 | 3.088 |

In order to verify the comparison between scalar OTA and TTA interconnect complexity, a series of multiplexers were synthesized for an FPGA architecture. The synthesis results for 32-bit multiplexers with a varying amount of inputs can be seen in Table 6.1. They are the result of out-of-context synthesis for the multiplexers, with each input and output registered. The synthesis used the same parameters as a performance-focused synthesis profile in Vivado.

Against expectations, the delay is not a monotonic increase as the input count is increased. For example, the six-input multiplexer is slower than the 7-input one. The utilization results are more straightforward and go some way to explaining the timing discrepancies. For up to four inputs, the implementation seems to be the same: one LUT for each bit of the input. A six-input LUT can implement a four-input multiplexer, with two of the inputs occupied by the select signal.

The seven- and eight-input multiplexers start using the F7 multiplexer resources, each selecting between two bits. The five- and six-input multiplexers do not use these, as their inputs are sourced from two LUTs, and implementing the logic with two LUTs directly is cheaper. However, cascading two LUTs incurs more delay than one layer of LUTs feeding into one F7 multiplexer. Similarly, once the multiplexers start using four LUTs per bit, they do not need to use the F7 or F8 multiplexers until the input count rises to 13.

Assuming a 32-bit datapath, the logic utilization for the multiplexers for the scalar OTA machine shown earlier is equal to 160 LUTs, while the TTA equivalent uses 192 LUTs, an increase of 20 %. The critical path, on the other hand, is 4.3 nanoseconds for the OTA implementation, and 2.6 for the TTA.

## 6.2 TTA Suboptimizations

There were a large number of individual improvements and possible parameters to the TTA implementation presented, and testing every combination of them would be infeasible. Instead, the configurations and optimizations were tested on their own and compared against the TTA implementation used before the improvements presented in this thesis.

### 6.2.1 Reset Regime

In order to establish the effect of reset on the utilization and performance of a TTA core, the two options were compared with the seven TTA architectures used in [37], with the same methodology for determining utilization and maximum frequency. The same FPGA device, a Xilinx Zynq 7020, was used. The results can be seen in Table 6.2. The difference in utilization between asynchronous and synchronous reset was, while not major in an absolute sense, unexpectedly high: since the LUT FFs support both reset methods, the only difference should be found in the hardened blocks.

Inspection of the resulting netlist revealed that the internal synchronous reset had to be emulated with logic in the data path when accompanied by an asynchronous external reset, because the FFs can only have one type of reset at a time. When the external reset was synchronous, the internal reset could be combined with it for a single reset signal. Additionally, in the instruction fetch logic, the asynchronous reset loaded a constant to the instruction pointer register, while with the synchronous reset the value was sourced from an external signal. When both were active, extra logic was present, the amount depending on the instruction word.

In order to verify that the differences were primarily from a mismatch between two resets, the synthesis was rerun, while disabling the local reset by setting it to a constant, inactive value. This allowed the synthesis tool to remove the local reset logic. The differences between the asynchronous and synchronous utilization results are significantly smaller now and mainly present in the instruction decode phase. This may be due to the synthesis

***Table 6.2**. Comparison between different reset regimes.*

| | | Internal | LUTs | | | | |
|---|---|---|---|---|---|---|---|
| | Reset | reset | | Instruction | Instruction | | $F_{Max}$ |
| Arch. | type | enabled | Core | decode | fetch | FFs | (MHz) |
| m-tta-1 | Async. | Yes | 819 | 68 | 81 | 499 | 206 |
| m-tta-1 | Synch. | Yes | 765 (93 %) | 47 (69 %) | 48 (59 %) | 499 | 217 |
| m-tta-1 | Async. | No | 767 (94 %) | 49 (72 %) | 48 (59 %) | 499 | 203 |
| m-tta-1 | Synch. | No | 762 (93 %) | 45 (66 %) | 48 (59 %) | 499 | 217 |
| m-tta-2 | Async. | Yes | 984 | 114 | 100 | 588 | 214 |
| m-tta-2 | Synch. | Yes | 903 (92 %) | 85 (75 %) | 48 (48 %) | 588 | 208 |
| m-tta-2 | Async. | No | 905 (92 %) | 87 (76 %) | 48 (48 %) | 588 | 216 |
| m-tta-2 | Synch. | No | 898 (91 %) | 81 (71 %) | 48 (48 %) | 588 | 206 |
| m-tta-3 | Async. | Yes | 2002 | 230 | 135 | 869 | 167 |
| m-tta-3 | Synch. | Yes | 1866 (93 %) | 180 (78 %) | 48 (36 %) | 869 | 169 |
| m-tta-3 | Async. | No | 1860 (93 %) | 175 (76 %) | 48 (36 %) | 869 | 171 |
| m-tta-3 | Synch. | No | 1857 (93 %) | 171 (74 %) | 48 (36 %) | 869 | 170 |
| p-tta-2 | Async. | Yes | 1160 | 141 | 101 | 606 | 206 |
| p-tta-2 | Synch. | Yes | 1101 (95 %) | 135 (96 %) | 48 (48 %) | 606 | 213 |
| p-tta-2 | Async. | No | 1095 (94 %) | 129 (91 %) | 48 (48 %) | 606 | 209 |
| p-tta-2 | Synch. | No | 1097 (95 %) | 131 (93 %) | 48 (48 %) | 606 | 207 |
| p-tta-3 | Async. | Yes | 1982 | 254 | 131 | 874 | 190 |
| p-tta-3 | Synch. | Yes | 1865 (94 %) | 220 (87 %) | 48 (37 %) | 874 | 191 |
| p-tta-3 | Async. | No | 1845 (93 %) | 200 (79 %) | 48 (37 %) | 874 | 189 |
| p-tta-3 | Synch. | No | 1854 (94 %) | 209 (82 %) | 48 (37 %) | 874 | 189 |
| bm-tta-2 | Async. | Yes | 1059 | 125 | 93 | 577 | 213 |
| bm-tta-2 | Synch. | Yes | 981 (93 %) | 91 (73 %) | 48 (52 %) | 577 | 206 |
| bm-tta-2 | Async. | No | 984 (93 %) | 95 (76 %) | 48 (52 %) | 577 | 213 |
| bm-tta-2 | Synch. | No | 982 (93 %) | 92 (74 %) | 48 (52 %) | 577 | 207 |
| bm-tta-3 | Async. | Yes | 1769 | 209 | 109 | 821 | 188 |
| bm-tta-3 | Synch. | Yes | 1658 (94 %) | 159 (76 %) | 48 (44 %) | 821 | 187 |
| bm-tta-3 | Async. | No | 1661 (94 %) | 162 (78 %) | 48 (44 %) | 821 | 190 |
| bm-tta-3 | Synch. | No | 1654 (93 %) | 155 (74 %) | 48 (44 %) | 821 | 188 |

tool combining some datapath logic where registers are set to their reset values with the reset signal, saving some logic in the synchronous case.

The differences in maximum frequency are more consistent regardless of local reset. The cause, however, is not as clear. For example, in both *m-tta-1* and *bm-tta-2*, the critical path moves from an internal path starting from the RF index register in the asynchronous case to a path between the instruction memory and the bus interface which would be used to control the core in the synchronous case. In *m-tta-1* this leads to an increase in maximum frequency, while in *bm-tta-2* it leads to a decrease.

It is possible that moving a register from FFs to the instruction memory block is faster in the smaller cores, where the routing between the instruction memory and its fetch logic is not a bottleneck, while the same approach reduces the freedom of placement for the register in the larger cores, where having the registers midway between the memory and instruction fetch would be ideal, but it is difficult to tell from the data. Regardless, the differences for larger cores are minimal, as the instruction fetch stage does not have a major effect on the internal paths.

### 6.2.2 Interconnection Network

***Table 6.3***. *Comparison between interconnection network implementations.*

| | | LUTs | | | | |
|---|---|---|---|---|---|---|
| Architecture | IC type | Core | Instruction decode | IC | FFs | $F_{Max}$ (MHz) |
| m-tta-1 | AND-OR | 895 | 40 | 265 | 507 | 215 |
| m-tta-1 | Multiplexer | 765 (85 %) | 47 (118 %) | 128 (48 %) | 499 | 217 |
| m-tta-2 | AND-OR | 1117 | 69 | 438 | 599 | 206 |
| m-tta-2 | Multiplexer | 903 (81 %) | 85 (123 %) | 208 (47 %) | 588 | 208 |
| m-tta-3 | AND-OR | 2249 | 149 | 932 | 895 | 178 |
| m-tta-3 | Multiplexer | 1866 (83 %) | 180 (121 %) | 517 (55 %) | 869 | 169 |
| p-tta-2 | AND-OR | 1270 | 114 | 542 | 619 | 208 |
| p-tta-2 | Multiplexer | 1101 (87 %) | 135 (118 %) | 352 (65 %) | 606 | 213 |
| p-tta-3 | AND-OR | 2508 | 187 | 1290 | 908 | 190 |
| p-tta-3 | Multiplexer | 1865 (74 %) | 220 (118 %) | 614 (48 %) | 874 | 191 |
| bm-tta-2 | AND-OR | 1130 | 78 | 438 | 590 | 202 |
| bm-tta-2 | Multiplexer | 981 (87 %) | 91 (117 %) | 276 (63 %) | 577 | 206 |
| bm-tta-3 | AND-OR | 2196 | 142 | 1023 | 850 | 180 |
| bm-tta-3 | Multiplexer | 1658 (76 %) | 159 (112 %) | 468 (46 %) | 821 | 187 |

Like the earlier cases, the two interconnect options were synthesized for the same seven architectures for comparison. The results can be seen in Table 6.3. The differences in the IC as an isolated component are significant, ranging from 35 to 55 percent reduction. As the control signal to an N-connected output socket only require $log_2(N)$ bits in the multiplexer-based approach, while the one-hot signalling of the AND-OR network requires $N$ bits, so the number of registers is also slightly reduced. A slight logic utilization increase can be seen in the instruction decode stage, but this is comparatively small. When taking the entire processor into account, every architecture sees a reduction in logic utilization, ranging from 13 to 26 percent.

For the most part, minor increases in maximum frequency can be seen. A minor decrease can be seen in *m-tta-3*. In that architecture, many buses have connections to five output sockets, which requires three stages of two-to-one multiplexers. With four connections, only two stages would be needed. In other words, the IC of *m-tta-3* is just on the undesirable side of a step up in critical path latency and an intelligently-pruned interconnect may settle below it.

The multiplexer-based interconnect should be straightforward: each of the busses and each of the input sockets contribute one multiplexer with as many inputs as the bus or socket. However, as can be seen in Table 6.4, this is not always the case, as the synthesis tool may

***Table 6.4****. Predictions and synthesis results for the multiplexer-based IC LUT utilzation.*

| Architecture | Predicted | Actual | Difference |
|---|---|---|---|
| m-tta-1 | 128 | 128 | 0 % |
| m-tta-2 | 320 | 208 | -35 % |
| m-tta-3 | 608 | 517 | -15 % |
| p-tta-2 | 448 | 352 | -21 % |
| p-tta-3 | 704 | 614 | -13 % |
| bm-tta-2 | 320 | 276 | -14 % |
| bm-tta-3 | 481 | 468 | -3 % |

find ways to optimize the interconnect further. For example, not all of the inputs are 32 bits: short immediates and boolean RF outputs are significantly shorter. In some cases, it may also be possible to implement the socket and bus multiplexers as a single multiplexer.

## 6.3 Total Effect of Optimizations

***Table 6.5****. Comparison between original and fully FPGA-optimized implementations.*

| | | LUTs | | | | | |
|---|---|---|---|---|---|---|---|
| Arch. | Optimization | Core | Instr. decode | Instr. fetch | IC | FFs | $F_{Max}$ (MHz) |
| m-tta-1 | Original | 955 | 68 | 81 | 265 | 507 | 215 |
| m-tta-1 | FPGA | 765 (80 %) | 47 | 48 | 128 (48 %) | 499 | 217 |
| m-tta-2 | Original | 1210 | 115 | 100 | 435 | 599 | 207 |
| m-tta-2 | FPGA | 903 (75 %) | 85 | 48 | 208 (48 %) | 588 | 208 |
| m-tta-3 | Original | 2398 | 211 | 135 | 932 | 895 | 178 |
| m-tta-3 | FPGA | 1866 (78 %) | 180 | 48 | 517 (55 %) | 869 | 169 |
| p-tta-2 | Original | 1341 | 132 | 101 | 542 | 619 | 209 |
| p-tta-2 | FPGA | 1101 (82 %) | 135 | 48 | 352 (65 %) | 606 | 213 |
| p-tta-3 | Original | 2647 | 244 | 130 | 1290 | 908 | 189 |
| p-tta-3 | FPGA | 1865 (70 %) | 220 | 48 | 614 (48 %) | 874 | 191 |
| bm-tta-2 | Original | 1213 | 116 | 93 | 438 | 590 | 217 |
| bm-tta-2 | FPGA | 981 (81 %) | 91 | 48 | 276 (63 %) | 577 | 206 |
| bm-tta-3 | Original | 2319 | 204 | 109 | 1023 | 850 | 185 |
| bm-tta-3 | FPGA | 1658 (71 %) | 159 | 48 | 468 (46 %) | 821 | 187 |

In order to verify the total effect of the reset and interconnect optimizations, the fully FPGA-optimized designs were compared with the implementations without any of the optimizations. The synthesis results can be seen in Table 6.5. The difference in the interconnect network utilization is dominant, and the difference in IC logic utilization is similar as was seen with only that optimization. The number of registers is slightly reduced as a result of the reset regime change. Overall, the cores see a logic utilization reduction of 18 to 30 percent.

The effect on maximum frequency is minor, and some cores even see a minor decrease in clock rate. Based on the critical paths reported for the final synthesis results, it seems that the multiplexer-based interconnect may improve timing in some cases: all but one of the unoptimized implementations had a critical path in the interconnect, while only three of the FPGA-optimized ones had such a critical path. Instead, their critical paths were either internal to the ALU or between the instruction memory and the external memory bus used to program the core.

## 6.4 Parallel Execution on TTAs

There are many ways to increase soft processor performance through parallelism. Multiprocessor systems are the simplest approach: simply instantiate multiple copies of the same core, and connect them together through an external interconnect. A multiprocessor system will be presented as a case study, but exploring multiprocessor organizations is outside the scope of this thesis. Since the main contributor to multiprocessor performance scaling — the interconnect — is not part of the processor, it can be assumed that multiprocessor organizations scale similarly whether the cores themselves are TTAs, RISC processors, or any other architecture.

Like VLIW architectures, TTAs can attain a higher performance by introducing more function units. This is a simple approach, but simply increasing the computational capacity of the processor will move the bottleneck to the memory components. With some applications, the LSU bandwidth will be the main bottleneck, while other applications may require a large or complex register file for intra-processor communication.

Another way to increase the computational performance of a single core is using SIMD function units. In an ideal application, doubling the SIMD element count halves the cycle count. One easily vectorizable calculation that gets close to this is matrix multiplication: as long as the matrix rows have a number of elements divisible by the SIMD element count, all arithmetic can be done in fully-occupied SIMD operations.

A vectorized matrix multiplication program was developed with the goal to have adjustable arithmetic precision and SIMD element count. For the execution of this program, TTA architectures were generated from three templates, one for each arithmetic precision, for five machines per template with varying vector widths.

The architectures were modified from the *bm-tta-2* machine. Modifications include a new vector unit capable of multiplication and addition and a simplified LSU containing only the scalar operations needed by the matrix multiplication program and the vector loads and stores to match the vector unit. Additionally, function units containing some operations to interface the vector and scalar datapath together — such as a broadcast instruction — were added.

The simple vectorization of matrix multiplication can be seen in the simulation results, in Table 6.6. The execution time is approximately inversely proportional to the SIMD element count across all architectures.

***Table 6.6**. Cycle counts executing a 64-by-64 element matrix multiplication with varying arithmetic precision and vector width.*

| | **Arithmetic precision** | | |
|---|---|---|---|
| **SIMD element count** | 8 bits | 16 bits | 32 bits |
| 2 | - | - | 957663 |
| 4 | - | 462047 | 478431 |
| 8 | 225774 | 226288 | 227120 |
| 16 | 113382 | 113640 | 114472 |
| 32 | 57122 | 57124 | 58020 |
| 64 | 31518 | 28958 | - |

***Table 6.7**. Synthesis results for SIMD machines.*

| | LUTs | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Vector width | Core | Instr. decode | IC | Vector ALU | LSU | FFs | DSP48 blocks | Maximum frequency |
| 8x8 | 1612 | 124 | 514 | 67 | 312 | 1049 | 11 | 166 |
| 16x8 | 2242 | 125 | 738 | 131 | 561 | 1572 | 19 | 156 |
| 32x8 | 3625 | 128 | 1186 | 259 | 1177 | 2623 | 35 | 135 |
| 64x8 | 6201 | 125 | 2082 | 521 | 2234 | 4705 | 67 | 125 |
| 128x8 | 12077 | 128 | 3874 | 1039 | 4553 | 8908 | 131 | 108 |
| 4x16 | 1535 | 126 | 514 | 67 | 233 | 1065 | 7 | 188 |
| 8x16 | 2120 | 125 | 738 | 131 | 439 | 1584 | 11 | 172 |
| 16x16 | 3302 | 128 | 1186 | 259 | 862 | 2619 | 19 | 164 |
| 32x16 | 5733 | 124 | 2082 | 515 | 1764 | 4688 | 35 | 135 |
| 64x16 | 11145 | 128 | 3875 | 1028 | 3632 | 8853 | 67 | 117 |
| 2x32 | 1443 | 126 | 510 | 98 | 164 | 934 | 9 | 197 |
| 4x32 | 2018 | 126 | 734 | 194 | 377 | 1322 | 15 | 186 |
| 8x32 | 3161 | 126 | 1182 | 386 | 795 | 2096 | 27 | 170 |
| 16x32 | 5521 | 126 | 2078 | 771 | 1701 | 3652 | 51 | 156[1] |
| 32x32 | 9885 | 126 | 3870 | 1540 | 3165 | 6754 | 99 | 122 |

[1] The synthesis tool failed to synthesize the 16x32 machine with the same synthesis options as the other machines. This maximum frequency is the result of a synthesis without the `-keep_equivalent_registers` option.

The synthesis results in Table 6.7 show that the maximum frequency declines as the vector width is increased. Especially for machines with many SIMD elements, the bottleneck is in the LSU logic, specifically the multiplexer instantiated to select a narrow load result from the wide vector, e.g. a 32 bit result from a 32-by-32 bit vector. The logic utilization of the interconnect and ALU also scale linearly with vector width, but the additional lanes perform the same operations as the others, and the number of logic levels does not change.

The total runtimes, derived from the maximum frequency and cycle count for each of the architectures, are presented in Figure 6.1. While the maximum frequency declines with wider vector widths, the cycle count decreases are still dominant across the tested architectures.

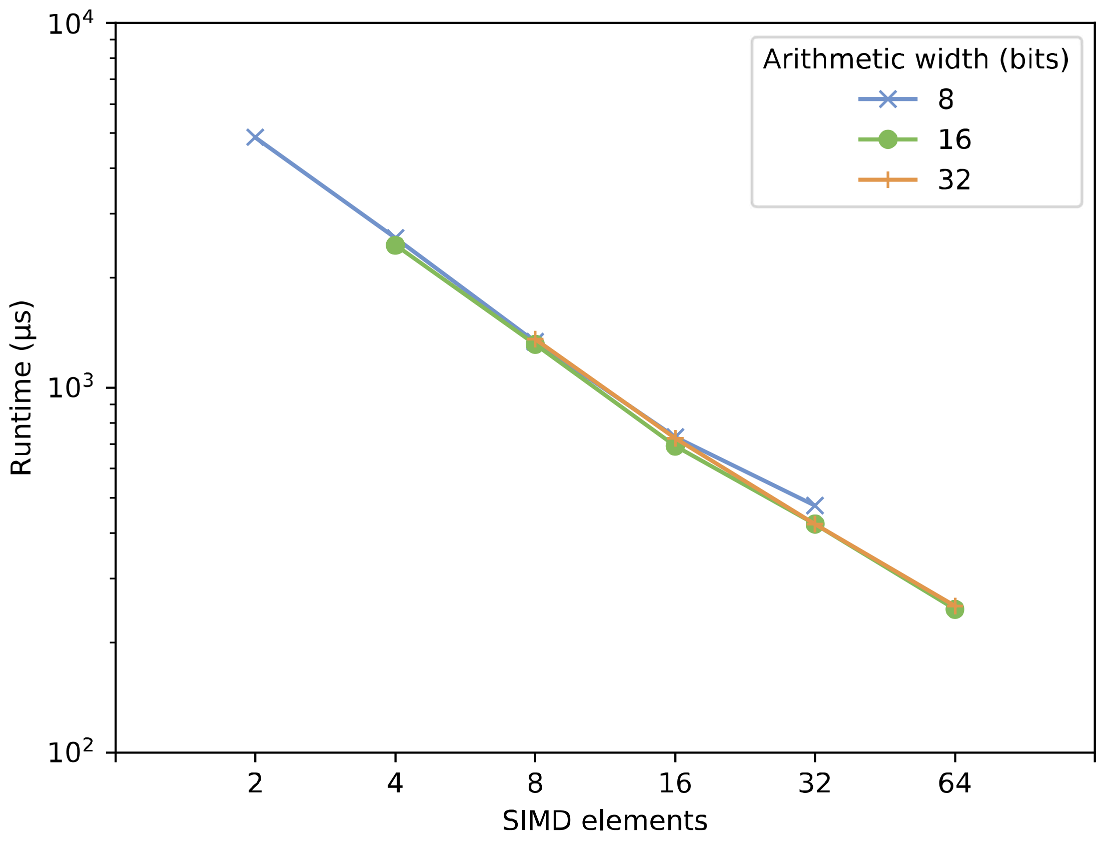


***Figure 6.1***. *Total runtimes of the vector architectures.*

## 6.5 Convolutional Neural Network Case Study

In this section, an FPGA-optimized design of an application-specific accelerator is presented. The architecture is an FPGA-focused redesign of the previously published AivoTTA accelerator [27], designed for convolutional neural network (CNN) applications. As the original AivoTTA design targeted ASIC technology, many changes had to be made in order to make the design suitable for FPGA devices.

In the original architecture, the interconnect was fully connected for the first three buses, i.e. there was a path from each of the input sockets to each output socket connected to those buses. This proved infeasible on FPGA and rarely-used connections were removed in order to simplify the inferred multiplexers. While this reduces the scheduling freedom of the compiler, only a minor cycle count decrease was seen as a result of pruning the connections.

The wide vector buses were found to be equally challenging for the synthesis tools. In the original design, there was one bus for each of the vector datapaths. The 1024-bit wide bus was split into two and given connections mainly to the multiply-add function unit and the register file. This improved the interconnect logic utilization while keeping the important paths intact.

While the clock frequency targets on the FPGA were significantly lower than the ASIC implementation, the FPGA implementation of some function units was found to be difficult at the latencies of the ASIC architecture. In particular, the single clock cycle latency for the vectorized multiply-add unit was not a reasonable target for the FPGA. Its latency was increased to 3 cycles and it was implemented using hand-instantiated hardened multipliers. Furthermore, the latencies of all FUs were increased to a minimum of 2 cycles, so that they could have a register at both ends of the instruction logic, isolating its their datapath from the interconnect network.

In addition to the increased latencies, several vector function units were merged together. As operand and output ports are combined, the complexity of the interconnect is reduced, simplifying the multiplexers there, while requiring multiplexers within the function unit. For simple operations, such as the SIMD greater-than instruction, the additional logic does not affect clock frequency. For more complex operations, i.e. those in the interpolation and multiply-add function units, were left as is. This helped simplify the 256-bit wide bus in particular.

The proposed architecture was used in two designs targetting two FPGA boards, the PYNQ-Z1, with a Zynq 7020 device, and the ZCU102, which has a Zynq UltraScale XCZU9EG device. The dual-core PYNQ-Z1 design could be used for embedded applications, such as camera drones, whereas the ZCU102 design targets high-performance use cases with 14 cores.

In order to evaluate the accelerators, the performance of a C application for face detection [24], also used in previous work [27], was measured. The CNN has four layers and uses layer fusion in order to improve data locality and reduce memory accesses. In both designs, it processes 720p frames.

In addition to the two programmable cores, the PYNQ-Z1 design includes fixed-function logic to interface with an HDMI video stream for a real-time face detection demonstration. This makes up approximately 14% of the logic utilization of the design.

For the calculation of the performance measurements, the application was simulated in the TCE toolchain simulator. The simulation results contain the utilization of the multiply-add operation as well as total cycle count. In the implemented design, however, memory accesses to off-chip memory introduce stalls which lower the utilization. The stall counts were read from debug registers after running the application for one frame.

The synthesis and performance results for the accelerator on both of the platforms is presented in Table 6.8. While the non-programmable accelerators reach significantly higher performance figures than the proposed design, particularly the multi-board designs presented by Zhang et al., the overall performance is acceptable.

There is still slack in the design. The memory accesses are costly for AivoTTA and the accelerator spends roughly a quarter of its execution time stalled due to memory accesses

***Table 6.8**. Comparison with state-of-the-art accelerators*

| Accelerator | Weight precision | Device and speed grade | $F_{max}$ (MHz) | DSP util. | Logic util. | GOPS |
|---|---|---|---|---|---|---|
| Fixed-function accelerators | | | | | | |
| Zhang et al. (2015) [94] | 32b float | Virtex-7 VX485T -2 | 100 | 2240 (80%) | 186251 (32%) | 61.62 |
| Zhang et al. (2016) [95][1] | 16b | Zynq 7045 -2 and Virtex-7 VX690T -2[2] | 150 | - | - | 290<br>825.6<br>1280.3 |
| Motamedi et al. [64] | 32b float | Virtex-7 VX485T | 100 | - | - | 84.2[3] |
| Qiu et al. [72] | 16b | Zynq 7045 -2 | 150 | 780 (87%) | 182616 (52%) | 137.0 |
| Li et al. [55] | 16b | Virtex-7 VX485T -2 | 156 | 2144 (77%) | 273805 (56%) | 565.9 |
| Ma et al. (2017) [57] | 8 and 16b | Arria 10 GX 1150 -2 | 150 | 1518 (100%) | 161000 (14%) | 137.0 |
| HLS accelerators | | | | | | |
| Suda et al. [78] | 8 and 16b | Stratix-V GSD8 | 120 | 727 (37%) | 120000 (17%) | 72.4 |
| Ma et al. (2016) [58] | 8 and 16b | Stratix-V GSD8 | 100 | 256 (13%) | 112000 (16%) | 117.3 |
| Runtime-reconfigurable accelerators | | | | | | |
| NeuFlow [21] | 16b | Virtex-6 VLX240T -1 | 200 | - | - | 147 |
| HLL programmable accelerators | | | | | | |
| CNP [22] | 16b | Virtex-4 SX35 | 200 | 54 (28%) | 31000 (90%) | 6.8 |
| Sankaradas et al. [76] | 16b | Virtex-5 LX330T -2 | 115 | 107 (56%) | 35263 (68%) | 6.74 |
| Proposed design | 16b | Zynq 7020 -1<br>Zynq ZU9EG -2 | 145<br>300 | 72 (33%)<br>504 (20%) | 46261 (87%)<br>212950 (78%) | 3.71<br>48.48 |

[1] Designs B and D omitted, as the latency-optimized designs fare strictly worse in this comparison.
[2] One Zynq 7045 device and one, four, or six Virtex-7 VX690T devices.
[3] Theoretical peak GOPS from simulation.

to off-chip DRAM. This could be avoided by the better use of the local memories for each of the cores, or an LSU with a two-stage load instruction that stalls only when the data is needed. Furthermore, the 1024-bit bus could be split into two 512-bit busses, which could be served by a single 512-bit LSU while keeping the vector FUs on that bus occupied. However, these would have meant a more radical overhaul of the accelerator.

# 7. RELATED WORK

Previous work on soft processors spans a wide range of designs. Often, the simplest solution is to use soft processors supplied by the FPGA vendors. All of the vendors presented in Section 4.1 provide RISC-like soft processors as intellectual property (IP) cores with some degree of customization [33, 49, 59, 87]. In particular, Xilinx and Intel provide MicroBlaze and NIOS II, respectively, both of which are single-issue, in-order RISC processors with many customization options, from a small control processor with a minimal instruction set to a high-performance configuration with memory management units and other resources for the capability of running a Linux distribution. For the Xilinx device targeted in this thesis, the MicroBlaze soft processor has been previously compared with a small TTA processor using some of the optimizations presented in this work [37].

Generally, high-performance designs for soft processors focus heavily on explicit parallelism, as the required control logic and iter-PE communication is simpler. In SIMD processors, communication is often done through memory rather than the RF, allowing a simple RF implementation that is connected to a single SIMD lane [13, 93], as was done in this thesis. There are some solutions for the complexity of the memory interface. For example, the SIMD processor design by Cho et al. [13] has a 17-bank memory access, so that the 16 PEs can access elements separated by common stride sizes – powers of two – simultaneously. Similarly, VEGAS [14] and the commercial MXP [77] have replaced the conventional RF with banked scratchpad memories with shift networks to read and write the data from and to the correct memory bank.

VLIW processors are another way to encode parallelism explicitly in the instruction word. The primary hurdle in implementing VLIW soft processors appears to be the implementation of the RF. Jones et al. [40] acknowledge as much, and attribute this to the expensive nature of wide multiplexers. While methods to reduce the RF complexity by partitioning a large RF into multiple smaller ones with fewer ports is common in ASIC implementations of VLIWs, FPGA designs have some novel approaches to the problem, possibly because the complex RFs are more costly to implement on FPGAs. Saghir et al. [75] simplify the RF by banking the register file of their 2-issue VLIW processor. As a result, two even or two odd registers cannot be updated simultaneously. Purnaprajna and Ienne [71] suggest changes to the FPGA architecture that would make it more suitable for the construction of multi-port RFs.

LaForest and Steffan [42] propose a multi-ported RF configuration that works within the existing architectures. For an RF with N read and M write ports, this design instantiates M banks with N two-port RFs. In contrast to simple banking, where one write port can only write to a subset of registers, the source of the read is selected by a live value table. It keeps track of which register bank has been last written. In a configuration with 4 write and 8 read ports, typical for a 4-issue VLIW processor, this RF design would utilize 32 times the block RAM required to implement a similarly-sized RF, and a sizable number of LUTs. While there is an approach to reduce LUTs [43], this increases the required number of RAM blocks further. This organization was used in the previous TTA soft processor work [37] comparing TTAs and VLIW processors, and was used as the basis of examining register file scalability in this thesis.

In terms of FPGA-optimized function units, more generic papers rarely go into detail about their operation logic implementations. However, there are designs focused on utilizing the hardended arithmetic blocks. Due to its inclusion in the hardened arithmetic blocks of FPGAs and usefulness in many DSP applications, combined multiply-add operations have featured in many processors [63, 75, 93]. DSP blocks can be a good choice used for shifting, as it would require comparatively expensive multiplexers if implemented in LUTs [92]. Efficient use of hardened arithmetic has been the focus of entire processor designs. iDEA [12] is a soft processor designed around the hardened resources of a Xilinx FPGA. In particular, the arithmetic-logic unit (ALU) of the design is mapped to a hardened arithmetic block and uses no LUTs. Octavo [43] is likewise a processor family designed from the ground up to map well on FPGA technology. With aggressive pipelining – up to 16 pipeline stages – the Octavo processors reach very high operating frequencies. The authors outline a plan to use the Octavo processors as the processing elements of SIMD and VLIW processors, an avenue for research likely also suited to iDEA.

# 8. FUTURE WORK

The TTA processor design space is large. Even when constraining the processor to a fixed set of operations, the processor can be altered in terms of the organization of operations into function units, operation latencies, the number of register files, the number of ports on the register files, the size of the IC and its connectivity. This can be overwhelming for a designer, even if they have a specific application to optimize for. As such, methods of approximating implementation costs, such as IC network logic utilization, from the processor architecture without a lengthy synthesis process could reduce design effort and speed up iteration times.

On the platform level, a multiprocessor system would reach a significantly higher performance on a parallel workload. This includes optimizing memory organization, selection of the processors and the interconnect to match the workload. Some work has been done with more general-purpose processors: Nilokov et al. [65] presented a multiprocessor platform generator using a high-level topological description, evaluated with MicroBlaze soft processors. In the same vein, Ma et al. [56] and Lebedev et al. [53] both presented frameworks for the automatic generation of multiprocessor platforms, including memory organization, from parallel OpenCL code. Ma et al. used MicroBlaze processors, while Lebedev et al. used a mixture of generic and customized processors for their platform. The degree of customization on TTAs is higher compared with traditional operation-triggered architectures, and an automatic framework would have more processor candidates to choose from than the ones in any of these papers. Therefore, accurate approximations of processor performance would be a requirement for such a system.

More generally, supporting high-level programming models like OpenCL can significantly ease the programming effort of TTAs, especially during processor and platform design space exploration. Abstracting data transfers between the host processor and the accelerator and internally between TTA accelerators removes some of the burden from the user, especially when the accelerators use local memories instead of or alongside caches. This could remove the need for long latency accesses to system-level memory. Another approach would be to divide the load into two operations, one which initiates the load and one which reads its value from a buffer. Since the processor only needs to stall when a value is read before it is ready, this can reduce the number of stall cycles for long-latency loads.

SIMT soft processors are a more recent research avenue, allowing for highly parallel exection. SIMT organizations tend to have more PEs in total and the approach generally targets very parallel workloads, with most soft SIMT processors composed of multiple SIMD blocks [1, 2, 4, 6]. Using barrel threading to spread warp computation over multiple clock cycles allows simplifying the RF since fewer simultaneous accesses are required [41].

Integrating TTA processors as the PEs of SIMT machines is an interesting research avenue. While TTAs can work as small, high-performance cores, the combination of static instruction scheduling on an exposed datapath processor and warp scheduling might introduce new obstacles to a high performance design.

Another way to leverage the strengths of FPGA devices is to use the reconfigurability during execution. Anjam et al. [5] presented a VLIW design that could be split e.g. from a single 4-issue processor to two 2-issue ones through partial reconfiguration, while TUKUTURI [70] allows customizing operations using the same. Both designs, however, require intervention from outside the soft processor to function. Nolting et al. [67] propose a self-reconfigurable processor which dynamically reconfigures its function units based on the executed software. At the processor level, TTA function units are very modular, and could benefit from a processor template combined with partial reconfiguration for a TTA design.

# 9. CONCLUSIONS

This thesis presented optimizations for TTA implementations for a soft processor use case. The optimizations have been integrated to the TCE toolchain and can be enabled without modifications to the processor architecture.

In this thesis, the TTA architecture was presented and contrasted with VLIW and scalar OTA machines. The complexity of TTA processors was examined from the point of view of FPGA implementations, showing that the transformation to TTAs does not increase the interconnect logic complexity. Therefore, the primary drawback remains to be the increased instruction word length and the consequent memory size increase.

Modern FPGA architectures are covered with a focus on the resources central to implementing operation logic and memory structures. The design approaches for FPGAs are covered to provide context for the soft processor use case, followed by a comparison between FPGA and ASIC technologies in terms of processor implementation.

The FPGA-centric optimizations were evaluated through synthesis on TTA processors with and without each optimization to determine the individual effects of the changes. The biggest difference was found to be from the interconnection network optimizations, where the network itself required up to 54 % less logic to implement with the optimizations than without. Taking all of the optimizations into account, the logic utilization of the entire core was reduced by up to 30 %. This improvement can be achieved without designer effort beyond enabling the processor generator option.

For architectural optimizations, this thesis explored wide SIMD machine implementations on FPGA devices. The primary bottleneck for the evaluated TTA processors is in the LSU access path, particularly selecting e.g. 32-bit values from a wide bus. This can be mitigated by, for example, increasing the latency of the scalar LSU operations. When the application can utilize wide vectors, the cycle count decrease has a greater effect than the clock period increase, resulting in an increase in performance.

Finally, a CNN processor originally designed for CNN applications was redesigned for FPGA implementation, having originally been implemented on ASIC technologies. The implemented changes were described and performance of the resulting architectures was evaluated on two FPGA devices and compared with fixed-function, runtime configurable and runtime programmable accelerators.

# REFERENCES

[1] A. Al-Dujaili, F. Deragisch, A. Hagiescu, W.F. Wong, Guppy: A GPU-like soft-core processor, in: Field-Programmable Technology (FPT), 2012 International Conference on, IEEE, 2012, pp. 57–60.

[2] M. Al Kadi, B. Janssen, M. Huebner, FGPU: An SIMT-architecture for FPGAs, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 254–263.

[3] Amazon EC2 F1 instances, Amazon Web Services, website. Available (accessed on 18.5.2018): https://aws.amazon.com/ec2/instance-types/f1/

[4] K. Andryc, M. Merchant, R. Tessier, FlexGrip: A soft GPGPU for FPGAs, in: Field-Programmable Technology (FPT), 2013 International Conference on, IEEE, 2013, pp. 230–237.

[5] F. Anjam, M. Nadeem, S. Wong, A VLIW softcore processor with dynamically adjustable issue-slots, in: Field-Programmable Technology (FPT), 2010 International Conference on, IEEE, 2010, pp. 393–398.

[6] R. Balasubramanian, V. Gangadhar, Z. Guo, C.H. Ho, C. Joseph, J. Menon, M.P. Drumond, R. Paul, S. Prasad, P. Valathol *et al.*, Enabling GPGPU low-level hardware explorations with MIAOW: an open-source RTL implementation of a GPGPU, ACM Transactions on Architecture and Code Optimization (TACO), Vol. 12, Iss. 2, 2015, p. 21.

[7] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown *et al.*, Tile64-processor: A 64-core SoC with mesh interconnect, in: Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, IEEE, 2008, pp. 88–598.

[8] A. Brant, G.G. Lemieux, ZUMA: An open FPGA overlay architecture, in: Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on, IEEE, 2012, pp. 93–96.

[9] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, W. Yoder, Scaling to the end of silicon with EDGE architectures, Computer, Vol. 37, Iss. 7, 2004, pp. 44–55.

[10] D. Capalija, T.S. Abdelrahman, A high-performance overlay architecture for pipelined execution of data flow graphs, in: Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, IEEE, 2013, pp. 1–8.

[11] K. Chapman, Get smart about reset: think local, not global, Xilinx, White Paper WP272 (v1.0.1), 2008, 7 p. Available: https://www.xilinx.com/support/documentation/white_papers/wp272.pdf

[12] H.Y. Cheah, F. Brosser, S.A. Fahmy, D.L. Maskell, The iDEA DSP block-based soft processor for FPGAs, ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 7, Iss. 3, 2014, p. 19.

[13] J. Cho, H. Chang, W. Sung, An FPGA based SIMD processor with a vector memory unit, in: Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on, IEEE, 2006, pp. 4–pp.

[14] C.H. Chou, A. Severance, A.D. Brant, Z. Liu, S. Sant, G.G. Lemieux, VEGAS: Soft vector processor with scratchpad memory, in: Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays, ACM, 2011, pp. 15–24.

[15] A. Cilio, H. Schot, J. Janssen, P. Jääskeläinen, Architecture Definition File: Processor Architecture Definition File Format for a New TTA Design Framework, Techn. rep., 55 p. Available: http://tce.cs.tut.fi/specs/ADF.pdf

[16] S. Collange, Multi-threading or SIMD? How GPU architectures exploit regularity, Presentation, 2011. Available (accessed on 3.10.2018): https://www.irisa.fr/alf/downloads/collange/talks/archi11_scollange.pdf

[17] S. Collange, Stack-less SIMT reconvergence at low cost, 2011. Available (accessed on 15.11.2018): https://hal.archives-ouvertes.fr/hal-00622654/

[18] B.W. Coon, J.E. Lindholm, System and method for managing divergent threads in a SIMD architecture, 2008. US Patent 7,353,369.

[19] H. Corporaal, Transport triggered architectures: Design and evaluation, dissertation, Technische Universiteit Delft, 1995.

[20] P. Dillien, And the winner of best FPGA of 2016 is..., EE Times, blog post, 2017. Available (accessed on 5.5.2018): https://www.eetimes.com/author.asp?section_id=36&doc_id=1331443

[21] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, Y. LeCun, Neuflow: A runtime reconfigurable dataflow processor for vision, in: Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on, IEEE, 2011, pp. 109–116.

[22] C. Farabet, C. Poulet, J.Y. Han, Y. LeCun, CNP: An FPGA-based processor for convolutional networks, in: Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, IEEE, 2009, pp. 32–37.

[23] FPGA-accelerated cloud server, Huawei Technologies Co., Ltd., website. Available (accessed on 18.5.2018): https://www.huaweicloud.com/en-us/product/fcs.html

[24] C. Garcia, M. Delakis, Convolutional face finder: A neural architecture for fast and robust face detection, IEEE Transactions on pattern analysis and machine intelligence, Vol. 26, Iss. 11, 2004, pp. 1408–1423.

[25] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B.C. Lee, S. Richardson, C. Kozyrakis, M. Horowitz, Understanding sources of inefficiency in general-purpose chips, in: ACM SIGARCH Computer Architecture News, ACM, 2010, Vol. 38, pp. 37–47.

[26] J.L. Hennessy, D.A. Patterson, Computer Architecture: a Quantitative Approach, 5th ed., Morgan Kaufmann, Waltham, MA, USA, 2012.

[27] J. IJzerman, T. Viitanen, P. Jääskeläinen, H. Kultala, L. Lehtonen, M. Peemen, H. Corporaal, J. Takala, AivoTTA: An energy efficient programmable accelerator for CNN-based object recognition, in: International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS XVIII), 2018. Forthcoming.

[28] Intel® Arria® 10 Device Overview, Intel, Literature Number: A10-OVERVIEW, 2018, 43 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf

[29] Intel® Cyclone® 10 GX Device Overview, Intel, Literature Number: C10GX51001, 2018, 21 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-10/c10gx-51001.pdf

[30] Intel® Cyclone® 10 LP Device Overview, Intel, Literature Number: C10LP51001, 2017, 10 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-10/c10lp-51001.pdf

[31] A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements, Intel, White paper, 2018, 6 p. Available (accessed on 4.10.2018): https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf

[32] Intel® MAX® 10 FPGA Device Overview, Intel, Literature Number: M10-OVERVIEW, 2017, 14 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/max-10/m10_overview.pdf

[33] Nios II Processor Reference Guide, Intel, Literature Number: NII-PRG, 2018, 233 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf

[34] Quartus Prime Standard Edition Handbook, Intel, Literature Number: QPS5V1, 2017, 1916 p. Available: https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf

[35] Stratix 10 GX/SX Device Overview, Intel, Literature Number: S10-OVERVIEW, 2017, 37 p. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf

[36] Intel Stratix 10 LAB and ALM Architecture and Features, Intel, Literature Number: UG-S10LAB, 2018, 18 p. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf

[37] P. Jääskeläinen, A. Tervo, G. Payá-Vayá, T. Viitanen, N. Behmann, J. Takala, H. Blume, Transport-triggered soft cores, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2018, pp. 83–90.

[38] P. Jääskeläinen, T. Viitanen, J. Takala, H. Berg, HW/SW Co-design Toolset for Customization of Exposed Datapath Processors, Springer International Publishing, 2017, pp. 147–164. Available: https://doi.org/10.1007/978-3-319-49679-5_8

[39] A.K. Jain, S.A. Fahmy, D.L. Maskell, Efficient overlay architecture based on DSP blocks, in: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, IEEE, 2015, pp. 25–28.

[40] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, J. Foster, An FPGA-based VLIW processor with custom hardware execution, in: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, ACM, 2005, pp. 107–117.

[41] J. Kingyens, J.G. Steffan, A GPU-inspired soft processor for high-throughput acceleration, in: Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, IEEE, 2010, pp. 1–8.

[42] C.E. LaForest, J.G. Steffan, Efficient multi-ported memories for FPGAs, in: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, ACM, 2010, pp. 41–50.

[43] C.E. LaForest, J.G. Steffan, Octavo: an FPGA-centric processor family, in: Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, ACM, 2012, pp. 219–228.

[44] S. Lahti, P. Sjövall, J. Vanne, T.D. Hämäläinen, Are we there yet? A study on the state of high-level synthesis, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, early access, 2018.

[45] CrossLink Family, Lattice Semiconductor, Data Sheet FPGA-DS-02007 Version 1.4, 2018, 58 p. Available: http://www.latticesemi.com/view_document?document_id=51662

[46] ECP5 and ECP5-5G Family, Lattice Semiconductor, Data Sheet FPGA-DS-02012 Version 1.9, 2018, 108 p. Available: http://www.latticesemi.com/view_document?document_id=51754

[47] iCE40 Ultra Family, Lattice Semiconductor, Data Sheet FPGA-DS-02028 Version 2.2, 2018, 50 p. Available: http://www.latticesemi.com/view_document?document_id=50666

[48] iCE40 UltraLite Family, Lattice Semiconductor, Data Sheet DS1050 Version 1.4, 2016, 37 p. Available: http://www.latticesemi.com/view_document?document_id=50945

[49] LatticeMico32 Processor Reference Manual, Lattice Semiconductor, Techn. rep., 2012, 101 p. Available: http://www.latticesemi.com/view_document?document_id=51108

[50] MachXO3 Family, Lattice Semiconductor, Data Sheet FPGA-DS-02032 Version 2.1, 2018, 91 p. Available: http://www.latticesemi.com/view_document?document_id=50121

[51] How to use GSR, PUR, and TSALL, Lattice Semiconductor, Techn. rep., 2009, 20 p. Available: http://www.latticesemi.com/view_document?document_id=31408

[52] C. Lattner, Llvm and clang: Next generation compiler technology, in: The BSD conference, 2008, pp. 1–2.

[53] I. Lebedev, S. Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, M. Lin, J. Wawrzynek, MARC: A many-core approach to reconfigurable computing, in: Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on, IEEE, 2010, pp. 7–12.

[54] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, S. Amarasinghe, Space-time scheduling of instruction-level parallelism on a RAW machine, in: ACM SIGPLAN Notices, ACM, 1998, Vol. 33, pp. 46–57.

[55] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, L. Wang, A high performance FPGA-based accelerator for large-scale convolutional neural networks, in: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on, IEEE, 2016, pp. 1–9.

[56] S. Ma, M. Huang, D. Andrews, Developing application-specific multiprocessor platforms on FPGAs, in: Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on, IEEE, 2012.

[57] Y. Ma, Y. Cao, S. Vrudhula, J.s. Seo, Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2017, pp. 45–54.

[58] Y. Ma, N. Suda, Y. Cao, J.s. Seo, S. Vrudhula, Scalable and modularized RTL compilation of convolutional neural networks onto FPGA, in: Field Programmable Logic and Applications (FPL), 2016 26th International Conference on, IEEE, 2016, pp. 1–8.

[59] Mi-V embedded ecosystem, Microsemi, website. Available (accessed on 15.6.2018): https://www.microsemi.com/product-directory/fpga-soc/3872-embedded-processing

[60] PolarFire FPGA Fabric, Microsemi, User Guide UG0680 Revision 4.0, 2018, 111 p. Available: https://www.microsemi.com/document-portal/doc_view/136522-ug0680-polarfire-fpga-fabric-user-guide

[61] RRG4 FPGA Fabric, Microsemi, User Guide UG0574 Revision 3.0, 2017, 124 p. Available: https://www.microsemi.com/document-portal/doc_view/134407-ug0574-rtg4-fpga-fabric-user-guide

[62] SmartFusion2 SoC FPGA and IGLOO2 FPGA Fabric, Microsemi, User Guide UG0445 Revision 6.0, 2017, 124 p. Available: https://www.microsemi.com/document-portal/doc_view/132008-ug0445-smartfusion2-soc-fpga-and-igloo2-fpga-fabric-user-guide

[63] M. Milford, J. McAllister, An ultra-fine processor for FPGA DSP chip multiprocessors, in: Signals, Systems and Computers, 2009 Conference Record of the Forty-Third Asilomar Conference on, IEEE, 2009, pp. 226–230.

[64] M. Motamedi, P. Gysel, V. Akella, S. Ghiasi, Design space exploration of FPGA-based deep convolutional neural networks, in: ASP-DAC, 2016, pp. 575–580.

[65] H. Nikolov, T. Stefanov, E. Deprettere, Efficient automated synthesis, programing, and implementation of multi-processor platforms on FPGA chips, in: Field Programmable Logic and Applications, 2006. FPL'06. International Conference on, IEEE, 2006.

[66] S. Nolting, G. Payá-Vayá, H. Blume, Optimizing VLIW-SIMD processor architectures for FPGA implementation, Proceedings of the ICT. OPEN, Vol. 2011, 2011.

[67] S. Nolting, G. Payá-Vayá, F. Giesemann, H. Blume, S. Niemann, C. Müeller-Schloer, Dynamic self-reconfiguration of a MIPS-based soft-processor architecture, in: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, IEEE, 2016, pp. 172–180.

[68] NVidia Telsa V100 GPU Architecture, NVidia, White paper WP-08608-001_v1.1, 2017, 36 p. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[69] Occupational outlook handbook, Bureau of Labor Statistics, website. Available (accessed on 18.5.2018): https://www.bls.gov/ooh/home.htm

[70] G. Payá-Vayá, R. Burg, H. Blume, Dynamic data-path self-reconfiguration of a VLIW-SIMD soft-processor architecture, in: Workshop on Self-Awareness in Reconfigurable Computing Systems, SRCS, 2012.

[71] M. Purnaprajna, P. Ienne, Making wide-issue VLIW processors viable on FPGAs, ACM Transactions on Architecture and Code Optimization (TACO), Vol. 8, Iss. 4, 2012, p. 33.

[72] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, Going deeper with embedded FPGA platform for convolutional neural network, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 26–35.

[73] RI5CY: RISC-V Core, Repository. Available: https://github.com/pulp-platform/riscv

[74] A. Rushton, VHDL for logic synthesis, 2nd ed., John Wiley & Sons, Chichester, United Kingdom, 1998, 375 p.

[75] M.A. Saghir, M. El-Majzoub, P. Akl, Datapath and ISA customization for soft VLIW processors, in: Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on, IEEE, 2006, pp. 1–10.

[76] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, H.P. Graf, A massively parallel coprocessor for convolutional neural networks, in: Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on, IEEE, 2009, pp. 53–60.

[77] A. Severance, G.G. Lemieux, Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor, in: Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, IEEE Press, 2013, p. 6.

[78] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.s. Seo, Y. Cao, Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2016, pp. 16–25.

[79] S. Swanson, K. Michelson, A. Schwerin, M. Oskin, WaveScalar, in: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2003, p. 291.

[80] M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, P. Stenstrom, FlexCore: Utilizing exposed datapath control for efficient computing, Journal of Signal Processing Systems, Vol. 57, Iss. 1, 2009, pp. 5–19.

[81] S.M. Trimberger, Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology, Proceedings of the IEEE, Vol. 103, Iss. 3, 2015, pp. 318–331.

[82] L. Waeijen, D. She, H. Corporaal, Y. He, A low-energy wide SIMD architecture with explicit datapath, Journal of Signal Processing Systems, Vol. 80, Iss. 1, 2015, pp. 65–86.

[83] H. Wong, V. Betz, J. Rose, Quantifying the gap between FPGA and custom CMOS to aid microarchitectural design, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 22, Iss. 10, 2014, pp. 2067–2080.

[84] 7 Series FPGAs Configurable Logic Block, Xilinx, User Guide UG474 (v1.8), 2016, 74 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[85] 7 Series FPGAs DSP48E1 Slice, Xilinx, User Guide UG479 (v1.10), 2018, 58 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[86] 7 Series FPGAs Memory Resources, Xilinx, User Guide UG473 (v1.12), 2016, 86 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[87] MicroBlaze Processor Reference Guide, Xilinx, User Guide UG4984 (v2018.1 ), 2018, 316 p. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug984-vivado-microblaze-ref.pdf

[88] UltraScale Architecture Configurable Logic Block, Xilinx, User Guide UG574 (v1.5), 2017, 58 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[89] UltraScale Architecture DSP Slice, Xilinx, User Guide UG579 (v1.7), 2018, 75 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[90] UltraScale Architecture Memory Resources, Xilinx, User Guide UG573 (v1.9), 2018, 136 p. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[91] P. Yiannacouras, J.G. Steffan, J. Rose, VESPA: portable, scalable, and flexible FPGA-based vector processors, in: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems, ACM, 2008, pp. 61–70.

[92] P. Yiannacouras, J.G. Steffan, J. Rose, Portable, flexible, and scalable soft vector processors, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 20, Iss. 8, 2012, pp. 1429–1442.

[93] J. Yu, C. Eagleston, C.H.Y. Chou, M. Perreault, G. Lemieux, Vector processing as a soft processor accelerator, ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 2, Iss. 2, 2009, p. 12.

[94] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ACM, 2015, pp. 161–170.

[95] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, J. Cong, Energy-efficient CNN implementation on a deeply pipelined FPGA cluster, in: Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ACM, 2016, pp. 326–331.