Tampere University

Malik Quamrus Samawat

EFFICIENT EXPOSED SIMD DATAPATH CONNECTIVITY ON FPGAS

Master of Science Thesis
Information Technology and Communication Sciences
Examiners: Asst. Prof. Pekka Jääskeläinen
M.Sc. Joonas Multanen
October 2020

ABSTRACT

Malik Quamrus Samawat: Efficient Exposed SIMD Datapath Connectivity on FPGAs
Master of Science Thesis
Tampere University
MSc in Information Technology
October 2020

Field-programmable gate arrays (FPGA) are popular for their ability to be configured and reconfigured at any instant of time. They consist of numerous logic blocks linked with programmable interconnects. FPGAs provide high parallelism, low power consumption, greater efficiency in comparison to most other similar devices available in the market. However, they are very difficult to program for the end-users as programming FPGAs require knowledge regarding hardware design.

Using soft-core processors can somewhat alleviate this issue. Transport-triggered architectures (TTA) can be used for processor designing. FPGAs can be programmed using TTAs as an overlay layer. They have gained popularity due to their exploitation of the parallel programming model. Though they are still under research, yet they have shown promising results in improving execution time, image size, and resource utilization. TTAs are simple to implement in FPGAs and are quite efficient.

TTAs implement instruction-level parallelism through different functional units. Parallel data transports and interconnections are essential to program these function units in parallel. Based on the efficient utilization of this datapath connectivity, the efficiency of TTA architecture in FPGAs can be enhanced quite a bit.

This thesis presents a TTA variation with reduced internal connectivity beyond standard VLIW processors to improve the implementation efficiency measured in terms of clock frequency and resource utilization. It demonstrates an exposed single instruction multiple data (SIMD) datapath connectivity model with a reduced number of connections that can successfully sustain one per cycle operation throughput. Additionally, the performance of this reduced connectivity template is compared with a previously demonstrated architecture based on an equivalent metric. After verification, the findings show promising improvement in terms of clock frequency and resource utilization.

Keywords: SIMD, datapath, reduced connectivity, TTA, FPGA, IC network

PREFACE

The work for this thesis has been done at Tampere University, Hervanta Campus. This project has received funding from ECSEL Joint Undertaking (JU) under grant agreement No 783162. The JU receives support from the European Union's Horizon 2020 research and innovation program and the Netherlands, Czech Republic, Finland, Spain, Italy.

I would like to thank Prof. Pekka Jääskeläinen for providing me with the opportunity to work on this project and help me broaden my knowledge in this field. I also offer my gratitude to him and M.Sc. Kati Tervo for their guidance and support throughout this thesis. I am grateful to M.Sc. Joonas Multanen for being the examiner for my thesis work along with Prof. Pekka Jääskeläinen. I cannot thank Topi Leppänen enough for helping with my work whenever I requested him. I am indebted to all my coworkers for a positive and supportive work environment. And finally, I would like to thank my family and friends for being by my side all through my life, studies, and during writing the thesis.

Tampere, 28th October 2020

Malik Quamrus Samawat

CONTENTS

1 Introduction 1
2 Background 3
2.1 Digital Design Concepts 3
2.1.1 Critical Path 4
2.1.2 Fan-in and Fan-out 5
2.1.3 Other Delays 5
2.1.4 Frequency, Execution Time, and Power 6
2.2 Components of a Processor 7
2.2.1 Control Unit 7
2.2.2 Processing Unit 7
2.2.3 Instruction Operands 8
2.2.4 Storage Units 8
2.3 Pipelining Concept 8
2.3.1 Example of Processor Pipelining 10
2.4 Parallel Computing 11
2.4.1 Types and Patterns of Parallelism 12
2.4.2 Single Instruction Multiple Data (SIMD) 12
2.5 Processor Architecture 14
2.5.1 RISC Architecture 14
2.5.2 Superpipelined Architecture 15
2.5.3 VLIW Architecture 16
2.6 From VLIW To TTA 16
2.7 Application-Specific Processors 19
2.8 TTA-Based Co-Design Environment (TCE) 20
3 Field Programmable Gate Arrays (FPGAs) 22
3.1 Architecture of FPGAs 22
3.1.1 Logic Block architecture 23
3.1.2 Memory Blocks and Registers 24
3.1.3 DSP Blocks 25
3.1.4 Routing Architecture 26
3.1.5 Input/Output (I/O) Architectures 28
3.2 Resource Utilization in FPGAs 28
3.3 Designing Tools for FPGAs 29
3.3.1 Xilinx-Vivado 29

4 Implemented Architecture Template 31
4.1 Wide SIMD-TTA Soft Core Baseline 31
4.2 Reduced Connectivity Architecture 32
4.3 Comparison between the Architecture Templates 34
5 Evaluation 36
5.1 Description of Architectures Based on Reduced Connectivity Template . . 36
5.1.1 The Architecture 37
5.1.2 The Assembly Code 37
5.2 Results 38
5.2.1 Critical Path 38
5.2.2 Maximum Frequency 40
5.2.3 FPGA Utilization 42
6 Conclusion 46
References 48

LIST OF FIGURES

2.1 A block design of a digital system 4
2.2 Critical path 4
2.3 Fan-in and fan-out 5
2.4 A generalized block diagram of processor components 7
2.5 An example of pipelined operations 10
2.6 A generalized view of a 5-stage pipeline architecture 11
2.7 The general idea of scalar vs SIMD operation 13
2.8 An example of a simple RISC pipelining 15
2.9 An example of a connectivity model of a superpipelined processor 15
2.10 A simplified VLIW architecture (republished with permission [12]) 16
2.11 VLIW architecture with exposed datapath (republished with permission [12]) 18
2.12 An example of TTA with VLIW-like connectivity 19
2.13 An example of a TTA design 20

3.1 An example of a logic cell 23
3.2 An example of the internals of register files 25
3.3 Generalized structure of a DSP flow graph 26
3.4 An example of a high-level description of the routing resources 27

4.1 A generalized idea of wide-SIMD TTA template [25] 31
4.2 8x128 architecture based on wide-SIMD TTA template 32
4.3 Top-level organization of the reduced connectivity architecture template . 33

5.1 32x4 architecture based on reduced connectivity template 37
5.2 Hand-crafted assembly code for the 32x4 architecture 37
5.3 Timing summary for different reduced connectivity architectures 39
5.4 Chart showing a comparison between OPS for architectures based on Thesis and Wide SIMD-TTA Soft Core templates 41
5.5 LUT utilization by different thesis machines 43
5.6 Graphical view of utilization for two 128b vector width with different precision 44
5.7 Graphical view of utilization for different vector width with the same precision 45

LIST OF TABLES

5.1 Critical Path of reduced connectivity architectures 40
5.2 Maximum frequency and operations per second (OPS) for different architectures 40
5.3 FPGA utilization of reduced connectivity architectures 42
5.4 LUT utilization/OPS of reduced connectivity architectures and architectures based on Wide SIMD-TTA Soft Core. 44

LIST OF SYMBOLS AND ABBREVIATIONS

AGU	Address generation unit for ALU
ALM	adaptive logic modules
ALU	Arithmetic logic unit
ASIC	Application-specific integrated circuit
ASIP	Application-specific instruction-set processors
b	bits
BRAM	Block Rapid access memory
CISC	Complex instruction set computer
CLB	Configurable logic blocks
CPI	Clock cycle per instruction
CPU	Central processing unit
CU	Control unit
DLP	Data level parallelism
DRAM	Distributed Rapid access memory
DSP	Digital signal processing
FF	Flip-flops
fmax	Maximum clock frequency
FPGA	Field Programmable Gate Array
FU	Function units
GPU	Graphics processing unit
HLS	High-level synthesis
IC	Interconnection
ILP	Instruction level parallelism
Kb	kilobits
LSU	Load-store units
LUT	Look-up Table
MHz	Megahertz

NRE	Non-recurring engineering
OPS	Operations per second
OTA	Operation-triggered architecture
PC	Processing counter
PE	Processing elements
RAM	Rapid access memory
RF	Register files
RISC	Reduced instruction set computer
RTL	Register-transfer level
SIMD	Single input multiple data
SIMT	Single input multiple threads
SoC	System on chip
SPMD	Single program multiple data
TAU	Tampere University
TTA	Transport-triggered architecture
TUNI	Tampere Universities
URL	Uniform Resource Locator
VALU	Vector ALU
vec-add	Vector addition
VFU	Vector function units

1 INTRODUCTION

Application-specific integrated circuits (ASIC) are microchips that are specially designed for performing a specific application. Once it is manufactured for a particular application, it cannot be reprogrammed. Field-programmable gate arrays (FPGA) devices offer a lower barrier to entry to logic design compared to ASICs due to their lower non-recurring engineering (NRE) costs [1]. Their reconfigurability makes them suitable for low volume production and prototyping as they can be updated for fixing erroneous behavior in the design [1].

FPGAs have gained popularity in logic designing due to their flexibility and accessibility. They are widely used for application-specific processing. FPGAs are re-programmable and hence, they provide a huge opportunity for customization. As a result, compared to other fixed function processors, they are more suitable for prototyping and low volume production. FPGAs can be implemented in some SoCs as well. However, one limitation of FPGAs is that they are hard to design by software engineers. This is because FPGAs require a certain level of experience in hardware designing which the software designers usually lack. But, the Occupational Outlook Handbook of the United States [2] mentioned that in 2019, the ratio of software engineers to hardware engineers is 20 to 1. Hence, it is better if the software engineers have tools to overcome this limitation.

High-level synthesis (HLS) and soft processors are ways to improve the accessibility of hardware design for software engineers. Soft processors are processors implemented on FPGA fabric. They can be customized competitively, as the devices can be reconfigured with different processors as soon as the application changes. Even though fixed-function RTL designs provide this benefit too, soft cores offer more flexibility by adding the software programmability as a way to switch the functions. Nevertheless, soft processor designing requires a detailed understanding of FPGA technology.

Transport-triggered architectures (TTA) provide enough opportunities to research on using them for soft processor designing as there have not been many exhaustive studies in this area. But previous researches [3] show that they are easy to implement, require less logic, and provide high frequency when compared with other traditional operation-triggered architectures (OTA). Their instruction encoding has explicit parallelism. Furthermore, they do not require complex decoding logic. All these benefits make them an

interesting and practical choice for FPGA implementation.

In a processor template, there are a number of register files (RF) and function units (FU). Each of these FUs are connected among themselves and to the RFs through interconnecting buses and sockets. This is known as the interconnection (IC) network. All the communications and data transfers in the processor take place through this IC network. Making these connections is expensive and TTAs make them visible to the programmer. Furthermore, due to the TTA's move programming model, the ICs can affect the instruction width. The IC network may occupy a huge portion of the processor, as well. In order to keep the design cost-competitive, it is essential to keep this network optimized and use a minimum number of connections as possible for the application to work fast. But while doing that, it should be made sure so that the performance of the processor, in terms of cycle count, does not reverse the benefits from higher clock frequency.

This thesis is a part of ongoing research regarding the reduced interconnection (IC) network for TTA processors. Here, a reduced connectivity structure for TTAs is explored, which is especially optimal for FPGA implementation. This is a very simple IC network that enables the datapath to be fully utilized while achieving high 'peak performance'. This thesis demonstrates that full utilization of the vector function units (VFU) is possible with reduced connections while improving the critical path, clock frequency, and area utilization in the FPGA.

Chapter 2 presents some background containing some basics of digital design, parallel computing, processor architecture, and some differences between application-specific processors and fixed-function accelerators. Development of TTA processors from Reduced instruction set computer (RISC) has been discussed briefly and some knowledge about the TTA paradigm and design environment is also presented. This is followed by Chapter 3, where FPGA architecture is discussed elaborately. This chapter ends with some reviews about FPGA design methods and tools. In Chapter 4, the general idea and high-level block design for the reduced connectivity template are proposed. Implementation and evaluation of the template have been presented in Chapter 5. The thesis ends with a summary process and discussion of what has been achieved and future research prospects in Chapter 6.

2 BACKGROUND

Efficient exposed SIMD datapath connectivity on FPGAs is a complicated topic. Before digging deep into the details of the topic, discussion about some relevant basics are mandatory. In this chapter, some fundamental concepts of digital design, parallel computing, and application-specific processors are introduced and briefly discussed for a better understanding of the succeeding chapters. Furthermore, processor architectures like RISC and VLIW and the development of TTA architecture are explained shortly.

2.1 Digital Design Concepts

The world shifted towards digital design from analog with the invention of CMOS-transistors and after that decades have passed in research in this field. Modern life cannot be imagined without fast processors built with microchips that can fit millions of transistors. Digital design is all about digital circuits and logic gates. Such designs consider explicit time and parallel operation of digital logic. The circuits have combinatorial and sequential logic implemented in them. Design views have various abstraction levels, hierarchy, modularity, datapath, and control.

Figure 2.1 gives a general idea of a digital system. Digital systems consist of combinatorial and sequential parts. In the combinatorial circuits, output depends only on input at the current time instant. Sequential circuits produce a sequence of input and output values, and to do that they need memory. Such digital circuits contain applications whose output values are dependent on the inputs at any specific time; past and present [4].

A microchip can be made of billions of transistors. At present, for a graphics processing unit (GPU) the highest transistor count is 54 billion MOSFETs [5]. Each of the logic gates made of transistors needs a minimum time and power to be turned on and perform the assigned work. Hence, it is very challenging for designers to make a circuit run fast with high frequency while satisfying the timing requirements. Moreover, each logic gate needs a definite amount of power to become active and execute the assigned job. The design should consume a minimum amount of power, in order to make it competitive in the market. But this vast number of transistors consume huge power which directly affects the cost negatively.

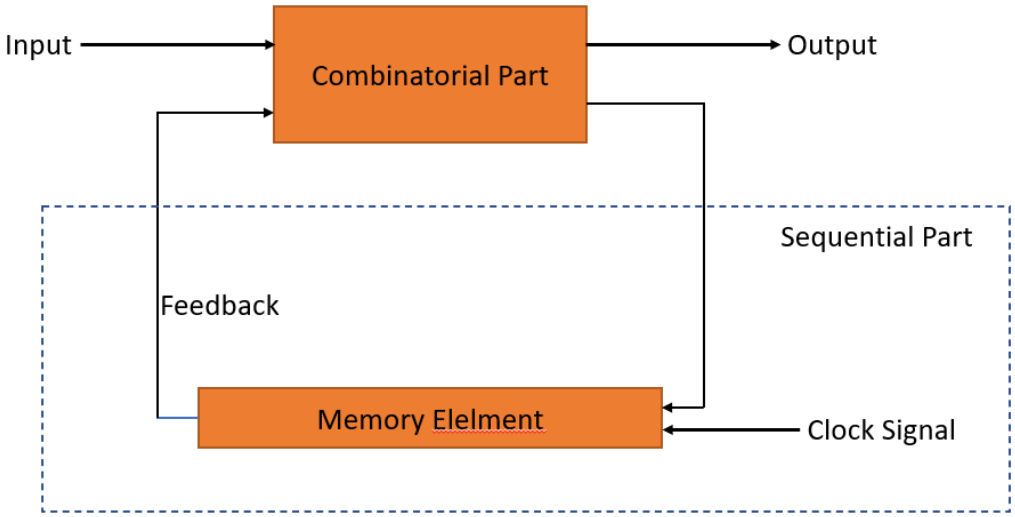

Figure 2.1. *A block design of a digital system*

2.1.1 Critical Path

In the combinatorial part of digital circuits, logic gates are connected one after another. A propagation delay takes place when input signals pass through these gates. The maximum time needed for input to change into the final output value is known as propagation delay [6]. Propagation delay is measured between 50% signal level of input and output signals. The total propagation delay is the sum of the propagation delays of each gate.

The maximum speed of a combinatorial circuit depends on the longest path a data needs to travel. It causes the longest propagation delay through the gate network. This is called the critical path. That means the critical path is the longest delay from network input to output. However, it is also true that the slowest path is not always the longest through the circuit.

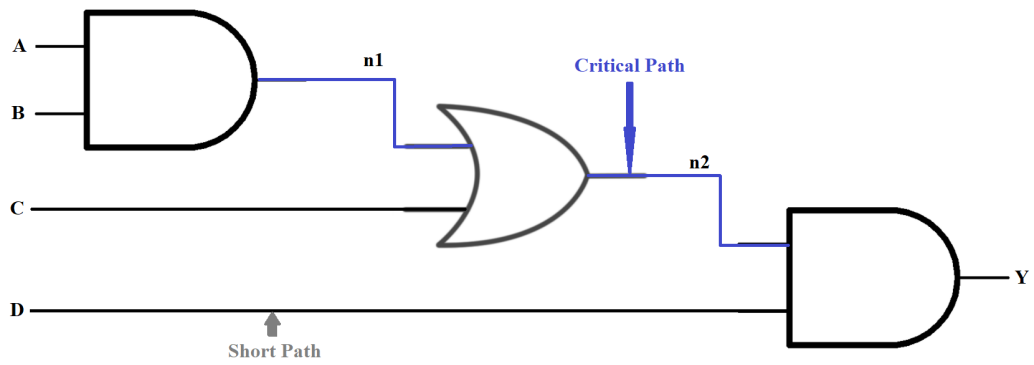

Figure 2.2. *Critical path*

Figure 2.2, is an example of a critical path. The connection in the blue line shows the critical path, which indicates the connections from `A` or `B` to `Y` via `n1` and `n2`.

2.1.2 Fan-in and Fan-out

Each logic gate may have one or more inputs and each output can be used as an input for multiple gates. The number of inputs is known as *Fan-in* in the logic gate and *fan-out* can be described as the number of gates the output can drive. Both of these are limited by physical constraints. The propagation delay is dependent on the load factor of logic gates. And, the load factor is dependent on the fan-ins and fan-outs. Therefore, gates with large fan-in are usually bigger and slower. Similarly, gates with larger fan-outs are also slower. These factors can influence the critical path quite aggressively.

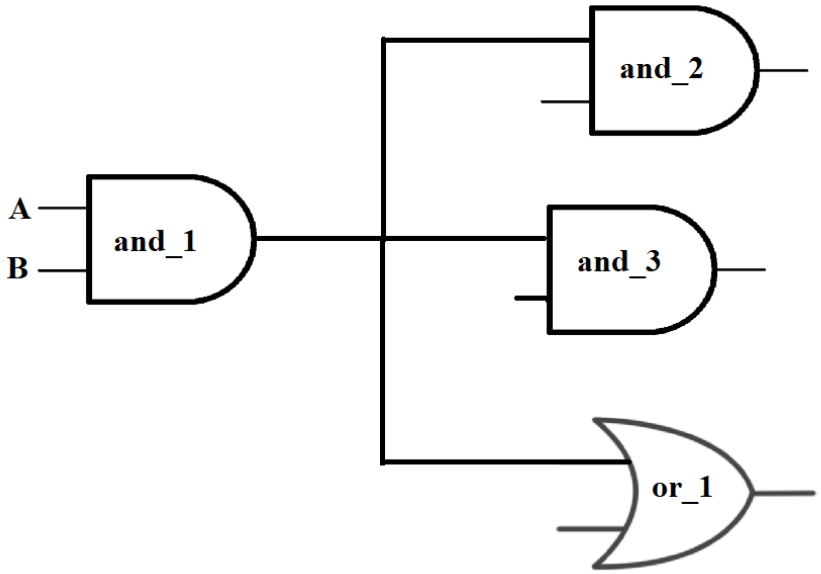

Figure 2.3. *Fan-in and fan-out*

Figure 2.3, shows fan-in and fan-outs for an AND-gate. Here, the gate `and_1` has two fan-ins, which are `A` and `B`, and three fan-outs to `and_2`, `and_3`, and `or_1`.

2.1.3 Other Delays

There are other delays or slacks as well that need to be considered when implementing a design on any device. Each signal needs to travel a certain distance or part of the system by a certain time for the system to work properly. This is known as the *required time*. In real cases, the signal might travel either slow or faster than the required time. the amount of time the signal actually takes to arrive at the endpoint is defined as *arrival time*. It is not always equal to the required time. The difference between these two is known as *slack*.

A negative slack value is an indication for a signal arriving at the endpoint later than the

required time. This causes timing requirement failure for the design, making it inefficient and unimplementable. As a result, for a design to be viable, it is mandatory to ensure that the timing is met with a positive slack.

Setup Slack = Required time for data propagation - Actual arrival time
Hold Slack = Actual arrival time - Required time for data propagation

In case of implementing the design on a field-programmable gate arrays (FPGA) device, the terms that indicate the difference between the required time and actual time of data propagation are called *setup* and *hold slack*. For a design to work accurately at a definite frequency, it must have a positive setup slack. A positive setup slack indicates that the design passes the timing requirement with some margin.

2.1.4 Frequency, Execution Time, and Power

Frequency is closely related to timing and influences the execution speed of the system. Generally, frequency means, the number of times an event has occurred in a definite amount of time. Its unit is hertz (Hz). Typically 1Hz is equal to one occurrence per second.

The time required by a processor, to complete the execution of a definite task or instruction is known as the *execution time*. It depends on various factors like the number of instructions to be executed, types of instructions and their frequency, clock cycle, etc. If multiple instructions are to be executed using the same hardware resource, the execution time will be much slower given they are carried out together than they are carried out in separate times [7].

In a processor, events or tasks is synchronized with time using a clock. This clock generates pulses at a definite frequency known as the *clock rate* or *clock frequency*. Each of these clock pulses is referred to as a *clock cycle*. Different instructions need a different number of clock cycles to be executed. As a result, if the processor has a high execution time, the maximum clock frequency of the system also increases.

The gates need some power to turn themselves on. So, for a digital system, the consumption of power gradually increases with the increase of transistors used. Developers have to keep this power consumption in check for making the design viable and cost-effective.

The high performance of the system depends on efficiently utilizing the execution time, power, area, and cost parameters. However, it is not typically possible to optimize the application in terms of all these aspects. The developer must pick the most important optimization goal and prioritize it as per the requirement of the system. For example, one could maximize the execution speed keeping a definite predetermined upper limit for power, etc.

2.2 Components of a Processor

The processor generally consists of a control unit and a processing unit [8]. Additionally, it has some instruction operands, storage units, and external buses. The external buses consist of address, data ad control buses.

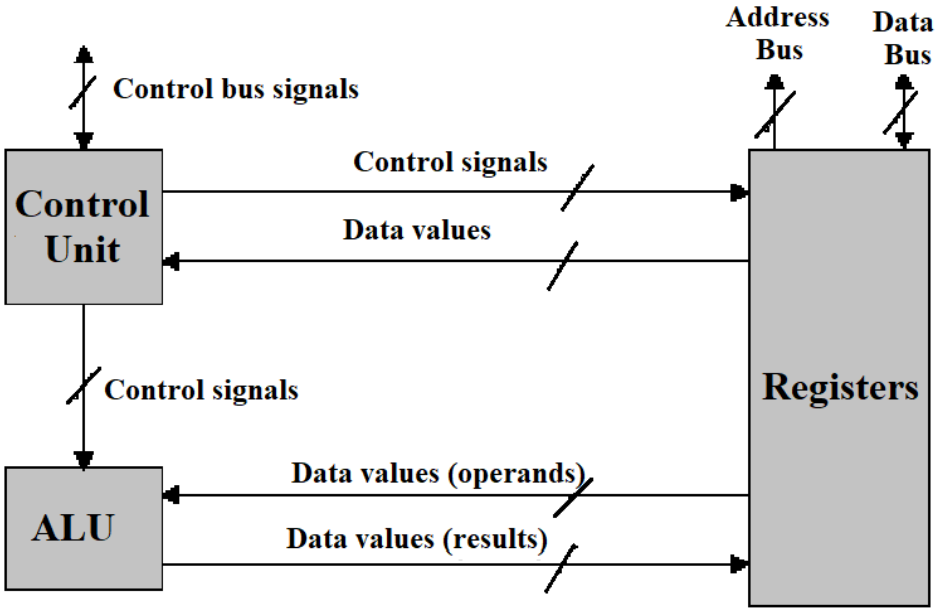

Figure 2.4. *A generalized block diagram of processor components*

2.2.1 Control Unit

The work of the control unit (CU) is to fetch instructions from the memory, decoding them, and indicating which processes are to be performed by the processing unit. The processes could be either arithmetic or logic or both. It also generates all of the necessary signals for executing the instruction. This step is known as the execution step.

2.2.2 Processing Unit

The processing unit executes the operations specified by the control unit. Handled information and intermediate results are stored in registers inside the processor or in the memory. The communication line that interconnects the control unit, processing unit, and the registers are called internal buses. These internal buses, registers, ALUs, etc. together form the datapath for performing all required operations. Some common components of a processor are discussed here in brief.

2.2.3 Instruction Operands

An instruction operates on operands. Operands are usually stored in memory cells or registers. They can also be stored as constants inside the instructions. To optimize data capacity and speed of the processor, computers store operands in different locations. Operands stored as constants or in registers can carry small amounts of data only. But, they are quick to access. Memory can store large data, but it is slow. So, additional data that doesn't need immediate access, is stored there.

2.2.4 Storage Units

Data can be stored in memory or registers. It takes a long time to retrieve operands stored in memory. But in order to run fast, instructions need to access them quickly. As a result, processors have registers in them to hold frequently used operands.

In comparison to the register, memory has more locations to store data [6]. However, accessing it takes longer. Register files are smaller than memory, but faster. However, it is possible for a program to access a huge amount of data by utilizing the correct combination of memory and registers, quite quickly.

Along with registers and memory, the use of constants is also quite common in processors. These constant values can be immediately accessed from the instruction [6]. So, they are also known as immediates. They do not require any access to the register or memory.

2.3 Pipelining Concept

Inside the processor, the datapath usually implements the fetch, decode, and execution cycles of an operation. A simple datapath is constructed with the *memory* for storing instructions, the *program counter (PC)* for storing the address of the instructions, and *ALU* for executing those instructions.

Datapaths could be single-cycled or multi-cycled. When all the instructions which a datapath has to implement are executed in one single cycle, it is known as a single-cycle datapath. Such a datapath can carry out each instruction simultaneously. So, each component in the datapath can be used only once per cycle. Hence, to make such design efficient, the datapath should either have multiple numbers of the same component or each component must have multiple numbers of inputs and outputs selected by a multiplexer. Single-cycle datapath operation is very good for the cycle per instruction (CPI) of the processor. However, it increases the critical path a lot causing execution time delays. Furthermore, as every instruction takes one cycle, all instructions move at the speed of the slowest one creating a bottleneck. It makes them inefficient for modern processors

where speed is a major aspect of measuring performance.

In a multi-cycle datapath, an operation is divided into small sub-operations so that they can be executed in multiple clock cycles. This is done to avoid long critical paths caused by stuffing all the executions into one cycle. Hence, sometimes, it can make the system run a bit faster than the one using single-cycle datapath.

A multi-cycle datapath is designed for loading, storing, and performing the computation in the fetch, decode, and execute cycle. But considering the complexity of control, a more efficient computing technique had been introduced. A single-cycle datapath in time can be overlaid for producing a different computational architecture known as pipelining. Pipelined processors have stages or segments which are arranged sequentially. Each of these stages carries out a particular task within a definite amount of time.

Pipelining is also a pattern of implementing parallelism in the processor. It helps in improving the instruction throughput of an application or digital system. Here, multiple instructions are overlapped during execution. So, pipelining can be defined as a set of data processing stages connected such that the result of one stage is used as input of the next stage. These stages, which are actually parts of different instructions, are usually executed in parallel. If the executions phase is too complex, this phase can be split into multiple cycles and pipelined. This enables the processor to implement a new operation in every cycle, even though it takes multiple cycles. This also allows storing and executing instructions in an orderly process.

While pipelining, the output of a stage is directly fed as an input to the succeeding stage. Data processing can start as soon as the first input has been received. Thus, the first results are ready earlier. In multi-issue processors, pipelining can be applied at a single core level by splitting the single instruction's execution into multiple stages.

Data-flow through these pipelines can be compared to the cars through an assembly line [9]. However, this long chain of pipelined operations causes a long datapath. Hence, such systems need to consider the worst-case delay through the datapath to measure the cycle time. The total throughput of pipelined architecture is limited by the slowest single-stage throughput i.e. critical path. Parallelism is also limited by the number of stages in the pipeline. With the increase of stages, the critical path also becomes longer.

Previous research [10] emphasized critical path analysis for a pipelined application. In a pipelined parallel application, operations are divided into small workloads. These workloads are implemented concurrently in the hardware components. Long critical paths create bottlenecks throughout these concurrent hardware components. As a result, the processing speed is slowed down affecting the performance of the application in terms of clock frequency. Hence, optimizing the application to minimize the critical path, in highly concurrent hardware, is essential for avoiding such bottlenecks. However, critical path

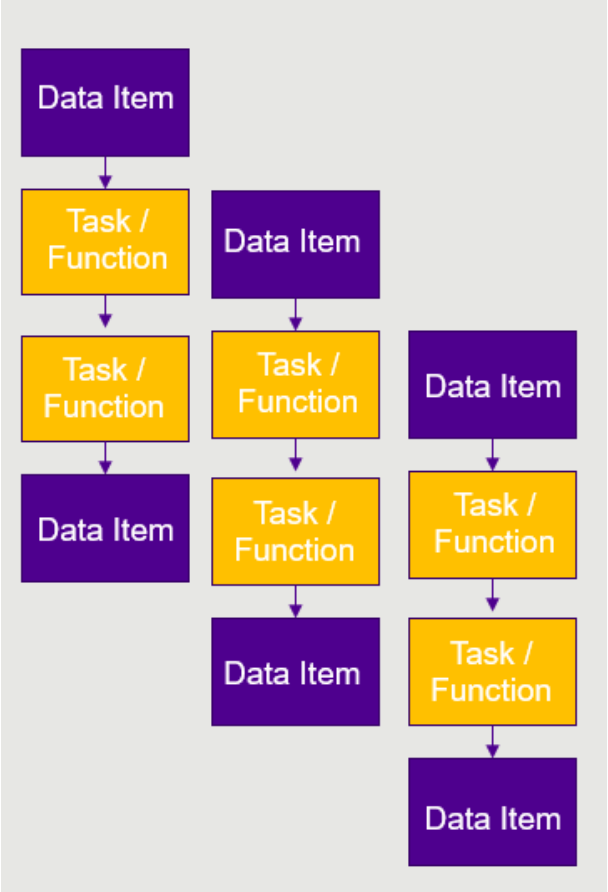

Figure 2.5. *An example of pipelined operations*

analysis needs detailed domain knowledge for understanding the gate-level delays, as it is one of the fundamental basics of the digital design concept.

2.3.1 Example of Processor Pipelining

In instruction pipelining, executing instructions include overlapping fetching, decoding, and executing stages of an instruction cycle. Such pipelining is implemented in multi-stage RISC processors. Here, a single-cycle processor is subdivided into five stages for pipelining. The longest delays in a processor are usually caused during interaction (reading from or writing to) with memory and RF or while the ALU is operating. In five-stage pipelining, each of the stages assumes exactly one of these slow steps. So, it can execute five instructions simultaneously. The entire logic is divided into five different portions into these stages. As a result, the clock frequency goes high by five times the previous one. Though the latency of each instruction has not changed ideally, the throughput is almost five times better [6].

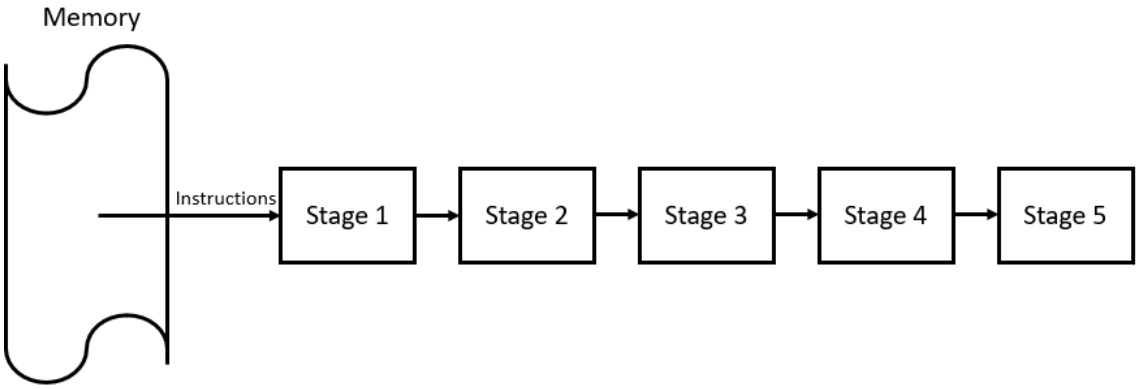

Figure 2.6. *A generalized view of a 5-stage pipeline architecture*

2.4 Parallel Computing

Speed and performance are two major specifications for processors. Increasing the number of transistors in the same chip can increase the speed to a certain limit, but it is not the most efficient way to improve performance. Furthermore, adding more transistors limits the frequency of the core and also increases heat radiation. In order to keep up with the growing demand for high-performance computing, hardware engineers rely on parallel programming to do more things at once rather than do one thing faster. Moreover, now computers have multi-core processors making it possible to implement parallel computing efficiently.

In parallel computing, multiple tasks take place at the same time. Here, a large problem is broken down into smaller parts and executed simultaneously using multiple processors. Each of these smaller parts is independent and often similar in nature. This similarity aids in executing these parts in parallel, simultaneously. Nonetheless, this is not mandatory. While executing the tasks in parallel, the resources communicate through shared memory. When execution is completed, the results are combined. The program algorithm ensures that after computing, it provides a collective output as per the set of instructions.

Parallel computing was introduced in the processor to improve throughput, latency, and power-efficiency. The main difference between concurrency and parallelism is, in concurrency, multiple tasks are interspersed alternately, and for that multiple hardware resources may not be required. But for parallel computing, tasks progress simultaneously with the help of multiple hardware resources. Since it requires more resources, it might seem to be more expensive. But it is a small trade-off when compared with the performance boost resulting from implementing parallel computing.

2.4.1 Types and Patterns of Parallelism

There are mainly 3 types of parallelism found in processors. *Instruction level parallelism (ILP)* is the ability to execute multiple operations, for instance, an addition or a multiplication, at the same time on multiple data. This indicates multiple parts of a function executing in parallel. *Data level parallelism (DLP)* is the ability to execute the same operation on multiple data. This is helpful for performing vector operations like adding multiple elements in parallel or executing the same operation from multiple iterations of a loop in parallel using a single instruction. *Task* or *Thread level parallelism (TLP)*, is the ability to execute multiple tasks at the same time, i.e. multiple functions executing in parallel.

Depending on the programs, implementing parallelism could be very easy without any need for synchronization. The algorithms could freely run in parallel without considering the other cores. But, this is not so simple in most cases. Synchronization is required among the collaborating cores for shared data manipulation in order to avoid any kind of data corruption. Hence, different patterns were introduced in parallel programming as the structure for parallel implementation. Collective patterns, pipelining, fork and join are some of the widely-used patterns.

2.4.2 Single Instruction Multiple Data (SIMD)

Single instruction multiple data or SIMD is a technique used for implementing Data level parallelism through vector coding. The same operation for multiple data items or vectors of data is executed by a single instruction. In SIMD computation, whole vectors are loaded and stored in address spaces. The addresses do not require to be continuous as SIMD does not necessarily involve serial memory operations. It is possible to read from and write to different addresses scattered in multiple memory areas in one single transaction. This memory handling technique is known as Scatter-gather.

For SIMD operations, it is essential that a single instruction or opcode defines the functionality for all vector elements. Masked SIMD instructions may have specifications defining which of the lanes are to be executed. One limitation of this type of computation is that only some workloads can be parallelized into this kind of SIMD.

In SIMD operation, processors can implement the same instruction on a number of elements. This is kind of like vector arithmetic. This execution model allows the processing elements (PE) to perform work being connected to each other through vector RFs.

Programming SIMD processors can be difficult. Even when the algorithm can be executed in vector operations, assembly level programming may be needed to utilize it [11]. However, compilers are available for SIMD algorithms. Such compilers attempt to vectorize programs automatically. Then the processor can use available operations, and libraries

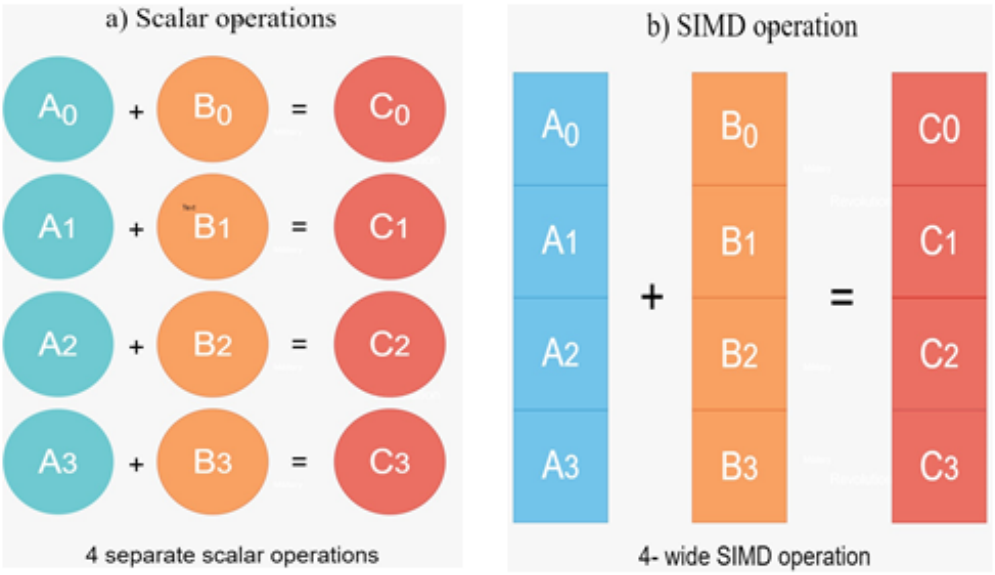

Figure 2.7. *The general idea of scalar vs SIMD operation*

with vectorized implementations of common algorithms [12].

SIMD processors usually execute one instruction at a time. Each of these instructions is applied to a set of data operands known as lanes. One instruction can be executed for all the SIMD lanes or any one of the lanes at once depending on the program specification. SIMD lanes require a lesser number of interconnects compared to]msother processors [13]. Hence, element-wise operations can be easily parallelized. However, communication among the SIMD lanes is quite complex. As the lanes are acting independently, generic shuffles, and scatter-gather memory operations require each lane to be connected to all the others [14].

To increase flexibility in parallel processing, single instruction multiple threads (SIMT) is generally applied in the graphics processing unit [15]. Here, multiple threads are executed simultaneously as warp in multiple PEs in parallel. The execution is more restricted than a multicore processor as when threads in a warp diverge from a branch instruction, it executes both paths serially causing the delay. Lower divergence means more active threads and better performance. So warp scheduling is a major challenge of SIMT architecture.

Single Program Multiple Data (SPMD) model is another way to describe massively parallel programs. In SPMD, the programmer describes what is done in a single thread or kernel instance or for a data item. This kernel description is executed multiple times to process multiple pieces of data in parallel. This is parallel execution by default with explicit synchronization. For SPMD execution, the same program is implemented at different points independent of each other by using multiple discrete processors. On the contrary, SIMD exploits lockstep on different data and requires vector processors for data manipu-

lation. GPU architectures are designed for the SPMD execution model due to their pixel or vertex processing origins. So, SPMD is quite common in GPU programming. This is also used for general-purpose CPUs.

2.5 Processor Architecture

Instruction set architecture (ISA) or architecture of a processor defines an abstract model of the fundamental features of the processor. It works as an interface between the hardware and software while specifying the supporting data types, input/output models, registers and memory structure, etc. When the processor architecture is comprehended or realized into a structure, it is known as processor implementation, or simply, implementation. The same processor architecture may be implemented differently.

There are a number of processor architectures and their implementations used for different purposes, each having many different approaches to meet their design goals. Some architectures use one-dimensional [12] [16] or two-dimensional [12] [17] [18] [19] arrays of small processing elements (PE) with interconnects between them, while others have an organization resembling VLIW processors with multiple PEs controlled by one instruction [17]. A few general processor architectures will be discussed here.

2.5.1 RISC Architecture

RISC or Reduced Instruction Set Computer architecture was initially introduced to enhance processor flexibility. It is a type of processor architecture that can process small but highly optimized instructions. They have a higher number of registers to reduce interactions with memory. Instruction pipeline is introduced in RISC processors which allows execution of different steps of the instructions simultaneously. This increases processor efficiency by facilitating optimizing the instruction set. Due to the high optimization of instructions, RISC processors have lower CPI; which means high throughput - theoretically it is one instruction per cycle.

A simple RISC processor has five stages. The first stage is *Fetch*, where the processor reads instructions from the memory. The next stage is the *Decode*. Here, the source operands are read by the processor from the register file. Also, the instruction read on the previous step is concurrently decoded for producing the control signals. In the *Execute* stage, the processor carries out operations involving the arithmetic logic unit (ALU). Read or write operations involving memory access take place in the *Memory* stage. Then in the *Writeback* stage, the processor writes the results to the register as instructed. [6]

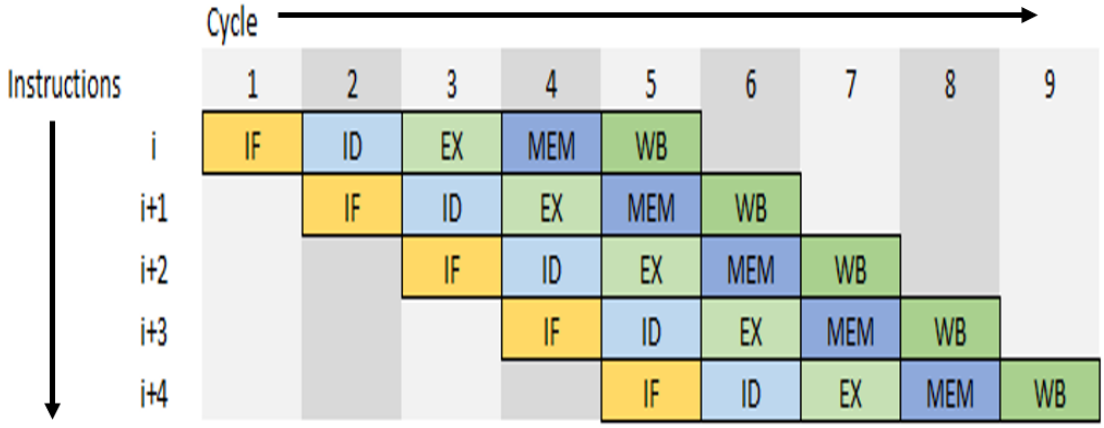

Figure 2.8. *An example of a simple RISC pipelining*

2.5.2 Superpipelined Architecture

The superpipelined architecture exploits this pipelining feature of RISC processors to improve the critical path of the processor. It can be done by pipelining the microinstructions i.e. pipelining the stages of RISC [13] further. Instruction fetch, execute and memory stages are the critical stages of pipelining. In superpipelined architecture, these stages are pipelined using bypasses. It exploits internal FU concurrency by reusing its hardware multiple times for different operations. Figure 2.9 shows an example of a connectivity model of a superpipelined processor.

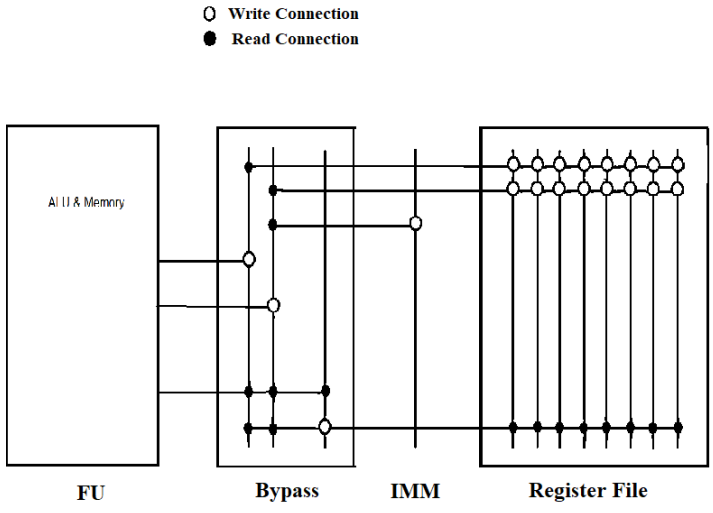

Figure 2.9. *An example of a connectivity model of a superpipelined processor*

Superpipelining effectively reduces the critical path. But, even though RISC pipelining helps to reduce CPI, superpipelining causes an increase in this CPI. The reason behind this is, it causes data hazards that cannot be resolved by the compiler. Not all delay slots can be filled effectively due to in sufficient parallelism. Furthermore, the bypassing com-

plexity grows linearly with the number of superpipeline stages. It also increases latches and area overhead. However, it does not affect the RF complexity [13].

2.5.3 VLIW Architecture

Very-long instruction word (VLIW) architectures support CISC or RISC style operations. Each VLIW instruction specifies multiple RISC operations [13]. The instruction word is divided into a number of smaller instructions. They execute independently on separate FUs. Each operation has latency. As a result, there is a possibility of the occurrence of different hazards. Some hazards can be resolved by the compiler [12].

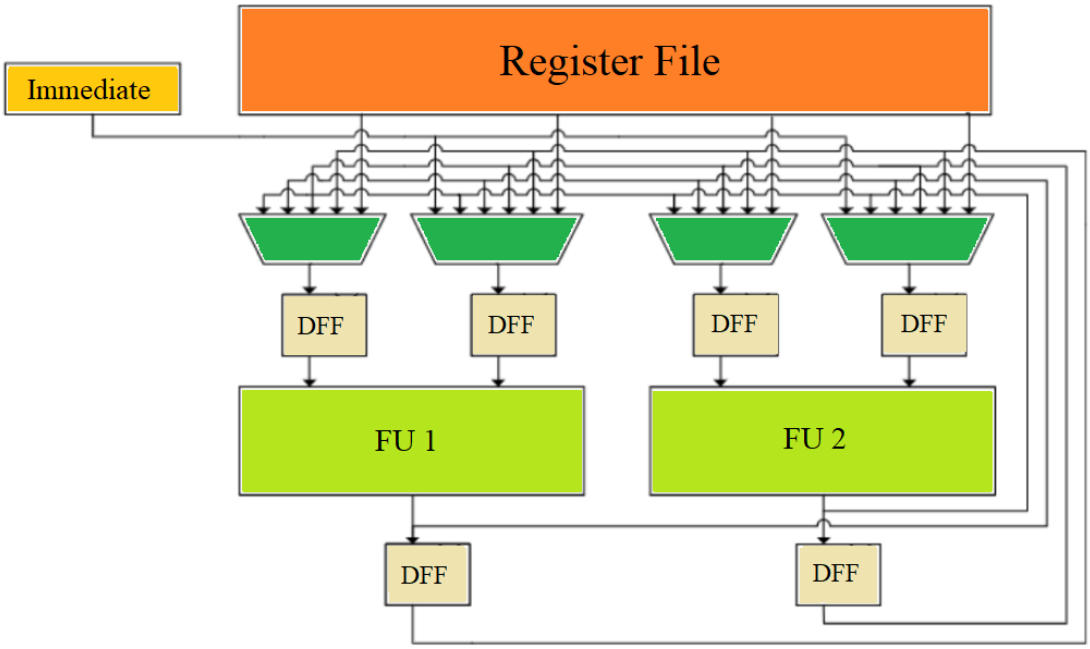

Figure 2.10. *A simplified VLIW architecture (republished with permission [12])*

Figure 2.10, shows a simplified VLIW architecture with two single-cycle FUs. For immediate values, the same bus may be shared by FUs. Each instruction can specify only one immediate. With the implementation of more FUs, the requirement for immediates also multiplies. Long immediates can be considered as a separate FU and programmed by a dedicated set of instructions [13].

2.6 From VLIW To TTA

In the 70s, transport-triggered architectures (TTA) were used in Tabak and Lipovsky's paper regarding MOVE architecture [20], where they used it for control processing. Prof. Corporaal [13] proposed the TTA approach as an improvement for VLIW architectures. Even though, researchers are still exploring various aspects of TTAs; works so far [3]

proves that they can be simply yet efficiently used for FPGA implementation. In fact, in many aspects, TTAs are very competitive to other vendor-supplied soft core processor templates. Soft-cores designed by TTAs can be highly customized. This helps to utilize all the advantages provided by FPGAs. Moreover, compared to non-specialized soft cores, TTA soft cores come with the possibility of better execution times and significantly improved resource utilization [21].

TTAs have derived from VLIW architectures and hence, these two have some basic similarities. Both architectures use static scheduling to implement explicit parallelism. This is achieved by first dividing instruction words into smaller, independently executable parts. Then explicitly defining the control unit which part is to be executed when. All of this is done before the program is run.

The flexibility and scalability of a VLIW architecture make it very useful for designing application-specific processors. However, for a typical FU with two inputs and one output, the RF requires three ports for each FU. The reason is in VLIW, it is needed to read two operands and write one operand of register data per cycle per parallel operation. It creates huge redundancy in the RF ports making it very complex and thus large and slower due to the ports needed. This is the main downside of VLIW implementations, especially on FPGAs. Bypassing can help by improving the latency, but still, the hardware programming model requires enough number of RF ports to satisfy worst-case access patterns or expensive dynamic scheduling logic for them. [12]

With the TTA approach, scalable support for ILP can be provided. But only ILP is not enough for obtaining high throughput. DLP support is also combined with it by introducing a SIMD operation. TTAs can effectively reduce RF and bypass complexity. Reducing RF complexity means reducing the number of ports in the register files. Implementing static control to the IC could be a solution for this. RF can be implemented as a separate FU having a limited number of read and write ports. [13]

Bypass complexity can be reduced by implementing parallel FUs for multiple functionalities. This enables more concurrency without increasing hardware requirements significantly. FUs having multiple outputs and superpipelined FUs can produce results at different latency aiding in reducing bypass capacity as well.

Figure 2.11 shows an exposed datapath VLIW architecture. Here, the source of every operand and their destination in the RF is clearly specified in the sub-operations and register indices respectively. As a result, the operation is quite expansive. For TTAs, the instruction word is longer than a similar VLIW architecture. This is a major drawback for TTAs.

Comparing figure 2.11 with figure 2.10 points out remarkable differences. The bypassing registers and the connecting buses present in figure 2.10 are not there in figure 2.11.

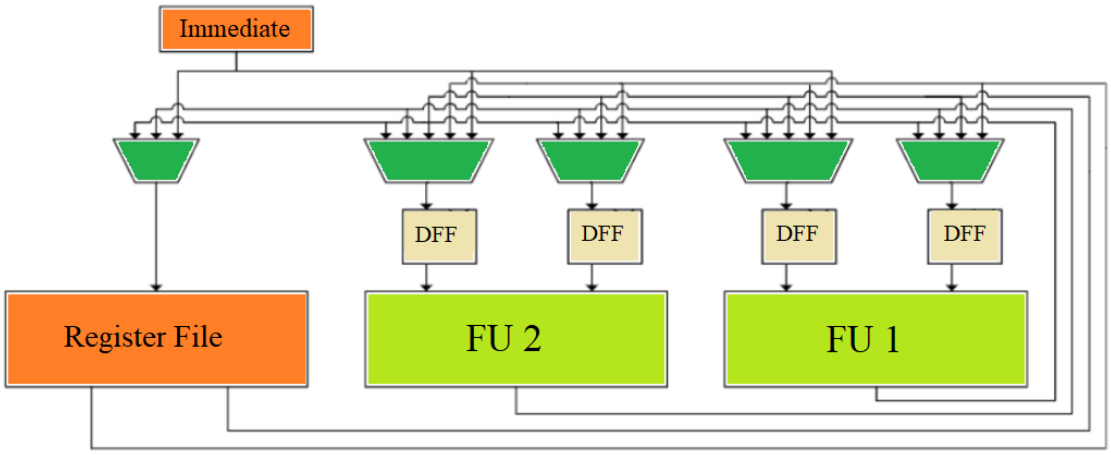

Figure 2.11. *VLIW architecture with exposed datapath (republished with permission [12])*

But internal bypassing is needed for read/write operations inside the RF. An extra MUX is present there for this purpose. This architecture is much cheaper to implement compared to an RF with multiple ports as the bypassing is restricted within one FU.

Though TTA architectures can support most cases where the use of RFs can be eliminated, their scheduling is quite limited. All operands are specified at once and hence, they are read at the same time. Subsequently, the possibility of bypassing the results from previous cycles is reduced. In order to introduce more scheduling opportunities, data transports towards operand ports are separately presented. Operations start when a transport is issued to a preset trigger port. This is the major difference between TTAs and OTA.

Since the operand values need to be stored while waiting for the starting of the operations, TTAs can use registers in operand and result ports. By decoupling data transfer of an operation's operand and results, this design allows for the final optimization where successive operations in the same FU can reuse one of the operands. [12]. Figure 2.12 represents an example of a similar TTA design with connectivity similar to VLIW.

In a VLIW template, all the FUs, RFs, and CU are connected with buses through input and output sockets. Together they form the IC Network. Each bus can carry out one data for an FU. Therefore, for implementing parallelism, the number of buses should be proportional to the number of FUs. That means the number of buses grows linearly with the number of FUs. As a result, the IC network design for a large system becomes quite impractical. Furthermore, even though more buses indicate a higher level of ILP, a program can only carry out parallelism to a certain limit.

TTA architecture can provide a solution to this IC network problem. IC network can be optimized by merging rarely used buses together. This helps to customize and remove rarely used connections. This simplifies the hardware implementation and reduces the

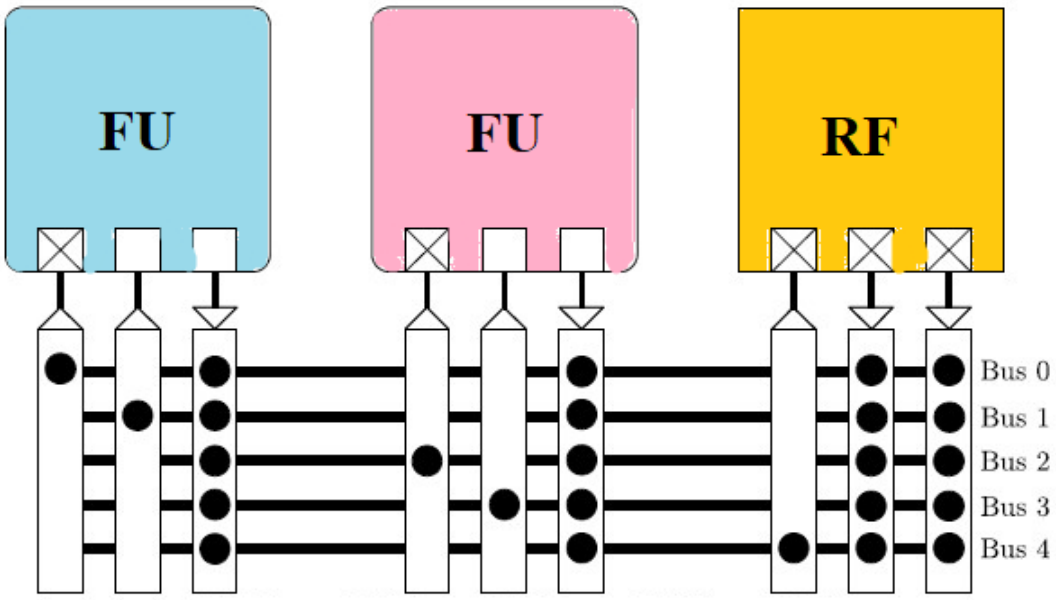

Figure 2.12. *An example of TTA with VLIW-like connectivity*

width of the instruction word. [12]

For a fixed set of features, FPGAs are a practical choice than ASICs for their lower NRE costs [1]. This enables vigorous tailoring of soft processors for the required application. TTAs can utilize this aspect due to their larger design space resulting from a highly customizable IC network. Especially, pruning the IC of rarely used connections has a direct impact on the multiplexer size.

The TTA approach is also flexible to use for the transformation of RISC or SIMD processors. A previous study [12] has proved, while transforming from OTA to TTA, due to static scheduling, TTA implementations do not require additional logic for resolving pipeline hazards. The compiler takes care of this issue. Moreover, for TTAs, the instruction word maps more directly to the multiplexer control signals making instruction decoding quite simple. These combined factors result in higher maximum frequency and better performance. A previous study [3] also points out that TTAs are very convenient as small soft processors.

2.7 Application-Specific Processors

Application-specific processors are aimed to perform some specific set of tasks for any particular purpose or functionality. Some common examples of such processors are network interface chips, cell phone chips, graphics accelerators, etc. Application-specific instruction-set processors (ASIP) can be implemented as new ASICs or in FPGAs as soft cores. They are used for specific applications providing a trade-off between flexibility and performance.

FPGAs, on the other hand, are extremely flexible and thus, also suffer from the flexibility in the overheads. It can be instantly programmed to work as per the design requirement of the project. The designs running on FPGA's are generally created using hardware description languages like VHDL and Verilog. They have limited operating frequency and consume more power compared to the same digital logic implemented directly on the chip, without FPGA. Their re-configurable aspect gives great flexibility for creating SoC prototypes and validating designs [22]. More about FPGAs is discussed in 3.

2.8 TTA-Based Co-Design Environment (TCE)

There is a specific set of tools available to aid hardware-software co-designing for the TTA processors. This is known as the TTA-Based Co-Design Environment (TCE) [23]. It has attributes like graphical interface for processor design, retargetable compiler based on LLVM [24], and instruction set simulator. The task of this compiler and cycle-accurate simulator is to re-target the designed architectures. It can be used to generate synthesizable RTL implementations of the designs [25].

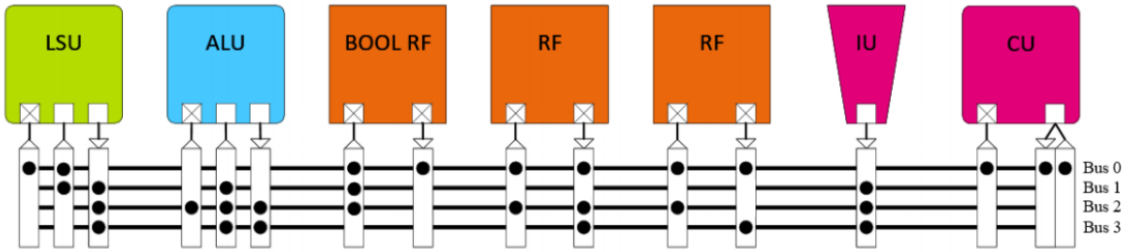

Figure 2.13. *An example of a TTA design*

Figure 2.13 represents a TTA processor template designed using the TCE toolset. The template generally consists of a number of FUs and RFs connected by the IC network. Every unit has one or multiple input and output ports. One input port of each FU must have to act as the trigger for the operations of that specific FU. The operands and results can be bound to any of the available input and output ports respectively, at any cycle, as per the requirement of the program. However, the cycles in which read /write operations take place specifies the latency for the FU.

For instruction fetch through branching and subroutine call, a control unit (CU) is introduced in the design. It defines the branching latency and number of delay slots. IC network is composed of buses, each of which corresponds to one move in the instruction word. A move can be issued between any two source and destination sockets connected to the bus. Any socket can be connected to any combination of buses and to any number of ports. Each bus may have short immediate value. For long immediates, separate FU is used.

There are a number of parameters for the processor generator that would generate the

RTL description as per requirement. The CU also has a number of optional features. FU and RF implementations are picked from the hardware database. FUs can also be constructed using VHDL or Verilog codes describing each operation.

3 FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

Field Programmable Gate Array (FPGA) is a semiconductor device family that offers a lower entry barrier than ASICs [1] to the field of logic design. They are widely used for their re-configurable architecture and accessibility than other similar devices in the market. Renowned cloud service providers like Amazon Web Service [26] and Huawei [27] have servers with FPGA devices.

FPGAs are built with a series of logic blocks. These are called configurable logic blocks (CLB). Logical cells made of LUTs are stacked into a block to create each CLB [28]. These blocks are connected through interconnects which can be programmed as per requirement. FPGAs can be reprogrammed depending on the functionality requirements of the application [29] making them compatible with designing application-specific processors. This is one of the major reasons for their popularity and wider adoption over ASICs.

This chapter provides an introduction to the general concepts related to FPGAs, their working procedure, and configuring techniques. This chapter also discusses in detail the architecture of different blocks inside an FPGA device and ends with a short discussion about the design tool Xilinx-Vivado that has been used for this thesis.

3.1 Architecture of FPGAs

The FPGAs are nowadays frequently used technology for electronic system developers. Due to their reprogrammable nature, they offer easy designing, debugging, and implementation for hardware applications. Compared to developing a custom silicon prototype for each application which can cost millions to produce and test, reconfiguring FPGAs is cost-efficient as well as saves both time and money.

With time, FPGA architecture has evolved significantly. FPGAs currently used can efficiently accommodate an optional system on chip (SoC) [30], integrating programmable logic with hard processor cores. FPGAs are specially differentiated from other similar devices for their architecture and computer-aided design tools [31]. Designing for FPGAs requires an immense understanding of these technologies. Modern architectures provide a wide span of resources for implementing desired functionalities on FPGAs.

3.1.1 Logic Block architecture

FPGAs are built with programmable logic blocks for implementing logic functions, programmable routing for interconnections, input/output blocks, etc. The utilization of FPGA devices is measured by the utilization of look-up tables (LUT), DSP blocks, and RAM blocks present in them. FPGA computing is largely dependent on LUTs as the LUT-based structure of programmable logic blocks is widely popular and is used to implement arbitrary logic functions.

The concept of lookup tables (LUT) is quite simple to understand. This is a customized truth table loaded with values relevant to the FPGA, based on the programmer's specific needs and instructions. There are some values arranged in the cellular form in a lookup table. A combination of input signals is used as a pointer to search for results. Each of the input combinations points towards a specific cell containing the desired value. A LUT can be considered as a small piece of RAM that is loaded whenever the FPGA chip is powered up.

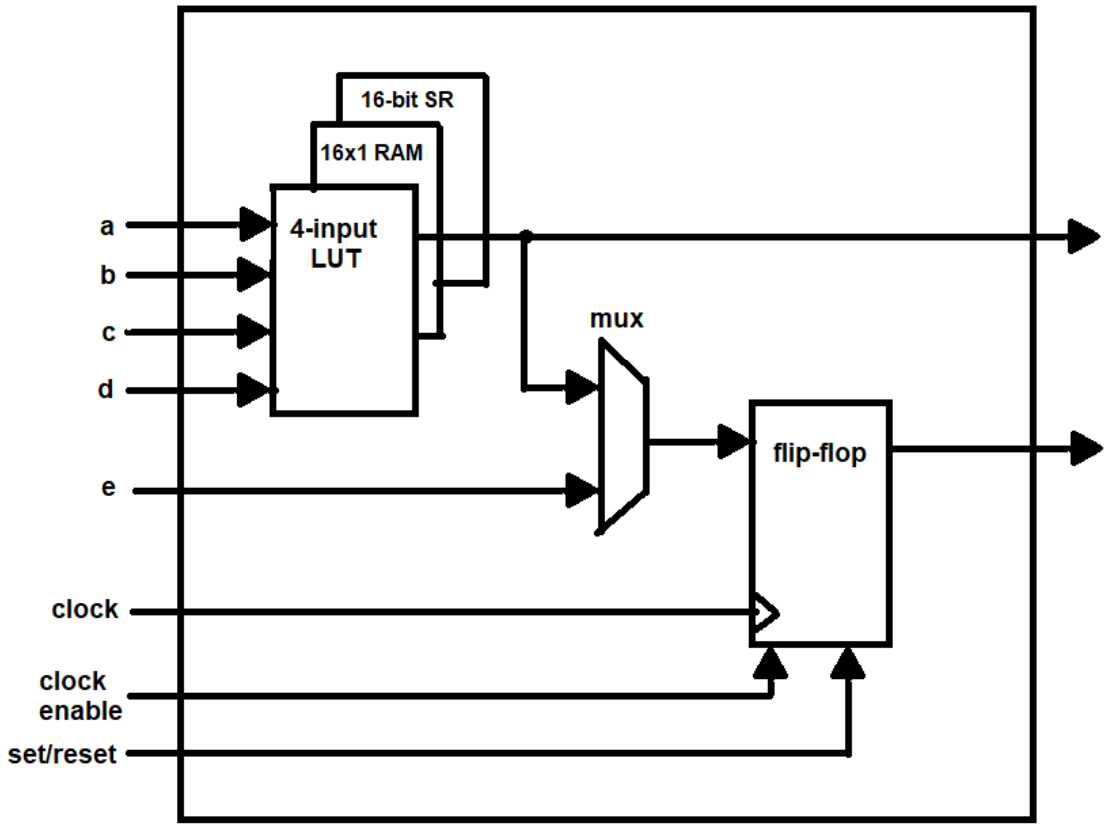

Figure 3.1. *An example of a logic cell*

LUTs are implemented by a number of inputs and outputs. An n-input LUT is able to carry out any n-input combinatorial logic function. More complex functions can be represented

by adding more inputs. But with every added input, the number of SRAM cells is doubled. Initially, 3-input LUTs were used for FPGAs. Gradually they evolved to have 3-, 4-, 5-, and even 6-input LUTs. This kind of development allowed developers to create devices by combining LUTs of different sizes (E.g. 3-input and 4-input LUTs). This was believed to imply optimal device utilization. But for logic synthesis, synthesis tools worked best under uniformity and regularity. [32]

As FPGA vendors are moving forward to bigger LUTs, the chances of spending all of a larger LUT on a smaller logic function increases. Therefore, they are adopting LUT architectures which can be used as multiple smaller LUTs, to avoid this issue. Xilinx devices can be used as examples. Presently used Xilinx devices have LUTs that can implement 6-input, single-output, or even 5-input, 2-output logic function. This enables the LUT to apply multiple smaller, such as 3-input and 2-input functions. Thus, the input and outputs of the LUTs can be customized as per the desired application. [12]

Figure 3.1 shows a simplified example of a logic cell (LC). This is an example of a modern FPGA core cell. Different inputs of the circuit such as polarity of the clock, enable port, and set/reset port can be configured as per requirement. The registers can also be configured in a way, so they can be used as latches or flip-flops as needed by the application. [32]

3.1.2 Memory Blocks and Registers

FPGAs need the support of DSP blocks and memory blocks for enhanced performance. They contain a number of small memory blocks or modules. All contemporary FPGAs contain memory blocks that comprise most of their area. Each of these can be used independently or combined to form larger memory blocks of any combination. Furthermore, they offer a variety of configuration options, such as multi-port or multi-registered functionality allowing simultaneous read and write operations [31] for data and address. This is one of the major advantages of FPGAs.

Register files are implemented in FPGAs as storage elements in multiple ways. Using block RAMs (BRAM) is one way to implement RFs. These are fast to access, large, and offer flexibility. Generally. temporary data are stored in register files so that ALU, floating-point units, or any computational engine of the digital application can easily access them. These register files are usually implemented with the adaptive logic modules (ALMs), each of which consists of two D-flipflops (D-FF). That means a 32-bit register has 16 ALMs. Static memory blocks can also be used for this purpose. Arithmetic blocks also have register files that can be used for pipelining arithmetic operations and bypassing.

An example circuit of a register file is shown in figure 3.2. In order to put the data on the data line or bit line, the register select signals (eg. sel_reg0, sel_reg1, etc.) are used for enabling the correct register. WE generates the control signal which is used for

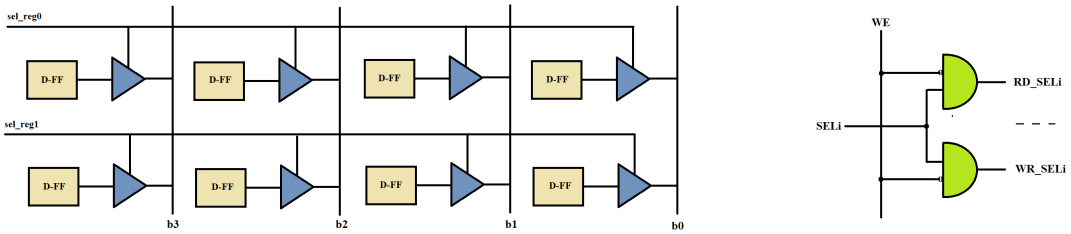

Figure 3.2. *An example of the internals of register files*

determining read/write operation in the register. The register identification (reg_id) points out the register to be accessed with the help of a one-hot code-word generating decoder that selects the desired register.

On-chip memory for processors can be built using individual registers. These are efficient for the smallest register files. However, smaller memories are not as flexible as the individual registers. But, they are faster and have better resource-efficiency. For example, Xilinx devices can provide multiple read ports. But the write port is restricted to one [12].

For larger memories like caches or local scratchpad memories, FPGAs are based on static RAM (SRAM) cells. These cells are reconfigurable and can be configured multiple times without any damage. The SRAM blocks usually have two bidirectional ports. These memory blocks have a word width of power of two. But for storing, their widths are slightly over the nearest power of two. [12]

3.1.3 DSP Blocks

Even though LUTs are the most flexible component of FPGAs, these are not enough for the implementation of complex arithmetic. Furthermore, only LUT based architecture causes huge power and area consumption.

To enhance performance and accelerate arithmetic computations, hard blocks are used nowadays. These blocks include embedded memories and integrated DSP blocks. Using the DSP blocks which are entirely made of dedicated silicon confirms the better performance and lower power usage [33].

In order to enable DSP processing, today's FPGAs have discrete multipliers. DSP applications constitute a pipelined network of DSP operations and data flows through these pipelines. Finite impulse response or FIR is an example of DSP filters.

An FIR is made of some adders, multipliers, sample delay blocks, and memories for coefficients. Multipliers and adders are used for implementing the pipelines for filtering or any other DSP applications. This pipelined network can also be implemented as an FPGA circuit.

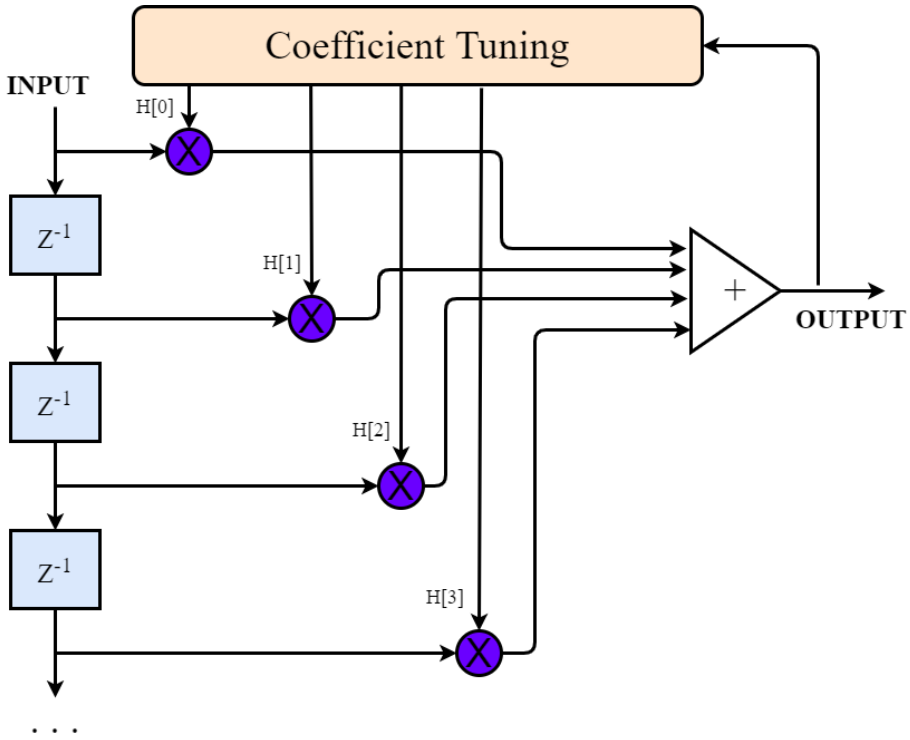

Figure 3.3. *Generalized structure of a DSP flow graph*

According to the manual published by Xilinx [30], their FPGAs support operation up to 500 MHz with the provision to use up to four levels of pipelining. Xilinx FPGAs consist of DSP blocks equipped with an 18x25-bit multiplier, a 25-bit pre-adder, and a 48-bit accumulator.

Kizheppatt et.el. [34] states Altera, which is now a part of Intel, uses ALMs as the basic building blocks in their FPGAs. ALMs contain an 8-input LUT that can be split as per the required combination of input and output, 4 flip-flops, and auxiliary circuits like adders and multiplexers. In addition, memory blocks in columns and variable precision DSP blocks were added for efficient implementation. This kind of combination of various logic elements, routing resources, DSP, and memory blocks enables developers to implement partial reconfiguration.

3.1.4 Routing Architecture

FPGAs have programmable routing for providing connections among different logic blocks to a custom circuit designed by the user. This routing is made of wires and programmable switches which help to make the required connections. Programmable routing structures contribute to a major portion of the FPGA area, its delay, and power consumption. Hence an optimized routing architecture ensures the overall performance of FPGAs [35].

Intel shows a high-level description of the routing network of their Stratix 10 devices [36]. These devices have registers embedded in their routing network for allowing the synthesis tools more flexibility in register placement [12] [37]. General-purpose routing resources

are higher in quantity with a high degree of connectivity through switches. There are also fixed connections between logic elements. Figure 3.4 is an example of a routing network.

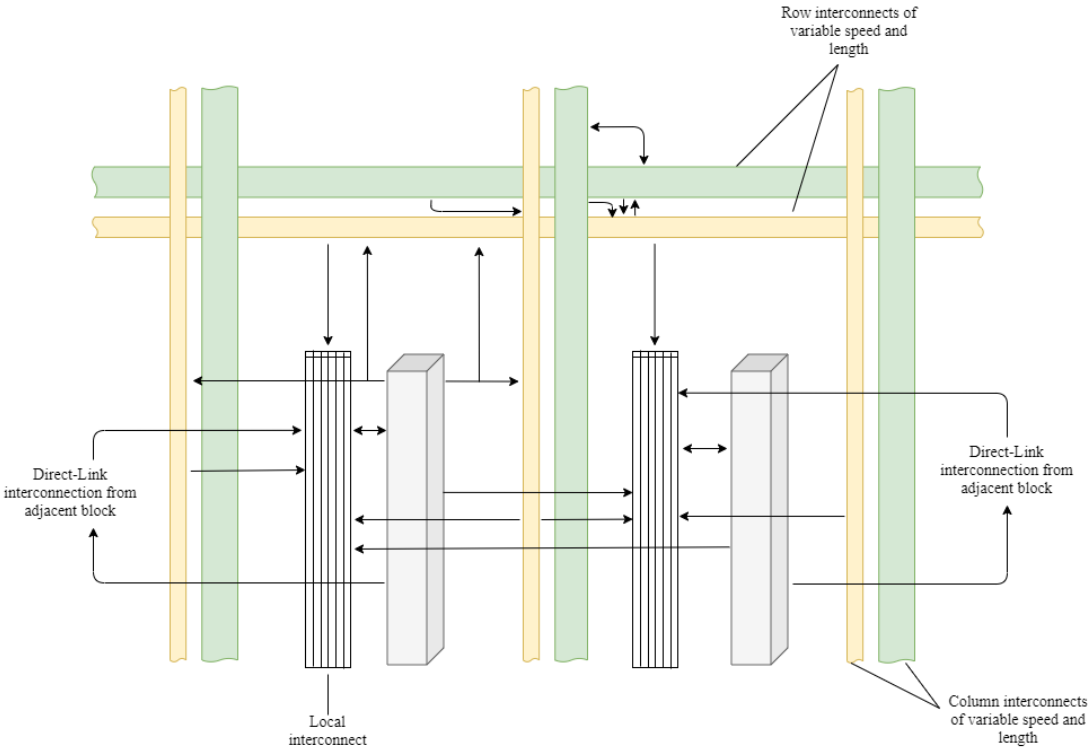

Figure 3.4. *An example of a high-level description of the routing resources*

The IC network in the FPGA consumes the most power as all communication among different blocks takes place through it. This is why routing architecture is of ultimate importance when power optimization is considered. FPGAs use routing channel segmentation. These routing tracks are then divided into wire segments. Each wire segment is of a different length.

Generally, the use of shorter segments provides better routability and reduces power consumption. It also decreases excess net loading and gives high logic density. However, short segmented net routing means more switch points and hence, higher delays. Using longer segments can minimize delays. But they need more routing tracks causing more power consumption and low logic density.

The exact routing details vary from vendor to vendor and device families. However, some general rules apply in all cases. Researches [38] [39] have proved that a combination of segment lengths in different routing tracks can significantly affect IC performance. This, in turn, has a great impact on overall FPGA performance. Therefore, an optimum average segment length combination is chosen for various routing tracks in practice.

3.1.5 Input/Output (I/O) Architectures

The logic, memory, DSP and routing structures described so far serve as a general-purpose platform for different applications. This platform needs to communicate with external components that are to be connected to an FPGA. This communicating interface is established through dedicated input/output (I/O) channels on FPGAs.

The I/O ports along with the logic and circuit supporting them are together known as I/O cells. These cells are very important components of FPGAs. They work as the interface for setting the external communication rate and speed. These cells and their surroundings occupy a large segment of the FPGA area.

General-purpose logic structures are usually able to implement any digital function. But I/O cells are restricted to implement the standards pre-selected by the designer. So, some trade-offs are necessary while designing. Hence, the selection of standards supported by it is an important decision for I/O architecture design and should be carefully chosen. But, making this decision is a complicated job. Selecting more standards causes a notable increase in the area required by the I/O cells. With each supported standard, the pin capacitance might increase, limiting the peak performance [31].

In contrast, the flexibility offered by FPGAs includes the ability to support different signaling standards. So, a limited number of standards directly conflict with the flexibility aspect. As a result, in most cases, these decisions are taken depending on the business factors of the vendors rather than the technical factors [31].

3.2 Resource Utilization in FPGAs

The generally accepted norm, for an architecture to be efficient, is that it should occupy the minimum possible area without hampering the operations it is supposed to perform. This indicates the design can fit into smaller FPGAs or it can perform better in the same space. Each FPGA contains a fixed amount of resources. Not using some of these resources does not necessarily mean saving those resources.

If a larger design can be fit into the FPGA, more throughput can be achieved at a theoretical maximum, as well as the number of vector additions or any other operation. If the design and the comparison have the same arithmetic and SIMD width they should have the same (or very similar) DSP utilization; the improvement comes from the leftover space, the difference will be in LUTs.

3.3 Designing Tools for FPGAs

Algorithms can be implemented on FPGAs by RTL design. This is a very low-level design method. Furthermore, despite some general conceptual similarities, the design language is very different from other software programming languages. So, high-level synthesis tools are used for transforming software programming language to a description which the synthesis tool can use for implementing the design on the FPGA.

Overlay architectures are a collection of coarse-grained components. This is another method for mapping algorithms to FPGAs. These components can be implemented on top of FPGA fabric, allowing higher levels of abstraction for programmers.

3.3.1 Xilinx-Vivado

Xilinx is one of the leading semiconductor manufacturers of standard FPGA chips. For this thesis, Xilinx FPGA ZYNQ Z7020 has been used for evaluation and verification. Vivado is the design tool used for programming this FPGA. Vivado design tools aid in all phases of FPGA designs. This offers support for loading design, simulation, synthesis, placing and routing, generating bitstream, debugging, and verification. It contains services for the development of software targeted for FPGAs as well.

The working procedures for this tool are clearly explained in different manuals, books [30], and internet sources [22]. This is revisited here in short. After loading the necessary files into the tool, the synthesis would be the first step towards processor generation. The RTL code is architecture-independent. Synthesis maps this code into primitives specific to the technology. Although synthesis tools are there to spare users from details, having knowledge of device primitives helps in fine-tuning synthesis behavior. Synthesis transforms the behavior code into a gate-level netlist. When the user loads the HDL based design, a syntax check starts the process. Then it is optimized by logic reduction. Redundant logic is eliminated and the size of the design is reduced. This makes it faster to implement. Finally, the technology is mapped out by connecting the design to the logic, estimating the required time, and executing the design netlist. It is then saved for the next step

Synthesis is followed by implementation. Here, the layout of the synthesized design is determined. At first, the tool gathers all constraints, such as - assignment and position of the pins, maximum delay, etc., set by the user along with the netlist files. After that, the tool compares the resource requirement specified in the netlist files to the resources available on the FPGA. The circuit is divided into logic blocks. The entire design is mapped-out in specific logic blocks into the FPGA. Next, all signals are connected and routed according to the user-defined constraints among all the logic blocks and I/O blocks.

The next step is to load the mapped out and completely routed design into the FPGA. For

this, Bitstream file is generated and loaded onto the FPGA board using a flash programmer device. After that when the FPGA is run, the board acts as the designed application.

Different reports are available after synthesis and implementation. The timing summary shows different types of slacks. The difference between the arrival time for the timing path and the required time is called slack. When a signal travels from the starting point of the timing path to the endpoint through the combinatorial logic within the time limit that is required for the circuit to operate correctly, it is called a positive slack. On contrary, for a negative slack, the data signal exceeds that limit resulting in the failure of correct circuit operation.

The design has to meet the minimum and maximum period, high pulse and low pulse time requirements, etc. for each clock pin. It is checked through the pulse width report. There are some requirements for maximum skew between two clock pins. This report checks whether this requirement is met in the implemented design at the same instance. There are also reports regarding power consumption, utilization, etc. to help in making decisions.

There are simulation checks available at every level. Usually, behavioral simulation is done at the design entry level. functional simulation is done after synthesis and timing simulation is done at the implementation level.

4 IMPLEMENTED ARCHITECTURE TEMPLATE

The TCE toolset can automatically generate RTL description for a processor architecture and a parallel program by compiling a higher-level program to this architecture. But this default RTL can be further developed with a better optimized TTA architecture for FPGAs. Various research is going on regarding the implementation of various modifications in the TTA architecture to make the design more efficient for FPGAs in regards to better speed, power, and area. However, these improvements are always a bargain among available resources and costs.

In this chapter, the basic idea regarding the exposed SIMD datapath connectivity on FPGAs is explained. The architecture template previously used in the TTA-SIMD Soft Core Processors [25] conference paper, is also described in short. Furthermore, some light is shed on how the new reduced connectivity, theoretically, is more efficient than the old template; the details about which will be discussed in the next chapter.

4.1 Wide SIMD-TTA Soft Core Baseline

In this section, a wide SIMD-TTA softcore template, used in the paper is discussed. The simplified template is shown in figure 4.1. This template has been drafted with the help of the TTA-based Co-design Environment's (TCE) [40] extended version. Some features for SIMD architectures [41] have also been implemented in this template.

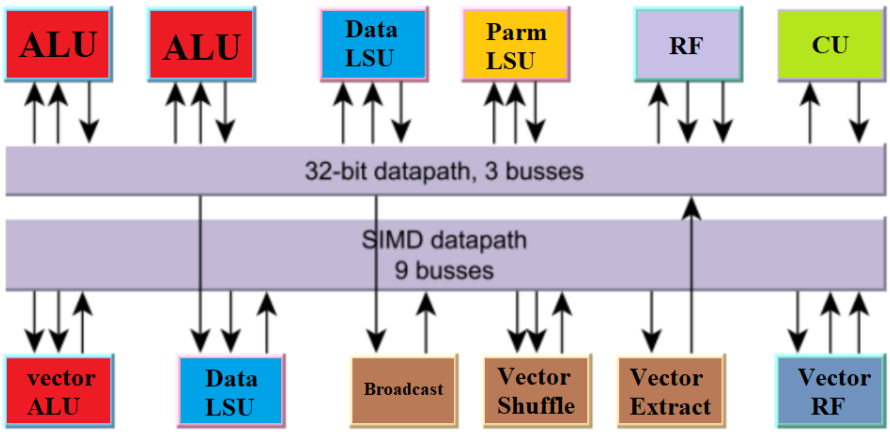

Figure 4.1. *A generalized idea of wide-SIMD TTA template [25]*

The TTA-based SIMD machines demonstrated in the paper has one set of buses for scalar FUs and another set for SIMD FUs. Scalar FUs are used for branching and calculating addresses, whereas SIMD FUs execute the computations mostly. The connections among the busses ensure that there are no unnecessary and hardly needed connections. This helps in minimizing logic utilization. There are two identical scalar ALUs complemented by a 4-cycle barrel shifter. Each ALU can perform basic arithmetic, comparison, and bitwise logic operations. For the SIMD ALU, the operations, latency, and vector width can be customized based on the application. The SIMD and scalar busses can communicate with each other through two FUs. These are a scalar-to-SIMD broadcast operation and a SIMD-to-scalar element extraction operation. There is another FU containing a set of predefined shuffle operations for reducing the implementation complexity further. There are two LSUs. One of them is aligned with the vector load and store. The other one is aligned with 32-, 16- and 8-bit loads and stores. The scalar LSU shares a 32-bit port to local memory with the slave AXI interface accessible from the ARM processing system. This helps in reducing logic complexity.

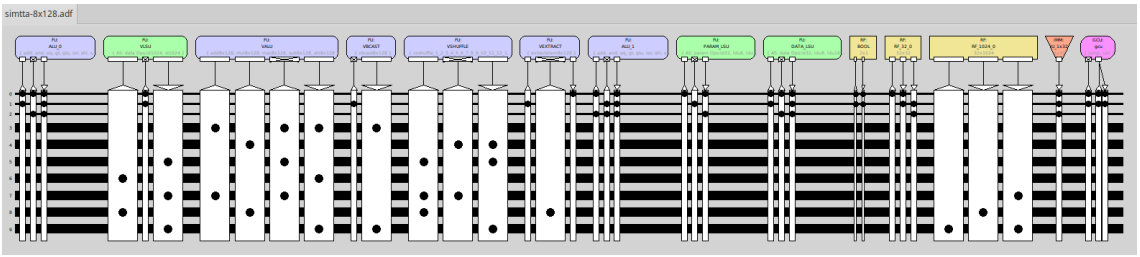

Figure 4.2. *8x128 architecture based on wide-SIMD TTA template*

In figure 4.2, an architecture based on the wide-SIMD TTA template [25] is shown. This architecture is for an 8x128 (1024b) machine.

4.2 Reduced Connectivity Architecture

The interconnection (IC) network is one of the basic yet crucial components for the TTA processors. It could be the largest component to affect the critical path. This is due to the reason that instructions and data are transferred among the FUs and RFs through these interconnections (ICs). Hence, modification in this IC network can highly affect the TTA performance. This reduced connectivity architecture refers to connectivity reduction for the data path only. Control path and control signals are out of the scope of the thesis.

Many TTA templates have been successfully implemented and tested on FPGAs previously. This thesis is a part of a research that studies different connectivity strategies for TTAs for implementing on FPGAs by digging deep into the TTA interconnection (IC) network structure laid in different FPGA fabrics. But instead, Here, an early result of the research is presented by exploring a reduced connectivity strategy for TTAs especially optimal for FPGA implementation.

The IC design template is to be implemented manually mostly. While optimizing, it is possible to access some automated IC optimizer. All FUs will not be connected to the RFs directly. The plan is to connect the LSUs to the RFs through the ALUs.

Each architecture based on this template must proficiently carry out some specific workloads when programmed. Since the IC network will be manually designed and there is no typical connection among all FUs and RFs, the compiler available in TCE might not be able to load any high-level language to the designed processor template. But it should work for hand-crafted assembly programs. Subsequently, we verify each architecture with assembly code.

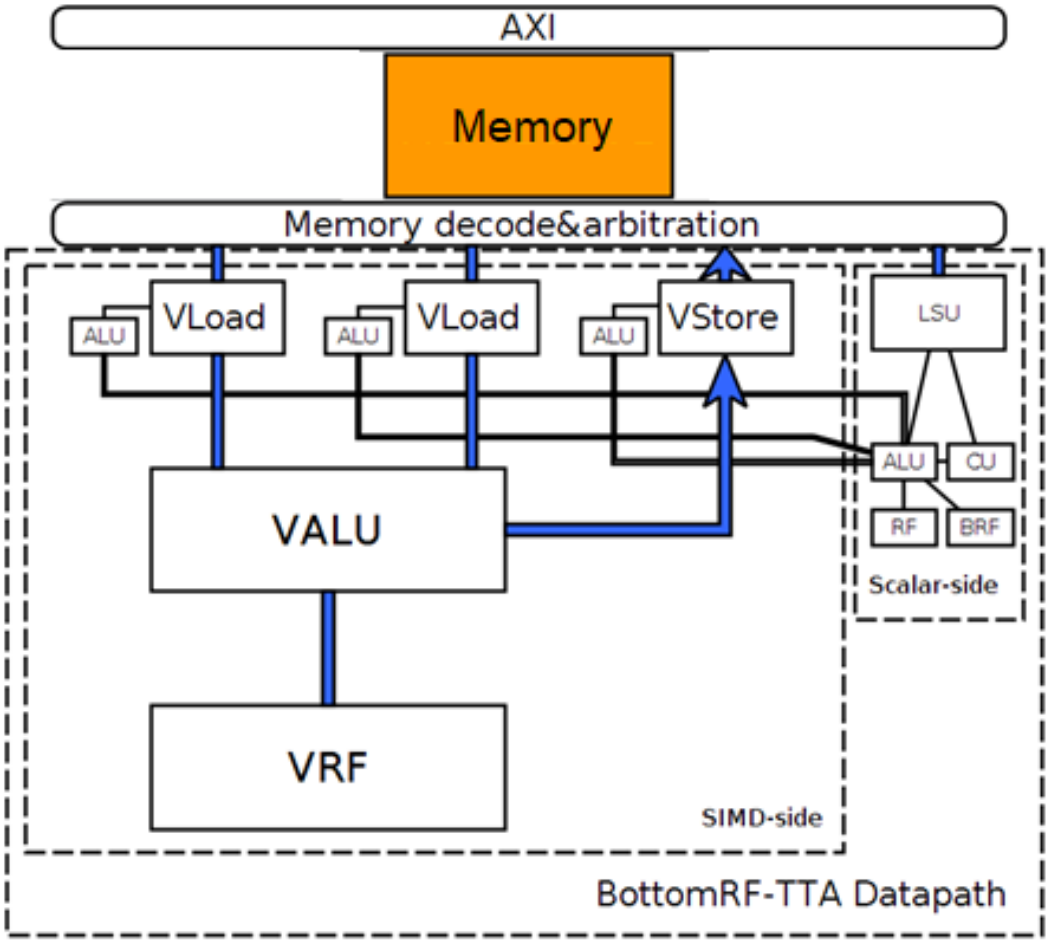

Figure 4.3. *Top-level organization of the reduced connectivity architecture template*

The high-level design for the template is as such, the LSUs are directly connected with the memory on top of the design. For the vector side of the template, the loads and store units are in separate FUs. Each LSU is connected to a dedicated ALU for generating memory addresses. These are known as address generation units (AGU). The LSUs are also connected to one or more VALUs, depending on the requirement of the program.

These are then connected to the RFs. Thus, there will be no direct connections between any of the LSUs and the RFs. The only connection would be through the ALUs.

The scalar part of the template has simpler connections compared to the vector section. The main purpose of this section is to implement branching, base address computation, index computation, etc. The scalar LSU is connected to the RFs through the scalar ALU. This is also connected to the AGUs for providing the base address and other complex arithmetic.

Figure 4.3 shows the top-level organization of the reduced connectivity architecture. In this figure, bi-directional connectivity is represented by an edge without arrows. Arrows denote a unidirectional connection. It is clearly visible that the RFs are connected only to the ALUs. Not connecting to the load units directly saves connectivity. The ALUs next to LSUs have only bypass connections to the load units. The scalar side is also connected to scalar ALU through bypass connection. The bypass network is customized to contain only the reasonable bypasses such as (V)ALU to LSU, ALU to LSU for address computation, LSU to ALU.

At the top level of the loop kernel, architectures based on the reduced connectivity template should be able to load the next vector operands, add the current vector, and store the previous one - all in parallel. In addition, it has to compute memory addresses for the two inputs and output in parallel in order to point towards the next inputs and output vectors in the memory. It also has to perform bounds checking and branching. All these are to be done in one single cycle.

4.3 Comparison between the Architecture Templates

The two architecture templates explained are both applicable for parallel, pipelined, and SIMD computation. However, one of the major differences between them is in the IC network. In the case of the conference paper template, the data can be stored and transferred through the RF units. That means, the LSUs had direct connections with the RFs. But in the template with reduced connectivity, there are no direct connections between the LSUs and the RFs. These are connected through ALUs. RFs are used only when absolutely necessary. Data can be transferred from the memory to LSUs or between the VFU and the VRF. This increases the number of FUs in the template. But considering the benefits it brings to the design, this is a small trade-off.

The paper architectures have local memory - both for storing data i.e the data address space and for the stack and OpenCL kernel arguments (the param AS). This means, all the data is close by to use by the processor - which is important especially with the stack, but without access to off-chip memory through the AXI master. It requires the OpenCL runtime or, whatever is running on the host processor, for transferring data between the

off-chip memory and local memory. The reduced connectivity template only has memory access through AXI; which means every access has more latency but it can handle a larger memory. It is a pretty standard trade-off between small and fast memories and large and slow memories; although usually it's not controlled directly by programmer-visible means in a dynamic cache.

Furthermore, the architectures based on paper template perform only one SIMD operation per cycle, theoretically. But in reality, it needs multiple cycles to start a vec-add in a loop due to the address computation in the loop, loop overhead instructions, etc. On the contrary, this thesis architecture template increases the efficiency, apart from the vector operation itself, by making it possible to perform four operations (one address calculation, two loads, and one store) in each cycle. This is further explained in chapter 5.

5 EVALUATION

This chapter consists of a performance evaluation of a set of developed TTA architectures based on a reduced datapath connectivity template. The reduced connectivity architecture template, designed in this thesis, has been crafted by the integration of components and programmed using software pipelining. The architecture template is designed and the assembly code is written as such, it ensures that the template is able to perform one vec-add operation in each cycle, which will be explained in this chapter. To measure the performance, the architectures were synthesized and implemented on the FPGA ZYNQ Z7020 (speedgrade-1).

The performance of the design is validated in terms of the critical path, maximum frequency, and resource utilization analysis. A hand-crafted program for the vec-add operation has been tested for different machines with a similar structure for different vector sizes. Then, their results are compared with each other - firstly, among the same sized machines with vectors of different arithmetic precision; and secondly, among different-sized machines with vectors of the same precision (32b). Next, the results are evaluated and compared to the normalized outcomes of the architecture represented in the conference paper TTA-SIMD Soft Core Processors [25]. For the rest of this chapter, the conference paper template is referred to as Wide SIMD-TTA Soft Core architecture or paper machines or the paper template.

5.1 Description of Architectures Based on Reduced Connectivity Template

For testing and verification of different features of the proposed reduced connectivity TTA architecture template, 5 different architectures were designed for 4 different vector widths (128b, 256b, 512b, and 1024b). Due to the aggressively reduced connectivity, `tcecc` command is currently unable to compile any high-level code for the proposed template. As a result, a handcrafted assembly code is written for each of these architectures and then compiled to verify them in the FPGA. This connectivity strategy is under research and based on the results from this thesis, the research will be taken further that includes improving the compiler.

5.1.1 The Architecture

in figure 5.1, an architecture based on the reduced connectivity template is shown. This architecture is for a 32x4 (128b) machine.

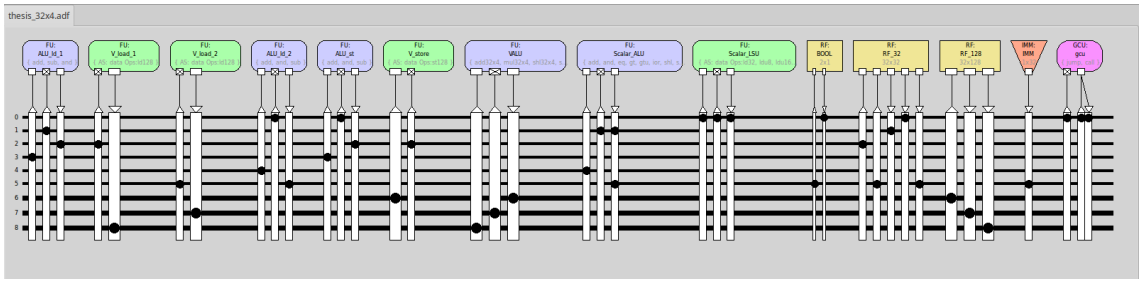

Figure 5.1. *32x4 architecture based on reduced connectivity template*

There are nine FUs. V_load _1, V_load _2, and V_store are for loading the two vectors (A and B) and storing the result (C) respectively. ALU_ld_1, ALU_ld_2, and ALU_st calculate the address space for loading the vectors and moving to the next address. Scalar_ALU and Scalar_LSU are for starting address space calculation. They also assist in loop iteration. There are three RFs - for BOOL, for address space (RF_32), and for vectors (RF_128); one long immediate unit (IMM) and one global control unit (GCU).

For the load-store units, both scalar and vector, the latency is 3. But in the case of all the ALUs, the latency is 2 for addition operation.

5.1.2 The Assembly Code

In figure 5.2, the hand-crafted assembly code for 32x4 (128b) architecture is shown. In the first 4 cycles, the RF_32 is loaded with the initial starting addresses for address space calculation for vector load-store operations and loop iteration. The code is written as such, it repeats after every 4 vector additions are performed and the result is stored; and it can execute a total of 12 iterations.

		0: B1	1: B2	2: B3	3: B4	4: B5	5: B6	6: B7	7: B8	8: B9
0	next> kernel	...	...	...	...	...	0 -> RF_32.0	...	...	...
1		...	...	...	...	...	IMM.0 -> RF_32.4	...	...	...
2		...	...	...	...	...	IMM.0 -> RF_32.3	...	...	...
3		...	...	...	...	...	IMM.0 -> RF_32.5	...	...	...
4		RF_32.4 -> ALU_ld_2.int.add	RF_32.3 -> ALU_ld_1.int.add	...	16 -> ALU_ld_1.in	16 -> ALU_ld_2.in	RF_32.0 -> RF_32.1	...	...	...
5		RF_32.4 -> ALU_ld_2.int.add	RF_32.3 -> ALU_ld_1.int.add	...	32 -> ALU_ld_1.in	32 -> ALU_ld_2.in	...	...	...	...
6		RF_32.4 -> ALU_ld_2.int.add	RF_32.3 -> ALU_ld_1.int.add	ALU_ld_1.out -> V_load_1.int.ld128	48 -> ALU_ld_1.in	48 -> ALU_ld_2.in	ALU_ld_2.out -> V_load_2.int.ld128	...	...	...
7		RF_32.4 -> ALU_ld_2.int.add	RF_32.3 -> ALU_ld_1.int.add	ALU_ld_1.out -> V_load_1.int.ld128	64 -> ALU_ld_1.in	64 -> ALU_ld_2.in	ALU_ld_2.out -> V_load_2.int.ld128	...	...	...
8		...	RF_32.1 -> Scalar_ALU.int.add	ALU_ld_1.out -> V_load_1.int.ld128	...	4 -> Scalar_ALU.in	ALU_ld_2.out -> V_load_2.int.ld128	...	...	...
9		RF_32.5 -> ALU_st.int.add	...	ALU_ld_1.out -> V_load_1.int.ld128	16 -> ALU_st.in	...	ALU_ld_2.out -> V_load_2.int.ld128	...	V_load_2.out -> VALU.int.add32x4	V_load_1.out -> VALU.in
10		RF_32.5 -> ALU_st.int.add	Scalar_ALU.out -> Scalar_ALU.int.eq	...	32 -> ALU_st.in	12 -> Scalar_ALU.in	Scalar_ALU.out -> RF_32.0	...	V_load_2.out -> VALU.int.add32x4	V_load_1.out -> VALU.in
11		RF_32.5 -> ALU_st.int.add	...	ALU_st.out -> V_store.int.st128	48 -> ALU_st.in	...	...	VALU.out -> V_store.in	V_load_2.out -> VALU.int.add32x4	V_load_1.out -> VALU.in
12		RF_32.5 -> ALU_st.int.add	...	ALU_st.out -> V_store.int.st128	64 -> ALU_st.in	...	Scalar_ALU.out -> BOOL.0	VALU.out -> V_store.in	V_load_2.out -> VALU.int.add32x4	V_load_1.out -> VALU.in
13		...	...	ALU_st.out -> V_store.int.st128	...	...	...	VALU.out -> V_store.in	...	...
14		!BOOL.0 4 -> gcu.pc.jump	...	ALU_st.out -> V_store.int.st128	...	...	...	VALU.out -> V_store.in	...	...
15		...	...	ALU_ld_1.out -> RF_32.3	...	...	ALU_ld_2.out -> RF_32.4	...	...	...
16		...	...	ALU_st.out -> RF_32.5	...	...	...	...	...	...
17		...	...	...	...	...	...	...	...	...

Figure 5.2. *Hand-crafted assembly code for the 32x4 architecture*

From cycle 5 to 8, address space calculation proceeds defining 8 addresses for loading

4 of each input - A and B vectors. Based on the latency, the input vector loading at those defined addresses starts from cycle 7 and continues until cycle 10. The address space calculation for the vector store unit starts from cycle 10. The input loads in the VALU also starts at the same cycle. Storing the result vector (C) starts from cycle 12 and ends at cycle 15. Finally, the ending addresses replace the initial addresses stored in the RF_32.

The loop calculation starts from cycle 11. The Scalar_ALU checks whether 12 additions have taken place, or not. If not, the loop restarts from cycle 5. When all the 12 additions are done and the result is stored in 12 different address spaces, the loop is discontinued and the program comes to an end.

5.2 Results

Similar assembly codes are tested on similar architecture templates and the output of different machines is obtained. These results are compared with each other and also with those of the outcomes from the Wide SIMD-TTA Soft Core architectures for the same vector width. In this section, the results are evaluated based on the critical path, maximum frequency, and FPGA utilization.

5.2.1 Critical Path

Critical path and timing summary are important factors to understand the speed of the architectures. These factors define the highest possible speed for the designs. Slack helps in static timing analysis by indicating whether timing has been met across a timing path.

Figure 5.3 shows timing summary for 16x8, 32x4, 32x8, 32x16 and 32x32 machine architectures. The machines are tested at the highest possible frequency at which they can run successfully with positive slacks. All the machines have positive slacks after implementation on FPGA, which means the designs pass the timing constraints.

Depending on the vector widths, the architectures based on the thesis template have various ranges of critical paths. Table 5.1 shows the critical path data for the thesis architectures.

The highest critical path delay is seen for 1024b machine architecture, which is 9.094ns. For this machine, the critical path started from the on-chip data memory and ended at the FU for scalar LSU. For other thesis architectures, however, the route for this path is different. The starting point of the critical path is RF_32, which is the register for the scalar part; and the ending point is in the out-port of the ALU for the second vector load FU.

Since the critical path starts from scalar register files, the delay is not affected by the vector part of the architecture template. This means the SIMD part of the design does not

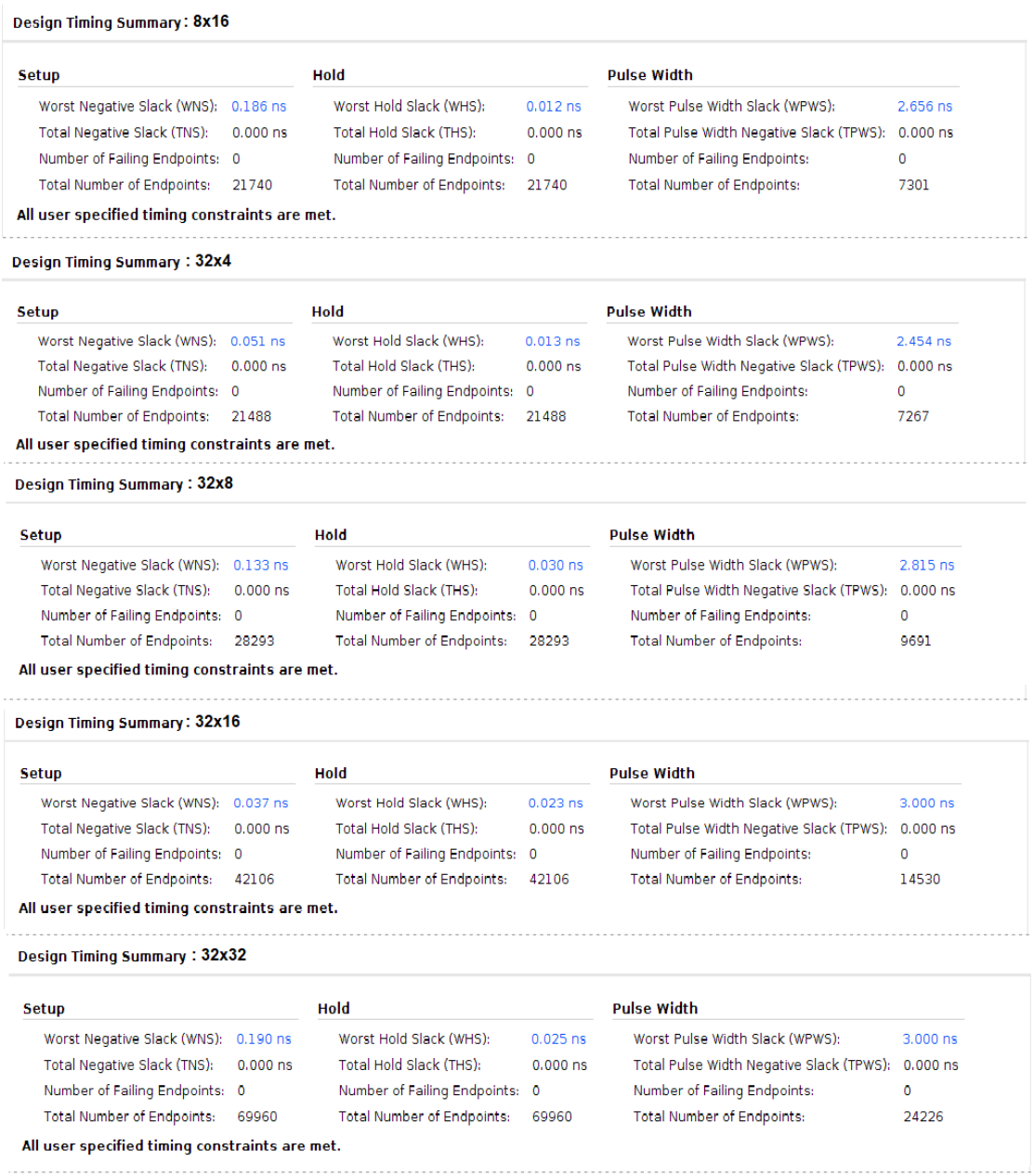

Figure 5.3. *Timing summary for different reduced connectivity architectures*

create any bottleneck in the workflow while the processor is running. This is one of the positive aspects of the design.

The critical paths for thesis architectures also indicate that the scalar part of the design is too complicated. This creates a long chain of propagation delay. This could be one cause for the current critical path delay. Furthermore, apart from being used during the initialization of the starting address for vector loads and stores, the scalar part is also used for loop count. As the same resource is utilized for executing different operations, the execution time is much slower. A solution to this could be simpler and less complicated interconnections for the scalar part, to help in better loop enrolling and other functions.

Table 5.1. *Critical Path of reduced connectivity architectures*

Precision	Architecture	Critical Path		
		Delay (ns)	From	To
8b	128b	7.473	/rf_RF_32/lvt_r_reg[18]/C	/fu_alu_st_generated/ data_out1_2_r_reg[29]/D
32b	128b	7.155	/rf_RF_32/lvt_r_reg[17]/C	/fu_alu_ld_2_generated/ data_out_2_r_reg[25]/D
32b	256b	7.729	/rf_RF_32/lvt_r_reg[10]/C	/fu_alu_ld_2_generated/ data_out_2_r_reg[31]/D
32b	512b	8.640	rf_RF_32_R2_opc_ reg_reg[1]/C	/fu_alu_ld_2_generated/ data_out_2_r_reg[31]/D
32b	1024b	9.094	/onchip_mem_data/RAM_ APR_reg_38/CLKARDCLK	/fu_scalar_lsu_generated/ data_out_1_r_reg[5]/D

This might help reduce the critical path delay even further.

5.2.2 Maximum Frequency

In table 5.2, fmax found for different machines for a hand-crafted vec-add is shown. It also shows the maximum clock frequency for the machines with a similar vector width, used in the paper.

Table 5.2. *Maximum frequency and operations per second (OPS) for different architectures*

Precision	Architecture	Thesis Template		Paper Template	
		fmax (MHZ)	OPS	fmax (MHZ)	OPS
8b	128b	128	128	155	51
32b	128b	135	135	153	51
32b	256b	123	123	145	48
32b	512b	115	115	115	38
32b	1024b	103	103	108	36

For smaller machines, fmax from the older design is much higher than the ones received from the modified design. For 512b, both the architectures resulted in the same maximum frequency, which is 115 MHz. For 1024b, fmax for older architecture is 5 MHz higher than that of the new design. But, this is a very unexpected result. Because connections are removed in the thesis template, which should mean the complexity reduces and thus the clock frequency increases. The reason behind this is, the machines based on the Wide SIMD-TTA Soft Core template is not exactly the same as that of the thesis machines. The

machines based on the Wide SIMD-TTA Soft Core template are much larger in functions; those are not just limited to vec-adds. Based on different functions and operations, the Wide SIMD-TTA Soft Core template implements different latencies and pipelining.

Because of the aforementioned reasons, a comparison between these two architecture templates based on the fmax only does not add much to the evaluation process. To make this more meaningful, the table also includes peak operations per second. Hence, we need to normalize the data received from the architectures based on Wide SIMD-TTA Soft Core. For this, operations per second (OPS) is used.

The architectures and peak operations per cycle are different for Wide SIMD-TTA Soft Core and thesis templates. As explained in assembly code, architectures based on the thesis template can perform 1 operation per cycle. On the other hand, the architectures based on Wide SIMD-TTA Soft Core perform 1 SIMD operation per 3 cycles i. e. one-third operations per cycle. We can obtain the OPS for thesis architectures by:

$$\text{OPS} = \text{Maximum frequency X operations per cycle}$$

In figure 5.4, we can see a comparison between the OPS for architectures based on the thesis and Wide SIMD-TTA Soft Core templates. The architectures have the same vector width, which is represented along the X-axis. Along the Y-axis, the OPS is portrayed. The blue line indicates the OPS for architectures based on reduced connectivity template and the orange line shows the OPS for the Wide SIMD-TTA Soft Core template-based architectures.

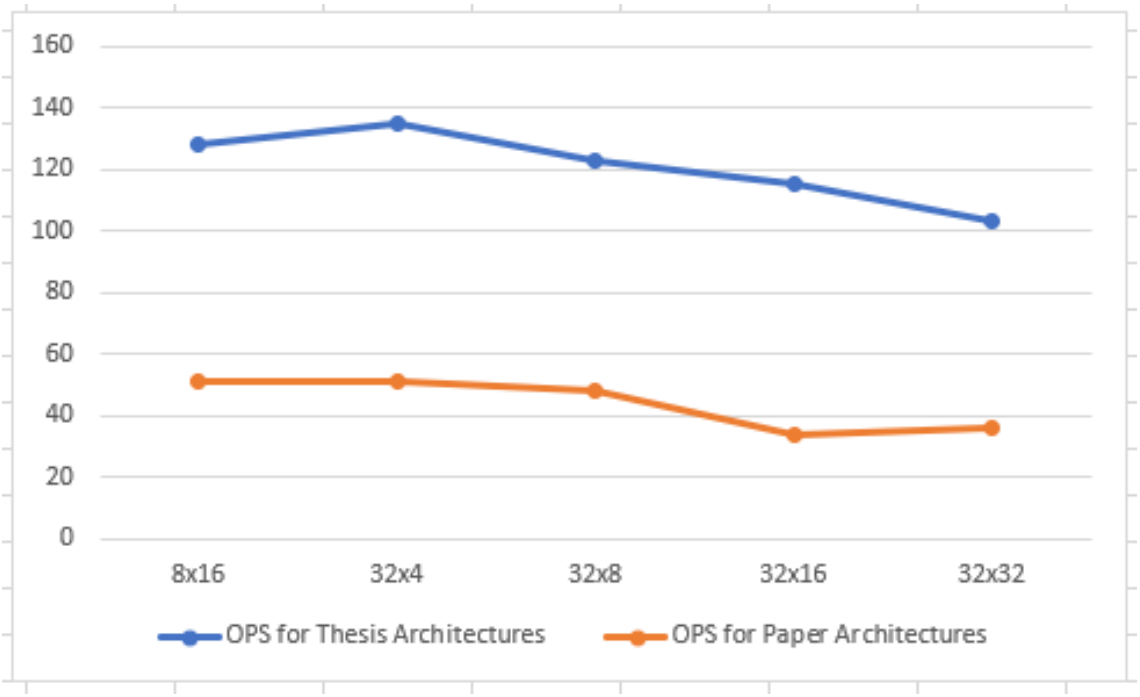

Figure 5.4. *Chart showing a comparison between OPS for architectures based on Thesis and Wide SIMD-TTA Soft Core templates*

In the chart, we see that for smaller machines, OPS for architectures based on Wide SIMD-TTA Soft Core are pretty much similar. However, it decreases for 512b; but again goes up for 1024b. But in the case of thesis architectures, for the same vector width, OPS is better for arithmetic precision 32 than 8. But OPS shows a decreasing trend as the element count increases, for a fixed arithmetic precision (32b, in this case). This could be due to the reason that, as the vector width increases, FPGA needs more resources to perform the same operation. The address spaces also get larger. More resources mean more timing constraints and as a result to meet those constraints a lesser number of operations take place per second. And for the same reason, the maximum frequency also declines.

5.2.3 FPGA Utilization

FPGA utilization numbers are related to its area efficiency or resource efficiency. Area or resource usage must be compared to the number of operations performed per second per occupied area. For this reason, utilization can be the metric of the peak number of OPS per consumed resource. In Table 5.3, FPGA utilization numbers in respect of LUTs, DSP blocks, and RAM blocks are shown for the thesis architectures. Here, percentages indicate how much of the resources available on the FPGA is consumed by the component.

Table 5.3. *FPGA utilization of reduced connectivity architectures*

Precision	Architecture	LUT	DSP blocks	BRAM blocks	DRAM blocks
8b	128b	6746 (12.68%)	19 (8.64%)	68 (48.57%)	432
32b	128b	6639 (12.48%)	15 (6.82%)	68 (48.57%)	432
32b	256b	9208 (17.31%)	27 (12.27%)	68 (48.57%)	600
32b	512b	14051 (26.41%)	51 (23.18%)	68 (48.57%)	944
32b	1024b	24187 (45.46%)	99 (45.00%)	68 (48.57%)	1624

From the table, we see that for 128b vector width, the utilization numbers for 32b precision is better than 8b precision. This is because, as the precision increases, fewer resources are needed for performing the same operations. For the same vector width, we can expect better efficiency as the peak OPS is reduced when wider datapath is used. However, it is true for all SIMD template. For a fixed vector width and a narrow data type, more parallel operations can be executed.

On the other hand, for the same arithmetic precision, FPGA utilization rises with the increase of vector width. As the width increases, FPGA needs more resources for performing the same function. This causes an increase in the FPGA area and resource use. Thus, the use of LUT and DSP blocks gets higher.

The utilization number for the DSP blocks doubles with the doubling of vector width. But it doesn't translate to doubling of LUTs. A comparison of LUT utilization by different thesis machines can be seen in figure 5.5. From the bar graph, we can see, as the vector width doubles, the utilization of LUTs increases by almost 1.5 times. This points out with the increase of vector width, resource utilization is improved.

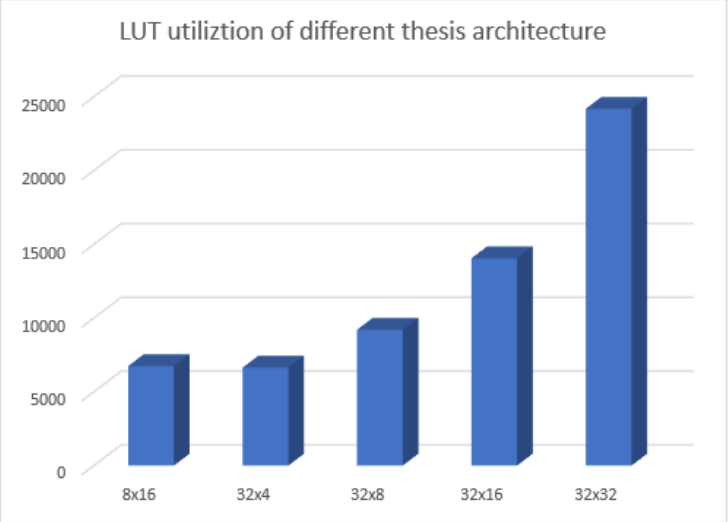

Figure 5.5. *LUT utilization by different thesis machines*

The RAM block utilization is usually always the same for similar architectures performing similar operations. In this case, also, it is no exception. This is because, BRAM utilization should be the same for architectures with the same address space sizes, which is 18 in the case of these machines. However, the vector RFs were implemented by distributed RAMs (DRAM) and they differ with the vector width. For the same vector width, DRAM remains the same for both 8b and 32 b precision. On the other hand, for the same precision, as the vector width increases, DRAM also increases. This is because, with the increase of vector width, a more distributed memory operation is needed by the VRF.

While comparing with the architectures based on Wide SIMD-TTA Soft Core, to make these numbers comparable, we need a meaningful performance metric. Hence, OPS per resource utilization seemed to be a reasonable metric given the time limit for this thesis. The results are in Table 5.4.

When compared, LUT utilization per OPS between thesis machines and paper (Wide SIMD-TTA Soft Core) machines show quite a big difference. Higher LUT/OPS means that more LUTs are needed to get 1 OPS. For 8x16 machines, the LUT requirement for Wide SIMD-TTA Soft Core architecture is almost 9 times that of the thesis architecture. For the 32x32 machine, it is almost 12 times. This indicates, to perform one operation per second i.e. 1 OPS, the Wide SIMD-TTA Soft Core template needs a lot more LUTs than the thesis architecture template. Due to the reduced connectivity, resource utilization by the thesis architectures has been improved significantly.

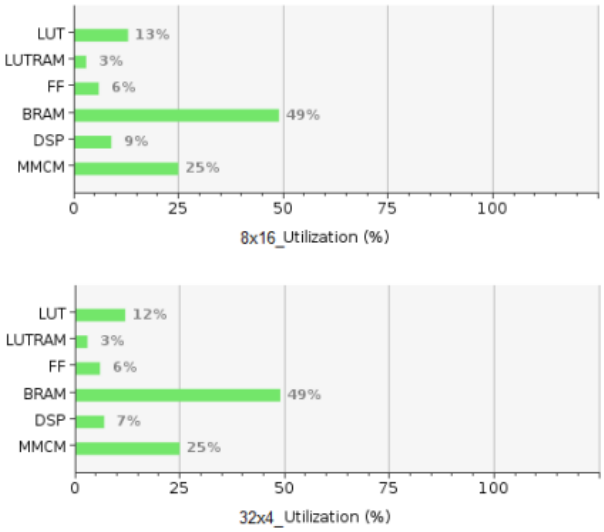

Figure 5.6. *Graphical view of utilization for two 128b vector width with different precision*

Table 5.4. *LUT utilization/OPS of reduced connectivity architectures and architectures based on Wide SIMD-TTA Soft Core.*

Precision	Architecture	LUT/OPS	
		Thesis Machines	Paper Machines
8b	128b	52	513
32b	128b	49	520
32b	256b	74	761
32b	512b	122	1539
32b	1024b	234	2908

The Wide SIMD-TTA Soft Core template has many more functionalities and hence, the thesis template is not exactly comparable with it directly. However, when these two are compared using an equivalent metric for comparison, the thesis template is proven to be a better solution in terms of frequency and utilization. The most powerful aspect of this architecture template is that it can sustain one operation per cycle due to better loop pipelining. This almost doubles the efficiency compared to the Wide SIMD-TTA Soft Core template performing a maximum of one operation per two cycles. Hence, it is definitely an optimized improvement.

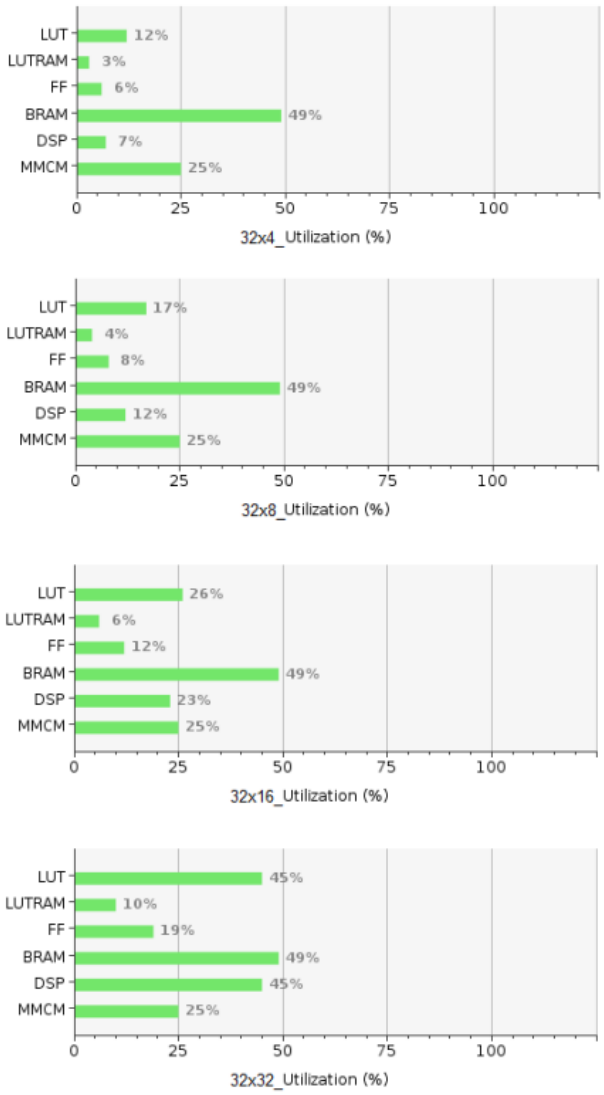

Figure 5.7. *Graphical view of utilization for different vector width with the same precision*

6 CONCLUSION

In this thesis, a reduced network of interconnects for the TTA template, especially for FPGA implementation, has been studied. It portrays an exposed SIMD datapath connectivity model with a reduced number of connections that can perform one vec-add operation in each cycle. It is a TTA variation with reduced internal connectivity beyond standard VLIW processors to improve FPGA implementation.

Different architectures have been designed based on this TTA template and explored. Each of the architectures has been programmed using hand-written assembly code and then implemented on FPGA ZYNQ Z7020 (speedgrade-1) and then evaluated based on different aspects.

A hand-crafted program for the vec-add operation has been tested for five machines with different vector sizes. Then, their results are compared with each other - firstly, among the same sized machines with vectors of different arithmetic precision; and secondly, among different-sized machines with vectors of the same precision (32b). Then, the results are evaluated and compared to the normalized outcomes of the architectures based on a Wide SIMD-TTA Soft Core.

The design is validated based on the critical path, maximum clock frequency, and resource utilization analysis. In the thesis machines, the critical path starts from scalar register files. It indicates that the delay is not affected by the vector part of the architecture template. Hence, it is safe to say that the SIMD part of the design does not create any bottleneck in the workflow while the processor is running. This is one of the positive aspects of the design.

Since connections are removed in the thesis template, the complexity reduces, and thus the maximum clock frequency increases. This is further seen in terms of OPS. In thesis architectures, for the same vector width, operations per second (OPS) is better for arithmetic precision 32 than 8. But OPS shows a decreasing trend as the element count increases, for a fixed arithmetic precision (32b, in this case). This is generally true for all SIMD processors. But when compared to architectures based on Wide SIMD-TTA Soft Core, the OPS is almost 3 times higher for the machines with the same vector width. This is a significant improvement.

When compared based on FPGA resource utilization, results show the LUT utilization

per OPS between thesis machines and machines based on Wide SIMD-TTA Soft Core has quite a big difference. Architectures based on Wide SIMD-TTA Soft Core needs a lot more LUTs than the thesis machines with the same vector width and arithmetic precision. Due to the reduced connectivity, resource utilization by the thesis architectures has been improved significantly.

After the evaluation of the reduced connectivity template, it clearly displays how it can upgrade the FPGA implementation. However, there are some limitations and there is much scope for further studies. The critical paths for thesis architectures indicate that the scalar part of the design is too complicated. A solution to this problem could be simpler and less complicated interconnections for the scalar part. This could be studied further to verify if this critical path could further be improved.

There has not been any thorough study of the power consumption for this thesis template. This is a very important aspect to decide on the practical and viable implementation of any design. For this reason, it should be explored in future studies before coming to any conclusion regarding this reduced connectivity template.

Furthermore, this template can only perform vec-add right now. It should be tested for other vector operations as well. There is no compiler currently that can directly compile a high-level code into a machine language compatible with implementing on this template. This could be an interesting research topic as well.

REFERENCES

[1] Trimberger, S. M. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. Vol. 103. 3. Mar. 2015, 318–331.

[2] *Occupational outlook handbook. Bureau of Labor Statistics*. URL: https://www.bls.gov/ooh/home.htm (visited on 10/13/2020).

[3] Jääskeläinen, P., Tervo, A., Paya-Vaya, G., Viitanen, T., Behmann, N., Takala, J. and Blume, H. Transport-Triggered Soft Cores. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018.

[4] Harris, D. M. and Harris, S. L. *Digital Systems: From Logic Gates to Processor*. 1st edition. Switzerland: Springe, 2017.

[5] Walton, J. *Nvidia Unveils Its Next-Generation 7nm Ampere A100 GPU for Data Centers, and It's Absolutely Massive*. May 14, 2020. URL: https://www.tomshardware.com/news/nvidia-ampere-A100-gpu-7nm (visited on 10/13/2020).

[6] Deschamps, J.-P., Elena, V. and Lluis, T. *Digital Design and Computer Architecture*. 2nd edition. Amsterdam: Elsevier/Morgan Kaufmann, 2012.

[7] Wolf, M. *High-Performance Embedded Computing. Architectures, Applications, and Methodologies*. 2nd edition. Morgan Kaufmann: Elsevier Inc., 2014.

[8] Blanchet, G. and Dupouy, B. *Computer Architecture*. London: ISTE, 2013.

[9] *Organization of Computer Systems: Processors. Computer & Information Science & Engineering*. URL: https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html (visited on 10/17/2020).

[10] Saidi, A., Binkert, N., Reinhardt, S. and Mudge, T. Full-System Critical Path Analysis. Apr. 2008, 63–74. DOI: 10.1109/ISPASS.2008.4510739.

[11] Yiannacouras, P., Steffan, J. and Rose, J. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. *Proceedings of the 2008 international conference on compilers, architectures and synthesis for embedded systems*. ACM. 2008, 61–70.

[12] Tervo, A. Optimizing Transport-Triggered Architectures for Field-Programmable Gate Arrays. MA thesis. Finland: Tampere University of Technology, 2018.

[13] Corporaal, H. *Transport Triggered Architectures: Design and Evaluation*. Technische Universiteit Delft, 1995.

[14] *OpenCL 1.2 API Specification*. Revision 19. Khronos OpenCL Working Group.

[15] J.L. Hennessy, D. P. *Computer Architecture: A Quantitative Approach*. 5th edition. Morgan Kaufmann. Waltham. MA, USA, 2012.

[16] Waeijen, L., She, D., Corporaal, H. and He, Y. A Low-Energy Wide SIMD Architecture with Explicit Datapath. Vol. 80. 1. 2015, 65–86.

[17] Thuresson, M., Själander, M., Björk, M., Svensson, L., Larsson-Edefors, P. and Stenstrom, P. FlexCore: Utilizing Exposed Datapath Control for Efficient Computing. Vol. 57. 1. 2009, 5–19.

[18] Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., Moore, C., Burrill, J., McDonald, R. and Yoder, W. Scaling to the end of silicon with EDGE architectures. *Computer* 37.7 (July 2004), 44–55.

[19] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V. and Amarasinghe, S. Space-time scheduling of instruction-level parallelism on a raw machine. *Operating systems review* 32.5 (Dec. 1998), 46–57.

[20] Tabak Daniel; Lipovsky, G. J. MOVE Architecture in Digital Controllers. *IEEE Transactions on Computers* C-29.2 (1980), 180–190.

[21] Hirvonen, A., Tervo, K., Kultala, H. and Jääskeläinen, P. AEx: Automated Customization of Exposed Datapath Soft-Cores. Aug. 2019. DOI: 10.1109/DSD.2019.00016.

[22] Pavan, P. S. *ASIC vs FPGA*. Feb. 26, 2020. URL: http://www.signoffsemi.com/asic-vs-fpga/#:~:text=ASIC%20means%20Application%20Specific%20Integrated,a%20specific%20purpose%20or%20functionality.&text=The%20difference%20in%20case%20of,a%20number%20of%20configurable%20blocks. (visited on 09/24/2020).

[23] Jääskeläinen, P., Viitanen, T., Takala, J. and Berg, H. HW/SW Co-design Toolset for Customization of Exposed Datapath Processors. Springer, Dec. 2016.

[24] Lattner, C. Llvm and clang: Next generation compiler technology. The BSD conference, 2008.

[25] Tervo, K., Samawat, M., Leppänen, T. and Jääskeläinen, P. TTA-SIMD Soft Core Processors. *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications*. accepted for publication. IEEE. 2020.

[26] *Amazon EC2 F1 Instances. Amazon Web Services*. URL: https://aws.amazon.com/ec2/instance-types/f1/ (visited on 10/21/2020).

[27] *FPGA Accelerated Cloud Server. Huawei Technologies Co. Ltd.* URL: https://www.huaweicloud.com/en-us/product/fcs.html (visited on 10/21/2020).

[28] Kharchenko V. Kondratenko Y., K. J. *Green IT Engineering: Components, Networks and Systems Implementation*. Springer, Cham, 2017.

[29] *Xilinx. Field Programmable Gate Array (FPGA)*. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html#:~:text=Field%20Programmable%20Gate%20Arrays%20(FPGAs,or%20functionality%20requirements%20after%20manufacturing. (visited on 09/21/2020).

[30] Churiwala, S. *Designing with Xilinx FPGAs Using Vivado*. 1st edition. Cham: Springer International Publishing, 2017.

[31] Kuon, I., Russell, T. and Jonathan, R. *FPGA Architecture : Survey and Challenges*. Hanover, MA: Now Publishers, 2008.

[32] Maxfield, C. *FPGAs*. Amsterdam;: Newnes/Elsevier, 2009.

[33] *Spartan-6 FPGA DSP48A1 Slice User Guide*. Version UG389 (v1.2). 2014. URL: https://www.xilinx.com/support/documentation/user_guides/ug389.pdf (visited on 09/21/2020).

[34] Vipin Kizheppatt ; Fahmy, S. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Computing Surveys (CSUR)* 51.4 (2018), 1–39.

[35] Mingjie Lin, J. W. and Gamal, A. E. Exploring FPGA Routing Architecture Stochastically. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.10 (2010), 1509–1522.

[36] *Intel® Stratix® 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. Intel. 2018, 5. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf.

[37] *A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements. White Paper*. Intel. 2018, 6. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf?wapkw=A%20new%20FPGA%20architectur%20Process%20technologye%20and%20leading-edge%20FInFET (visited on 10/21/2020).

[38] Betz Vaughn, J. R. and Marquardt, A. *Architecture and CAD for Deep-Submicron FPGAS*. Vol. 497. Boston, MA: Springer US, 1999.

[39] Guy, L. and Lewis., D. *Design of Interconnection Networks for Programmable Logic*. Boston, MA: Springer US, 2004.

[40] Esko, O., Jääskeläinen, P., Huerta, P., La Lama, C. de, J.Takala and Martinez, J. Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. *Int. Conf. on Field Programmable Logic and Applications (FPL)* (2010). ISSN: 1946-1488. DOI: http://doi.ieeecomputersociety.org/10.1109/FPL.2010.51.

[41] Järvelä, M. Vector Operation Support for Transport Triggered Architectures. MA thesis. Finland: Tampere University of Technology, 2014.