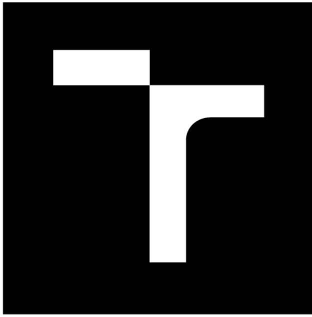
BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2017

Bc. Jakub Žádník



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS

ÚSTAV RADIOELEKTRONIKY

IMPLEMENTATION OF FAST FOURIER TRANSFORMATION ON TRANSPORT TRIGGERED ARCHITECTURE

IMPLEMENTATION OF FAST FOURIER TRANSFORMATION ON TRANSPORT TRIGGERED ARCHITECTURE

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR Bc. Jakub Žádník

AUTOR PRÁCE

SUPERVISOR prof. Ing. Roman Maršálek, Ph.D.

VEDOUCÍ PRÁCE

BRNO 2017

Diplomová práce

magisterský navazující studijní obor **Elektronika a sdělovací technika**
Ústav radioelektroniky

Student: Bc. Jakub Žádník ***ID:*** 154916
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Implementation of Fast Fourier Transformation on Transport Triggered Architecture

POKYNY PRO VYPRACOVÁNÍ:

Study FFT algorithm and identify properties, which can be exploited in hardware implementations. Specify special function units, which can be used to customize a programmable processor based on transport triggered architecture. Develop instruction schedule for mixed-radix FFT computation, which effectively exploits the custom features of the processor.

Develop RT_ description of the suggested processor architecture and verify its operation. To minimize the power consumption due to instruction memory access, implement an instruction loop buffer and modify the processor such that the kernel for FFT computations can be realized with a single instruction with hardware looping capabilities. Use the customized processor in an audio application as an example.

DOPORUČENÁ LITERATURA:

[1] PITKANEN, T. Fast Fourier Transforms on Energy-Efficient Application-Specific Processors. Tampere, Finland, 2014. Doctoral dissertation.

[2] CORPORAAL, H. Microprocessor architectures: from VLIW to TTA. New York: J. Wiley, 1998. ISBN 04-71-
-7157-X.

Termín zadání: 6.2.2017 ***Termín odevzdání:*** 15.8.2017

Vedoucí práce: prof. Ing. Roman Maršálek, Ph.D.
Konzultant: prof. Dr.Tech. Jarmo Takala

prof. Ing. Tomáš Kratochvíl, Ph.D.
předseda oborové rady

UPOZORNĚNÍ:
Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRACT

The thesis proposes an energy-efficient processor architecture for computing a Fast Fourier Transform (FFT) using a Transport Triggered Architecture (TTA) template. The architecture was specifically tailored to a custom instruction schedule using several custom functional units (FU). The instruction schedule for computing the algorithm was developed in a way that most of the computation is done in a loop containing only one instruction word. This word is stored into an instruction loop buffer which is more power-efficient than a regular memory storage. Thus a power consumption is reduced. Python programs for reference generation and automatic verification of the timed model were developed to aid the design process. A timed model of the processor and the instruction schedule were developed, the approach was verified, and further improvements are suggested.

KEYWORDS

Fast Fourier Transform, Transport-Triggered Architecture, TTA-Based Co-Design Environment, Application-Specific Processor, Python, C, VHDL

ABSTRAKT

V této práci je navrhnut energeticky úsporný procesor typu TTA (Transport Triggered Architecture) pro výpočet rychlé Fourierovy transformace (FFT). Návrh procesoru byl vytvořen na míru použitému algoritmu pomocí speciálních funkčních jednotek. Algoritmus byl realizován jako posloupnost instrukcí tak, že většina výpočtu probíhá ve smyčce obsahující pouze jedinou paralelní instrukci. Tato instrukce je umístěna do instrukčního bufferu, odkud je potom volána místo instrukční paměti. Díky tomu je docíleno nižší spotřeby, neboť čtení z instrukčního bufferu je efektivnější než čtení z běžné paměti. Součástí práce jsou rovněž pomocné programy v Pythonu, které slouží ke generaci referenčních výsledků a automatické simulaci a porovnání výsledků simulace s referencí. Program byl zkompilován na časovém modelu procesoru a časová simulace potvrdila správnost návrhu.

KLÍČOVÁ SLOVA

rychlá Fourierova transformace, TTA, TCE, aplikačně-specifický procesor, Python, C, VHDL

ŽÁDNÍK, Jakub. *Implementation of Fast Fourier Transformation on Transport Triggered Architecture*. Brno, 2017, 51 p. Master's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky. Advised by prof. Jarmo Takala

Vysázeno pomocí balíčku thesis verze 2.63; http://latex.feec.vutbr.cz

DECLARATION

I declare that I have written the Master's Thesis titled "Implementation of Fast Fourier Transformation on Transport Triggered Architecture" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

On the Finnish side, I would like to thank my supervisor, Jarmo Takala, for providing a guidance, consultations and necessary materials to complete the thesis. My thanks also go to Muazam Ali, Jingui Li and Fahad - my friends, co-workers and lunch-mates - for helpful discussions, friendship and company. Without the CPC research team and their TCE tools it would have never been possible for me to finish the thesis and without the previous work of Teemu Pitkänen it would have not been possible to even begin. Thanks Lily for her love and gorgeous meals during the pressing times. And without naming, thanks to all the friends I made during my Finnish year for contributing to this amazing experience!
On the Czech side, I would like to thank Roman Maršálek for a thesis supervision and support. My thanks also go to Tomáš Frýza and other members of the Department of Radioelectronics for their patience and a supportive attitude during my foreign stay. On this note, I would also like to thank Hana Šindelářová for her immense help dealing with all the administrative things. My thanks go to all my classmates and teachers I've been meeting four years at school and who provided friendships and an incredibly peaceful atmosphere to study and live in. Especially thanks to my friend Ondřej Sládek who made sure this thesis gets printed as a physical object and ends up in the right hands while I am still in Finland. Also cheers to my family and dog Ríša for their patience and presence, even when 2000 km away.

Brno . . . .

author's signature

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

ACKNOWLEDGEMENT

Research described in this Master's Thesis has been implemented in the laboratories supported by the SIX project; reg. no. CZ.1.05/2.1.00/03.0072, operational program Výzkum a vývoj pro inovace.

Brno . . . .

author's signature

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

CONTENTS

List of symbols, physical constants and abbreviations **11**

Introduction **12**

1 Fourier Transforms **13**
1.1 Radix-p Algorithms 13
1.2 Radix-2 14
1.3 Radix-4 15
1.4 Mixed Radix 16
1.5 Twiddle Factors 17
1.6 Memory Optimization 17

2 Transport Triggered Architecture **19**
2.1 TTA Overview 19
2.2 TTA-Based Co-Design Environment 20

3 Untimed Reference Model **23**
3.1 Program Overview 23
3.2 Detailed Description 23

4 Timed Simulation Model **27**
4.1 Simulation Workflow 27
4.2 Proposed Architecture 28
4.3 Functional Units 30
4.3.1 Address Generator 30
4.3.2 Twiddle Factor Generator 31
4.3.3 Complex Adder 32
4.3.4 Complex Multiplier 33
4.3.5 Rotating Register 33
4.4 Instruction Schedule 34
4.5 Summary 37

5 Hardware Implementation Approach **40**
5.1 Design Workflow 40
5.2 Functional Units 41
5.2.1 Address Generator 42
5.2.2 Twiddle Factor Generator 42
5.2.3 Complex Adder 45

5.2.4 Complex Multiplier 45
5.2.5 Rotating Register 47
5.3 Memories 47

6 Conclusion **50**

Bibliography **51**

LIST OF FIGURES

1.1 Radix-2 butterfly . . . 14
1.2 Radix-4 butterfly can be computed combining several radix-2 butterflies 16
1.3 Twiddle factors of FFT size 64 . . . 18
2.1 Example of a TTA processor configuration . . . 20
3.1 Abbreviated Python script for computing FFT . . . 25
4.1 Simulation workflow in TCE; White ellipses are files; Dark boxes are TCE programs; White boxes are what the programs contain; Grey circle is a user's input . . . 29
4.2 Diagram of the proposed TTA architecture with an interconnection network . . . 30
4.3 Schedule of a complex adder . . . 32
4.4 Radix-4 butterfly instruction schedule and reservation table . . . 35
4.5 Instruction schedule and reservation table of FFT of size 16. . . . 38
5.1 Processor implementation workflow in TCE . . . 41
5.2 Block diagram of an address generator . . . 43
5.3 Block diagram of a twiddle factor generator . . . 44
5.4 Block diagram of a complex adder . . . 46
5.5 Block diagram of a complex multiplier . . . 47
5.6 Block diagram of a rotating register . . . 48

LIST OF TABLES

4.1 Address Generator - ports 31
4.2 Twiddle Factor Generator - ports 31
4.3 Complex adder - ports 33
4.4 Complex multiplier - ports 33
4.5 Rotating register - ports 34
4.6 Instruction notation format 36
4.7 Number of required clock cycles for all supported FFT sizes. 39

LIST OF SYMBOLS, PHYSICAL CONSTANTS AND ABBREVIATIONS

AG	Address Generator
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASP	Application Specific Processor
BRAM	Block RAM
CADD	Complex Adder
CLI	Command Line Interface
CMUL	Complex Multiplier
DFT	Discrete Fourier Transform
DIF	Decimation-in-frequency
DIT	Decimation-in-time
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
FU	Functional Unit
GCU	General Control Unit
GUI	Graphical User Interface
GPP	General Purpose Processor
HDL	Hardware Description Language
IC	Interconnection Network
IM	Instruction Memory
LSB	Least Significant Bit
LSU	Load-Store Unit
LUT	Lookup Table
MAU	Minimum Addressable Unit
MSB	Most Significant Bit
RF	Register File
ROM	Read Only Memory
RTL	Register-Transfer Level
SRAM	Static Random-Access Memory
SQNR	Signal-to-Quantization-Noise Ratio
TCE	TTA-Based Co-Design Environment
TTA	Transport-Triggered Architecture
TFG	Twiddle Factor Generator
TUT	Tampere University of Technology
VLIW	Very Long Instruction Word

INTRODUCTION

In the current days of a massive integration of digital circuits into smaller and smaller chips and growing demands on computation complexity, we face several important decisions when designing a DSP (Digital Signal Processing) application. One of them to choose the optimal platform.

On one end there are ASICs (Application Specific Integrated Circuits) which are custom made chips finely tuned for the application. This approach offers the best possible performance, energy-efficiency and area requirements. ASIC design requires a lot of time and production of these circuits is expensive when done in smaller series. Most often ASIC circuits are optimized for one single use and therefore are inflexible - meaning that for another application we need another ASIC.

On the other side stand programmable solutions such as GPP (General Purpose Processor). These offer very good flexibility and short design times. If our application is very specific or demanding, though, they might suffer from limited performance and too high power consumption.

A solution to the dilemma can be found in ASPs (ASPs). They are hardware-optimized towards one application (i.e. computing FFT - Fast Fourier Transform) or an application domain (a group of related applications). Their performance can be brought closer to an optimized ASIC. However, they are still programmable and thus have a certain level of flexibility. With the use of modern development tools (such as TCE, see 2.2) it is possible to prototype and simulate architectures quickly which allows for rapid development.

The processor architecture proposed to this work is a Transport-Triggered Architecture (TTA). It resembles a traditional VLIW (Very Long Instruction Word) architecture. Compared to VLIW it allows the designer to specify data transports over the transportation network which allows more fine-tuned control over the program flow while retaining the flexibility of a programmable processor. The TTA's internal structure also suggests strong use of parallelism where several instructions can be executed at the same time. Hardware optimizations can be done by designing special hardware functional units which are directly accessible from the instruction set. This makes TTA suitable for demanding DSP applications.

The goal of this thesis is to design a TTA processor for computing FFT with a programmable length. A processor architecture suggested in a dissertation of Teemu Pitkänen [Pitkänen, 2014] has been used as a basis for the work. This thesis further improves the functionality, most notably reducing the length of a main computation loop to only one instruction.

1 FOURIER TRANSFORMS

Discrete Fourier Transform (DFT) is arguably one of the most frequently used algorithms in digital signal processing. It converts a finite sequence of equally spaced samples of a signal into a sequence of equally spaced coefficients. These represent amplitudes of complex sinusoids. We can write this relation as follows [Oppenheim and Schafer, 1989]:

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) W_N^{kn} \ , \tag{1.1}$$

where W_N denotes a complex sinusoid:

$$W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \ . \tag{1.2}$$

The sequence $X(k)$ is the Fourier series and $x(n)$ is the input sampled signal. Both k and n are integers in the interval $[0, N-1]$ where N is the size of the Fourier transform. j is the imaginary unit.

A direct calculation of 1.1 would require N^2 operations (complex multiplication followed by a complex addition). It is possible to reduce the computation complexity by decomposing the DFT into a series of smaller DFTs [Cooley and Turkey, 1965]. Sections 1.1 - 1.4 are dedicated to this problem.

By examining the coefficients $W_N = e^{-j2\pi/N}$ it is possible to observe certain periodicities and symmetries which can be exploited to reduce both a memory usage and a computation complexity. This topic is further described in Sections 1.5 - 1.6.

1.1 Radix-p Algorithms

According to Cooley-Turkey principle [Cooley and Turkey, 1965], if we can divide $N = PQ$, then N-point DFT can be computed using P-point and Q-point DFTs. If we can further decompose the DFT in a way that $N = p^q$, we talk about radix-p FFT algorithm. In this work we concentrate only on radix-2 and radix-4 algorithms.

In general there are two approaches to the FFT computation. First one, Decimation-in-time (DIT) is based on successively decomposing the input sequence $x(n)$ in smaller subsequences. Inversely, Decimation-in-frequency (DIF) algorithm is based on dividing the output sequence $X(k)$ [Oppenheim and Schafer, 1989]. Both approaches are equivalent in terms of an arithmetic complexity. The DIT algorithm, however, seems to show better SQNR (Signal-to-Quantization-Noise Ratio) for radix-2 and radix-4 FFTs implemented with a finite word length arithmetic [Chang and Nguyen, 2008]. Further implementation will consider this approach.

1.2 Radix-2

If it is possible to decompose the DFT into a structure where only 2-point DFTs are used, the algorithm is called *radix-2*. This means that the length of the sequence must be $N = 2^q$. We can divide the input sequence $x(n)$ into two subsequences of even and odd numbered n-s. By substituting $n = 2r$ for n even and $n = 2r + 1$ for n odd we obtain [Oppenheim and Schafer, 1989]

$$\begin{aligned} X(k) &= \sum_{r=0}^{N/2-1} x(2r)W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1)W_N^{(2r+1)k} \\ &= \sum_{r=0}^{N/2-1} x(2r)(W_N^2)^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)(W_N^2)^{rk} \ . \end{aligned} \tag{1.3}$$

With the knowledge of $W_N^2 = W_{N/2}$ we can rewrite the previous equation in the following form (substituting n by $2r$ or $2r + 1$):

$$\begin{aligned} X(k) &= \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk} \\ &= G(k) + W_N^k H(k) \ . \end{aligned} \tag{1.4}$$

The coefficients W_N^k are called *twiddle factors*. In the former equation, $G(k)$ and $H(k)$ are two $N/2$-point DFTs. Since $N = 2^q$ we can apply the same principle by dividing each of them into two parts again. Thus we would obtain 4 $N/4$-point DFTs. By repeating q-times we can proceed until the computation is reduced to only 2-point DFTs. This elementary 2-point DFT is called a *radix-2 butterfly* and its dataflow is illustrated in Fig. 1.1. Arrows denote a multiplication (when no operand is present, 1 is assumed - therefore just a straight transmission), additions are denoted by a plus sign in a circle. Solid black dot denotes a signal junction.

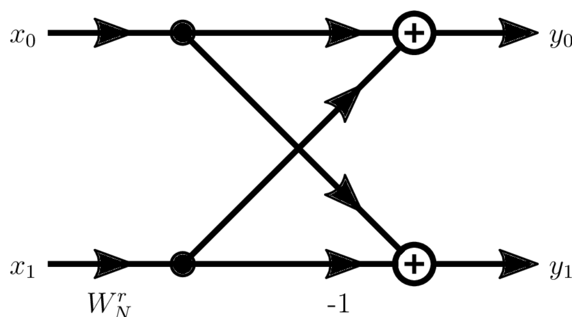


Fig. 1.1: Radix-2 butterfly

For the radix-2 butterfly it is possible to write

$$\begin{aligned} y_0 &= x_0 + W_N^r x_1 \\ y_1 &= x_0 - W_N^r x_1 \ . \end{aligned} \tag{1.5}$$

The resulting arithmetic complexity for one butterfly is two complex additions and one complex multiplication. Each stage contains $N/2$ butterflies and the whole algorithm has $\log_2 N$ stages. For the whole N-point FFT $N \log_2 N$ complex additions and $N/2 \log_2 N$ complex multiplications are needed.

1.3 Radix-4

If a sequence can be computed using only 4-point DFTs, the algorithm is called *radix-4*. This implies that the size of the sequence has to be $N = 4^q$. Radix-2 algorithm divides the input sequence $x(n)$ into two subsequences according to n being even or odd (mod 2). Radix-4 FFT divides the sequence into 4 $N/4$ DFTs based on n mod 4. In the next stage we break each of the $N/4$ stage into 4 parts again and continue until the computation is reduced to an elementary 4-point DFT - *radix-4 butterfly*. The resulting sequence can be written as follows (with n substituted by $4r$, $4r+1$, $4r+2$ or $4r+3$ and $W_N^4 = W_{N/4}$):

$$
\begin{aligned}
X(k) &= \frac{1}{N} \sum_{k=0}^{N-1} x(n) W_N^{kn} \\
&= \sum_{r=0}^{N/4-1} x(4r) W_{N/4}^{rk} + W_N^k \sum_{r=0}^{N/4-1} x(4r+1) W_{N/4}^{rk} \\
&+ W_N^{2k} \sum_{r=0}^{N/4-1} x(4r+2) W_{N/4}^{rk} + W_N^{3k} \sum_{r=0}^{N/4-1} x(4r+3) W_{N/4}^{rk}
\end{aligned}
\tag{1.6}
$$

The radix-4 butterfly structure is shown in Fig. 1.2 and its output can be written according to the Equation 1.7.

$$
\begin{aligned}
y_0 &= x_0 + W_N^{k_1} x_1 + W_N^{k_2} x_2 + W_N^{k_3} x_3 \\
y_1 &= x_0 - jW_N^{k_1} x_1 - W_N^{k_2} x_2 + jW_N^{k_3} x_3 \\
y_2 &= x_0 - W_N^{k_1} x_1 + W_N^{k_2} x_2 - W_N^{k_3} x_3 \\
y_3 &= x_0 + jW_N^{k_1} x_1 - W_N^{k_2} x_2 - jW_N^{k_3} x_3
\end{aligned}
\tag{1.7}
$$

One radix-4 butterfly has an arithmetic complexity of 6 complex additions and 3 complex multiplications as shown in Fig.1.2. Each stage contains $N/4$ butterflies and the whole radix-4 FFT can be performed in $\log_4 N$ stages. Therefore, the total arithmetic cost is $3N \log_4 N$ complex additions and $\frac{3N}{4} \log_4 N$ complex multiplications which is less than in radix-2 algorithm. Apart from that the number of stages is halved compared to radix-2.

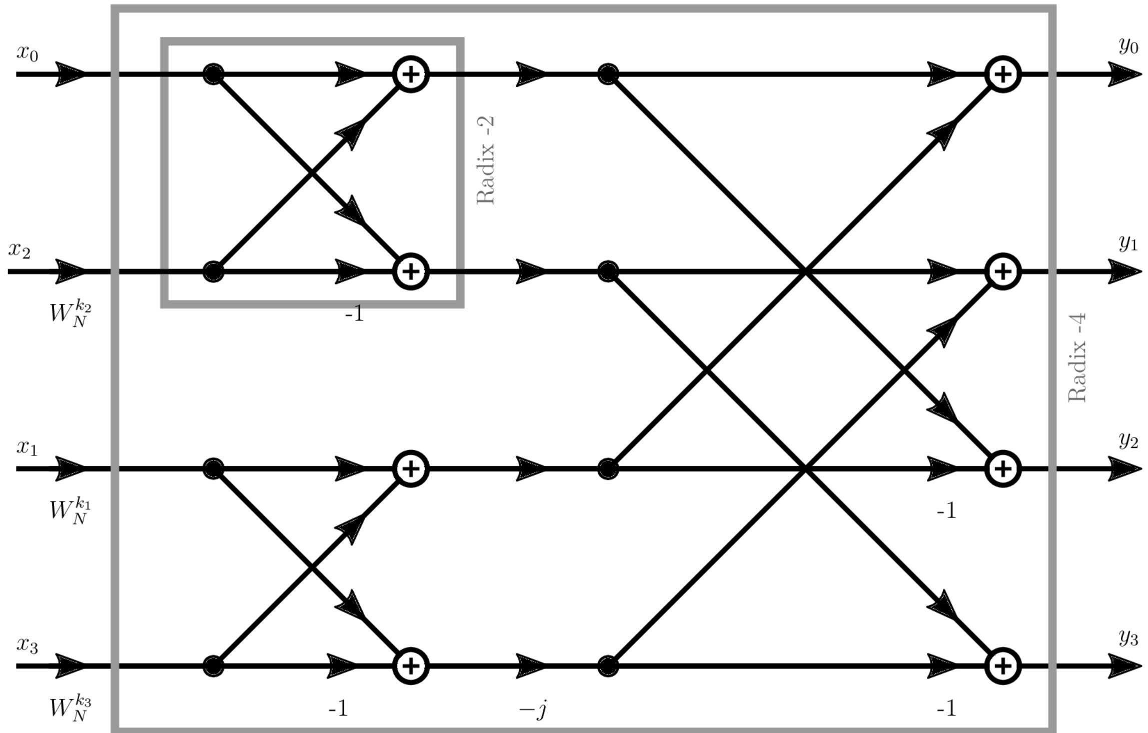


Fig. 1.2: Radix-4 butterfly can be computed combining several radix-2 butterflies

Higher-than-four radices (i.e. 8) are also possible to reduce the arithmetic complexity but they require more complex butterfly coefficients than trivial ± 1 and $\pm j$.

The main limitation of radix-4 (and any higher) FFT is that it requires N to be a power of four which greatly reduces the possible FFT sizes. This limitation can be overcome by using a *mixed radix* approach described in Section 1.4.

1.4 Mixed Radix

Mixed radix FFT is used in cases where we want the advantages of radix-4 FFT but also need to support all FFT sizes $N = 2^q$. The mixed radix FFT computes all stages except the last one using radix-4 butterflies. If q is an odd number the last stage is computed with radix-2 butterflies, otherwise radix-4 is used even for the last stage. This is the approach used in the thesis work.

1.5 Twiddle Factors

A special attention needs to be paid to the FFT coefficients - twiddle factors. They have the following form:

$$W_N^k = e^{-j2\pi k/N} = \cos(2\pi k/N) - j\sin(2\pi k/N) \ . \tag{1.8}$$

It can be observed that they are points on a complex unity circle, as shown in Figure 1.3. The figure contains all twiddle factors needed for computing FFT of size 64 (radix-4). Twiddle factors of sizes 32 (mixed radix) and 16 (radix-4) are shown using different markers and it is possible to obtain them from the 64-FFT coefficients.

In [Pitkänen, 2014], different methods of generating the twiddle factors are compared. Generally speaking we want to avoid direct computation which can create an unnecessary resource overhead. Instead, a method based on a Lookup Table (LUT) is used. All the necessary twiddle factors are pre-computed and stored into a memory for a future access. For relatively small FFT sizes (on which we focus) this method is the fastest and more memory efficient than some iterative polynomial methods. The LUTs require more memory (and therefore area) with increasing FFT sizes. However, it is not necessary to store all the twiddle factors. Possible memory optimization is described in Section 1.6.

1.6 Memory Optimization

According to Sections 1.2 and 1.3 the total number of radix-2 (radix-4) butterflies is $N/2\log_2 N$ $(N/4\log_4 N)$. Each butterfly contains 1 (3) multiplications with a non-trivial (not equal to one) twiddle factor. If we assume $N = 64$, the needed amount of memory entries in the LUT would need to be 192 (144).

However, if we allow some additional logic before the LUT it is possible to reduce the number of memory entries to the first $N/8+1$ coefficients [Pitkänen et al., 2007] from the first octant (sector B0 in the Figure 1.3). FFT of a size 64, therefore, needs only the first 9 values. Other values can be computed from the first octant by simple operations such as inverting the real and imaginary parts and multiplying by -1. In Figure 1.3, the relation needed to obtain the value in each sector is shown in the sector. Asterisk sign $*$ denotes a complex conjugate operation.

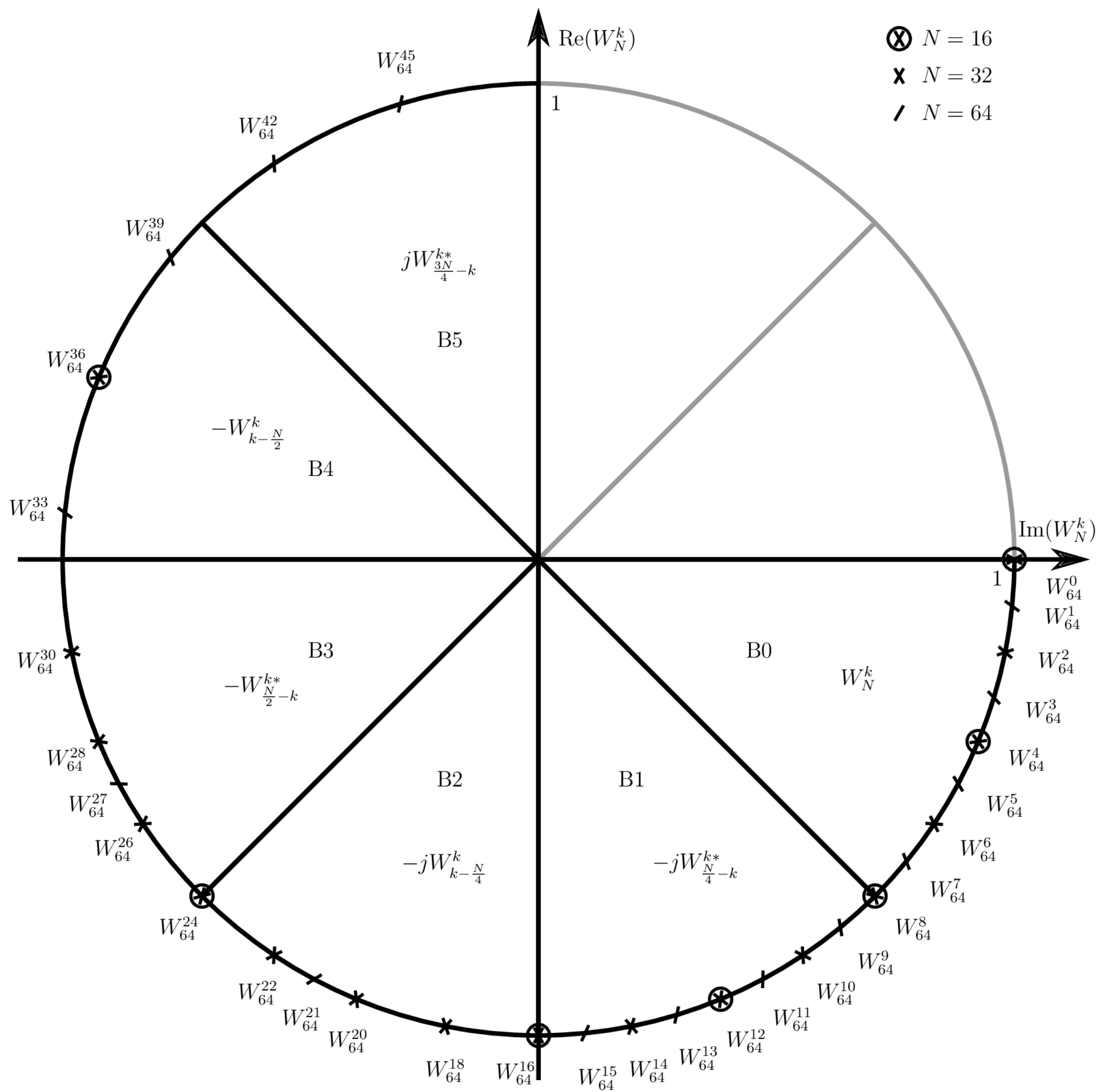


Fig. 1.3: Twiddle factors of FFT size 64

2 TRANSPORT TRIGGERED ARCHITECTURE

The proposed architecture for our FFT application is based on a Transport-Triggered Architecture (TTA) template [Corporal, 1998]. The architecture resembles a VLIW (Very Long Instruction Word) processor architecture ([Fisher, 1983], [Philips Semiconductors, 2011]). Its principle and advantages over VLIW are described in Section 2.1. A special tool for designing TTA processors, developed by Department of Pervasive Computing at TUT, is introduced in Section 2.2.

2.1 TTA Overview

Transport Triggered Architecture is a processor architecture where a designer has a full control over the data transports, specifying both a source and a destination. The instruction set has only one instruction: *move*. To perform an operation (e.g. AND) we first *move* an operand data to an operand input of a FU (Functional Unit). Operation itself is triggered by *moving* a data to a trigger input of the FU. After the operation is performed we can collect the result from the output of the FU.

The data is transported using an *interconnection network* which consists of a number of parallel buses connected to the inputs and outputs of FUs. The maximum number of *moves* performed in one clock cycle is limited by the number of buses. One bus can perform only one data transport per clock cycle (since it is just a wire). The number of buses is customizable as well as which FU inputs/outputs are connected to which buses. Furthermore, there is no limitation on the number of inputs and outputs of FUs.

The key property of TTA is that we can *move* data between functional units without the need to store them in Register Files (RF). FUs can distribute data between each other and thus reducing the pressure put on RFs. We can have RFs with a lower number of read and write ports and it is possible to implement them as traditional functional units [Corporal, 1998].

Some FUs have an access to a data memory. These are called LSUs (Load-Store Units). The program instructions are stored in an IM (Instruction Memory) and executed by a GCU (General Control Unit) which decodes them into the data transports.

An example architecture can be found in Fig. 2.1. The Interconnection Network consists of 6 buses. To these buses inputs and outputs of FUs, GCU, RFs and LSU are connected in a desired way. Connections to the buses are marked by a dot. Small asterlisks near the inputs of FUs mark the trigger input. By writing data to this port the desired operation will be triggered. Each FU can have multiple operations which are distinguished by an opcode provided with a data in the trigger port (the

opcode is generated automatically by the compiler). These operations can be generic (ADD, MUL, SHIFT, etc.) or we have the possibility to design our own and connect them via a standardized interface.

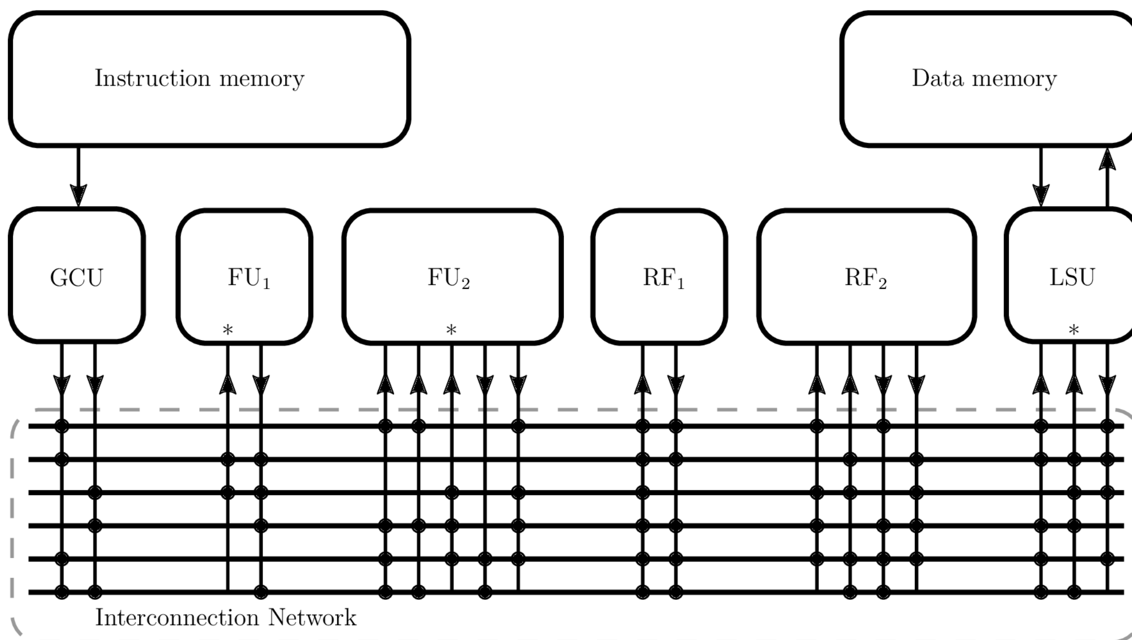


Fig. 2.1: Example of a TTA processor configuration

2.2 TTA-Based Co-Design Environment

TTA-Based Co-Design Environment (TCE) [Esko et al., 2010] is a TTA design toolset for a complete processor design and simulation from the high-level languages (C/C++) down to RTL (Register-Transfer Level) description (with support of VHDL or Verilog).

The toolset consists of a group of programs more or less tied together, sometimes wrapped in a GUI (Graphical User Interface). Future paragraphs give a brief overview of the ones used in this project. More info can be found in the toolset's documentation [CPC Group, 2017].

ProDe Processor Designer is a GUI program where the architecture is defined. FUs and their operations, registers and memory interface units are defined connected to the TTA interconnection network. The exact behaviour of the FU does not have to be implemented in order to be able to set it in ProDe. The only mandatory parameter is the operation's latency. The units in ProDe are more like „shells“ for the operations. The operations' behaviour can be added or changed later. Prode can

also produce implementation definition files which map RTL descriptions of FUs to the FUs placed within the software. These files can later be used in ProGe.

tcecc (tceasm) Compiler for a software written in C/C++ (or parallel assembly) for a TTA architecture. The compiler is retargetable which means that together with the program source file (C/C++ or parallel assembly) it is necessary to specify also the target architecture.

Proxim (ttasim) Proxim is a GUI version of ttasim - a processor simulator. It loads an architecture from ProDe, program from tcecc and simulates the program running on the architecture. It is possible to get various statistics such as a FUs' usage or a cycle count. The program also allows stepping through the program by one clock cycle, observing the instructions' execution and contents of a data memory, registers and FUs' ports. For the simulation it is necessary that operations of FUs have a defined behaviour in C/C++.

OSEd This program is a library and a manager of operations defined in C/C++ for the purpose of a simulation. It is possible to assign source files to particular operations and compile them. Also various operations' properties are set there (number of inputs/outputs, description, etc.). OSEd also contains a tester where it is possible to test the operation before it is used in a program. Many basic operations (add, substract, bit operaitons, memory read/write, etc.) are already included. It is possible to make a fully functional processor only with operations provided by TCE.

HDBEditor HDBEditor is responsible for mapping FUs to their RTL descriptions. This is not needed for the simulation purposes but it is necessary if we want to generate the RTL description of the processor. The tool creates a hardware database where a user groups the desired descriptions together. It is possible to have several variants of one FU and choose which one to use upon the processor generation. All the operations included in TCE by default are also implemented in a HDL and added to the default databases. There is, however, a difference between OSEd and HDBEditor in terms of what they describe. While OSEd describes operations separately, HDBEditor describes whole FUs. This means that for example an ALU (Arithmetic Logic Unit) functional unit with operations „add“ and „shl“ and an ALU with operations „add“, „shl“ and „sub“ need to have two different RTL implementations even though they share some operations. However, the FUs' interface is standardized and thus creating new FUs from existing operations is a matter of copy-pasting.

ProGe Processor Generator produces a synthesizeable RTL description of the whole processor. Its inputs are the processor architecture file and the implementation definition file. There is also an option to generate testbenches for the processor verification. ProGe also includes a platform integrator which can be used to interface the processor with specific FPGA FPGA boards.

PIG Program Image Generator converts a compiled program (from tcecc or tce-asm) into a binary image. This can be uploaded into a target machine's memory and executed as a program on an implemented machine. It is possible to apply a compression to reduce an instruction memory size.

3 UNTIMED REFERENCE MODEL

A Python program `refftta` [Žádník, 2017a] was developed as a high level model. The purpose of this model is to get familiar with the idea of the algorithm and architecture and to generate reference results which will be used later in a timed simulation and final RTL implementation. Section 3.1 gives an overview of the program and its features. Section 3.2 is dedicated to the description of the FFT algorithm used by `refftta`.

3.1 Program Overview

It follows the same dataflow as in the TTA processor. FUs are modelled as functions and their results correspond to the significant computation steps in the algorithm. However, as an untimed model it does not provide any details about an efficiency of the program/architecture and makes some assumptions and shortcuts not present in the lower levels of abstraction.

The program is written in a way which makes it possible and to retrieve data (intermediate results, input addresses, etc.) from the algorithm. The retrieved data can be used, e.g. for the evaluation in the terminal window or exported into a file as reference data. All computations are done using a 64-bit floating point data format. For the purpose of generating the reference files the data is converted to a fixed point representation.

A Python package NumPy [Oliphant and Community, 2017] was used for the numerical computing. It is a library optimized for computations with array objects in a same fashion as MATLAB. Apart from the array objects it can be used as a general numerical computing library since it contains many functions useful for numerical computing. While NumPy provides the necessary functionality and speed comparable to MATLAB, Python as a general purpose programming language is well suited to conveniently handle the rest of the features (such as command line options and file generation).

3.2 Detailed Description

The code in Figure 3.1 is the core part of the Python program. It has been modified and shortened for a better readability for the purpose of this document. Therefore, it should be treated more as a pseudo-code despite using a Python syntax.

A computation of one butterfly consists of several fundamental steps:

1. Generate input addresses for a butterfly and fetch the butterfly inputs.
2. Generate k-coefficients for twiddle factors.

3. Based on the k, fetch a twiddle factors from an LUT.
4. Multiply the butterfly inputs with twiddle factors.
5. Perform the butterfly operation (complex adder).
6. Store the results back to the same addresses.

First the program generates a lookup table of twiddle factors of size 2049 ($N/8+1$ for $N = 2^{14}$). The information about the current FFT size is stored within the $Nexp$ parameter: $N = 2^{Nexp}$. Then it iterates over the computation stages, each time checking which radix to use. In every stage, linear index (`lin_idx`) is iterated with a step of four. The algorithm computes one radix-4 butterfly or two radix-2 butterflies at the time. In the final implementation, the linear index iterates with a step of one and the results are completed sequentially.

Based on the linear counter, four addresses of input operands are generated (`ag`). The addresses are generated by taking two (one in case of $Nexp$ being odd) LSBs (Least Significant Bits) and placing them between the rest of the bits. The exact position where to insert the LSBs is determined by `stage` (see Section 4.3.1). In the last stage, the operands are accessed linearly.

Next the indices (k) for twiddle factors (W_N^k) are computed by `gen_tf_k`. The indices computed are different for radix-4 and radix-2, therefore we need to specify `rdx2_flag` in the input. From Figure 1.3 it is apparent that twiddle factors for smaller FFTs can be derived from the larger sizes by taking only every 2nd, 4th, 8th, etc. twiddle factor. Therefore, after the generation of k-s the function `gen_tf_k` also performs scaling according to the current FFT size. The scaling itself is just bit-shifting k-s to the left as necessary. By using the k-scaling we do not need to pre-compute a twiddle factor LUT for each FFT size separately since it is enough to have only one for the largest supported N.

After getting the necessary k-s we can feed them to the twiddle factor generator and get four complex valued twiddle factors. The twiddle factor generator generates the appropriate values using principles described in Section 1.5.

Now is the time to compute the butterfly itself. The butterfly input (`X`) are four values selected from the input sequence (`x`) specified by the generated addresses (`addr`). We multiply the butterfly input with twiddle factors and feed the products (`P`) into a modified complex adder (`cadd`). This function either performs one radix-4 butterfly according to 1.7 or two radix-2 butterflies according to 1.5 based on `rdx2_flag`.

The result of the butterfly is stored into `Y` and these values are stored back to `x` to the same addresses from where they were read to `X`. This technique is called *in-place computation*. It is used to save memory by using only one vector `x` where both inputs and outputs are stored. When the butterfly computation is completed, its results rewrite the input values.

```python
# Generate lookup table of twiddle factors
tf_lut = gen_tf_lookup(2**14)

# Compute FFT (input sequence x)
for stage in range(total_stages(Nexp)):
    # Whether to use the radix-2 stage (radix-4 is default)
    rdx2_flag = rdx2_stage(stage, Nexp);
    # For every butterfly (4 samples)
    for lin_idx in range(0, Nexp, 4):
        addr = ag(lin_idx, stage, Nexp)
        k    = gen_tf_k(lin_idx, stage, Nexp, rdx2_flag)
        tf   = tfg(k_sc, tf_lut, Nexp)
        # Butterfly input
        for i in range(4):
            X[i] = x[addr[i]]
        P    = multiply(tf, X)
        # Compute butterfly
        Y    = cadd(P, rdx2_flag)
        # Assign the result to the original memory
        for i in range(4):
            x[addr[i]] = Y[i]

# Rearrange result
for i in range(0, Nexp):
    idx = bit_reverse(i, Nexp)
    y[i] = x[idx]
```

Fig. 3.1: Abbreviated Python script for computing FFT

After performing all butterfly computations in a given stage, the algorithm proceeds to the next stage, checks whether this stage is radix-2 and radix-4 and repeats the above procedure. After running through the whole computation the result is in a permuted order and needs to be reordered. This is done by reversing bit pairs (in case of $Nexp$ being odd the LSB pair is not a bit pair but a single bit) of the indices of the result's samples.

4 TIMED SIMULATION MODEL

In this chapter, a timed simulation model of both the suggested TTA architecture and the FFT algorithm is provided. A simulation workflow in the TCE environment is described in Section 4.1. In Section 4.2, the suggested TTA architecture is suggested. Custom FUs and their operations are discussed in Section 4.3. In Section 4.4, the FFT computation program is described. It is written in the TTA assembly language as a cycle-accurate succession of parallel instructions. Section 4.5 summarizes the results of the timed simulation.

4.1 Simulation Workflow

The typical simulation workflow is illustrated in Fig. 4.1. In dark boxes, TCE programs are listed (their overview is in Section 2.2). The programs' contents examples are displayed in white boxes. Files are denoted by white ellipses. User's input is marked by the light grey circle. Generally speaking, a simulator's user needs to specify the TTA architecture, then write and compile a program for it. The simulator then simulates the compiled program on the specified architecture. The simulation itself is performed by Proxim (GUI) or ttasim (CLI - Command Line Interface).

First, a user needs to decide which operations are necessary. The TCE toolset already containing a wide selection of common operations. If some non-typical operation is needed, the user needs to specify the operation's static behaviour in a form of a C/C++ source file. This source file is then compiled by OSEd. OSEd also creates the operation's entry in the operation database and specifies its parameters (i.e. the number of input/output ports and their width). The operations parameters are stored in the operation description file.

This file is a part of the operations database and is necessary for ProDe to recognize the operation and add it to the design. In ProDe a full processor architecture design is done. Also operations' latencies are defined there. Each operation needs to have specified an estimated hardware latency (number of clock cycles needed to perform the operation). This latency is used by the simulator to perform a clock cycle-accurate simulation. This includes defining the FUs, adding operations to them and connecting them to the IC network. Address spaces (i.e. data and instruction memory) are defined. Data widths of transport buses as well as their number is also customizable. Apart from these, many other options can be set up. ProDe generates and architecture description file (in a XML format) as its output.

The second user's contribution (this time mandatory) is the intended program. It can be either written in C or the TTA parallel assembly language. In both cases, the program is compiled (by tcecc or ttasim) into a program binary file. Both compilers

require the architecture description file generated by ProDe as the second input. It is therefore possible to compile the same program on more machines resulting in different binary files. This way it is possible to test an architecture's performance for the program and tweak the architecture based on the simulation results. This is most useful when compiling C programs. Programs written in the assembly language have all the moves specified manually.

When the program is compiled it can be finally loaded into the simulator (together with the architecture definition file). The simulator runs the program on the architecture, monitoring all data transfers on all IC buses in every clock cycle. It is possible to monitor the contents of data memories and registers and data on the input and output ports of FUs. It is also possible to obtain statistics such as FU and register usage percentage and total clock cycle count.

4.2 Proposed Architecture

The proposed TTA architecture schematic is on Fig. 4.2. The IC consists of ten 32-bit buses (B0–B9) and one 1-bit but (bool). To these buses, FUs are connected. A connection is denoted by a black dot. In the upper row, there are custom FUs created for the purpose of this thesis. Their parameters are described in a section 4.3. In the lower row are FUs and modules provided by TCE. From the left to the right they are: adder, load/store unit (read only), load/store unit (write only), shifter, register file and general control unit. Trigger ports are denoted by an asterisk. Apart from FUs connected to the IC, there are also three types of memories: data memory containing samples of the computed FFT sequence, instruction memory containing the program encoded into instruction words and a LUT containing precomputed twiddle factors for TFG.

The adder serves as a linear counter for the program. Shifter unit and the register file are used in the setup stage of the program (section 4.4). General control unit is responsible for decoding and executing instruction words from the instruction memory. A part of the unit is also a *loop buffer* which allows caching one or more instruction words for efficient repeated reading (section 4.4).

Two load/store units were used to fetch and store computed samples to/from a data memory. During the computation, only one read and one write operation are used simultaneously. Therefore, read-only and write-only units can be used. In the timed model, the data memory consists of only one memory unit which differs from an actual implementation.

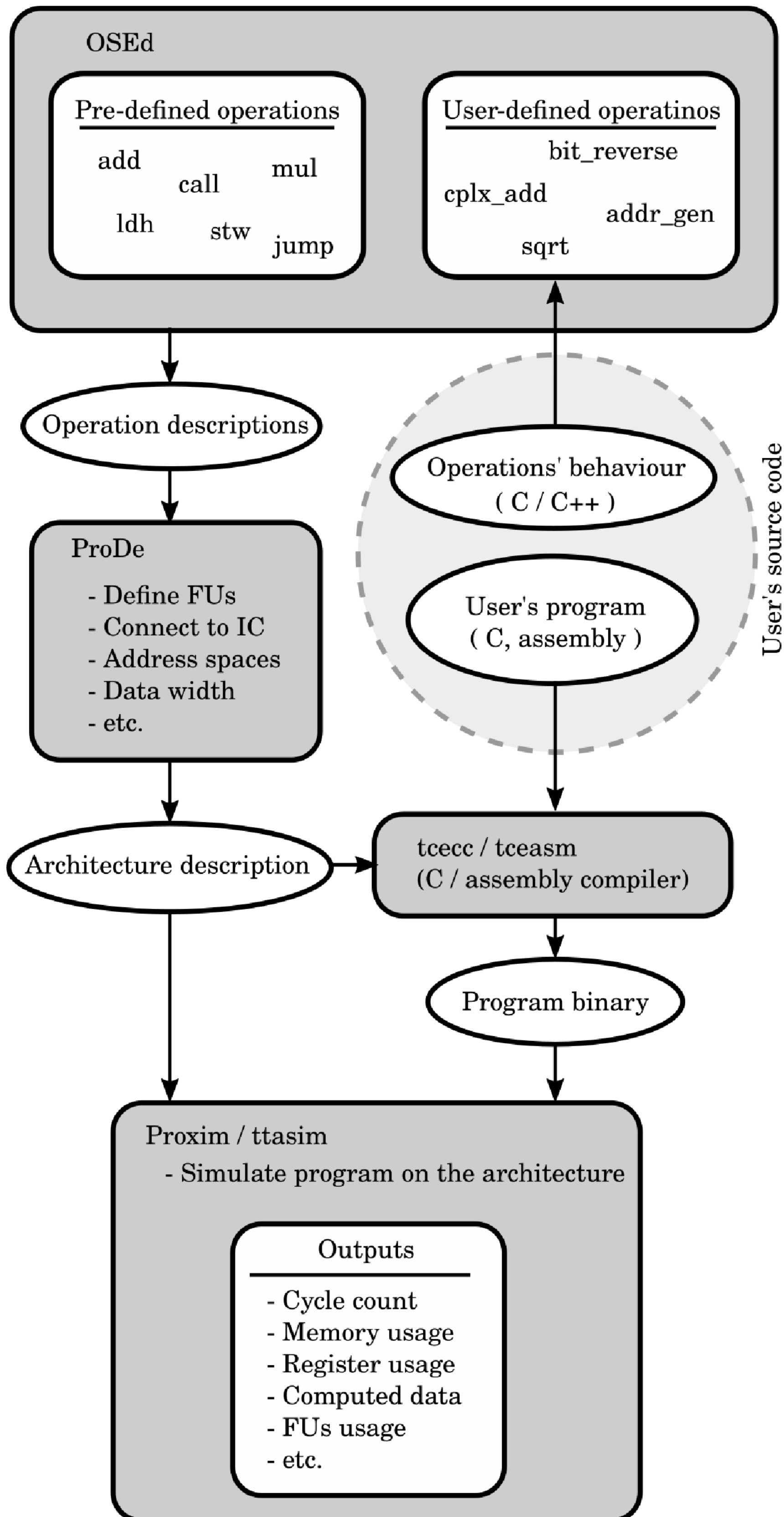


Fig. 4.1: Simulation workflow in TCE; White ellipses are files; Dark boxes are TCE programs; White boxes are what the programs contain; Grey circle is a user's input

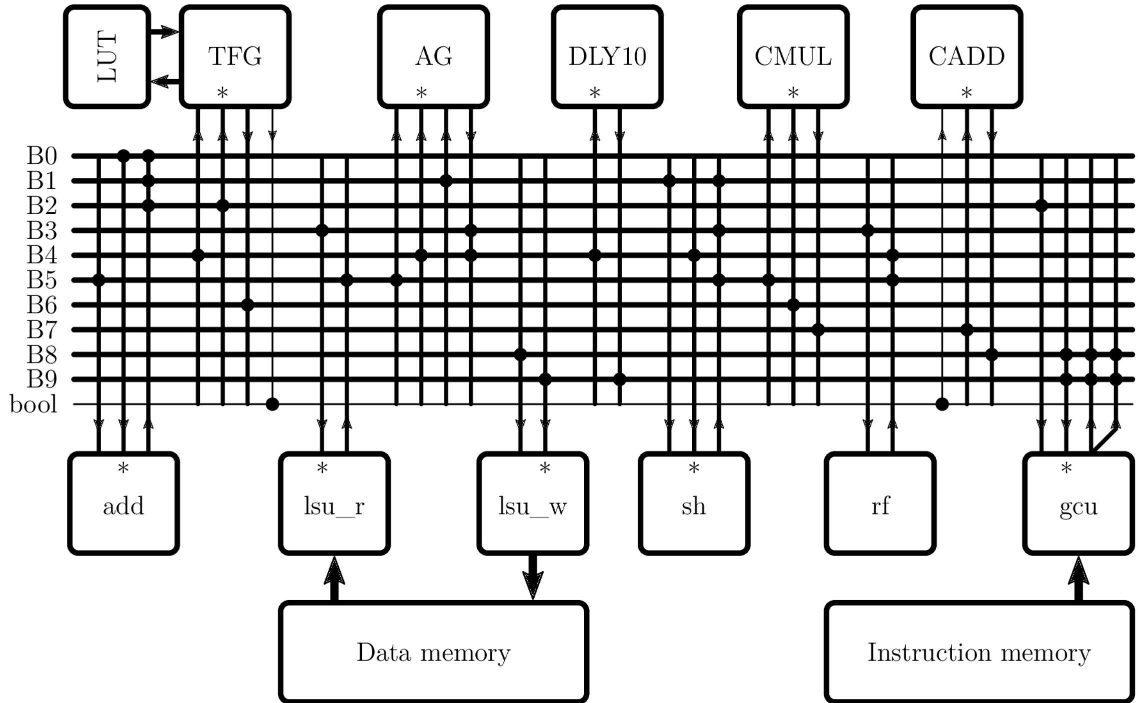


Fig. 4.2: Diagram of the proposed TTA architecture with an interconnection network

4.3 Functional Units

Before considering the hardware implementation of the FUs, their simulation behaviour needs to be defined (in C/C++) and the FUs need to be tested properly. In this work, five custom FUs were defined: Address Generator (AG), Twiddle Factor Generator (TFG), Complex Adder (CADD), Complex Multiplier (CMUL) and a delay block (DLY). All of the FUs are implemented into the TCE operation database and can be used in user programs.

Each of the custom FUs contains only one operation corresponding to the FU's title. These operations are described in following subsections. Each subsection contains an operation's overview table which summarizes its outputs, inputs (trigger denoted as $*$) and estimated latency. The initial estimate of latency for each FU was based on [Pitkänen and Takala, 2011].

4.3.1 Address Generator

Address Generator (Table 4.1) computes a memory address of an operand which needs to be accessed. The necessary information about a current stage can be obtained by separating `N_exp` (see below) LSBs from a linear counter. This counter is

not reset on each stage and counts from 0 to the value of $N \cdot M - 1$ where M is the total number of stages.

The FFT length is not given as an absolute value. Instead it is passed as an exponent (q) of 2^q using the `N_exp` input. `base_addr` is used to point the beginning of the input sequence. Without this parameter we would always address operands on memory addresses starting at 0.

In order to fit into the schedule, either AG needs to have a latency 3 or TFG latency 5. I chose the pessimistic variant and estimated AG to have a latency 3.

AG	

Inputs	∗`lin_idx`
	`N_exp`
	`base_addr`

Output	`addr`
Latency	3

Table 4.1: Address Generator - ports

4.3.2 Twiddle Factor Generator

Twiddle Factor Generator (Table 4.2) uses the same concept of a linear counter an FFT size exponent as an AG. Contrary to the Python script the generation of k is embedded inside the FU.

Apart from a twiddle factor, TFG also outputs a radix-2 flag which is fed into a complex adder to control the desired operation.

TFG	
Inputs	∗`lin_idx` `N_exp`
Outputs	`tf` `rx2`
Latency	6

Table 4.2: Twiddle Factor Generator - ports

		-1	0	1	2	3	4	5	6	7	8	9	10	11
ports	t	-	a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3	0	0	0	0
	$rx2$	-	0	0	0	0	1	1	1	1	0	0	0	0
	r	-	0	0	0	0	r_0^a	r_1^a	r_2^a	r_3^a	r_0^b	r_1^b	r_2^b	r_3^b
memory	m_0	0	a_0	a_0	a_0	a_0	b_0	b_0	b_0	b_0	0	0	0	0
	m_1	0	0	a_1	a_1	a_1	a_1	b_1	b_1	b_1	b_1	0	0	0
	m_2	0	0	0	a_2	a_2	a_2	a_2	b_2	b_2	b_2	b_2	0	0
	m_3	0	0	0	0	a_3	a_3	a_3	a_3	b_3	b_3	b_3	b_3	0
	$rx2$	0	0	0	0	0	1	1	1	1	0	0	0	0
operands	o_0	0	0	0	0	a_0	a_0	a_0	a_0	b_0	b_0	b_0	b_0	0
	o_1	0	0	0	0	a_1	a_1	a_1	a_1	b_1	b_1	b_1	b_1	0
	o_2	0	0	0	0	a_2	a_2	a_2	a_2	b_2	b_2	b_2	b_2	0
	o_3	0	0	0	0	a_3	a_3	a_3	a_3	b_3	b_3	b_3	b_3	0
	$rx2$	0	0	0	0	0	0	0	0	1	1	1	1	0
		$op0$	$op1$	$op2$	$op3$	$op0$	$op1$	$op2$	$op3$	$op0$	$op1$	$op2$	$op3$	$op0$

Fig. 4.3: Schedule of a complex adder

4.3.3 Complex Adder

Complex Adder (Table 4.3) takes four operands and performs their summation according to the `opcode`. The FU can calculate one y_i from either Equation 1.7 or Equation 1.5. The opcode specifies the desired radix and an output being computed, thus altering the behaviour of the adder. Complex addition is composed by a several real additions (of 16 bit integers). To avoid an overflow the result of each real addition is divided by two.

Operands are fed one by one. After four of them are loaded, CADD sequentially produces the desired four outputs, one at a time. The CADD's schedule is shown on Fig. 4.3. The input `rx2` determines whether to perform one radix-4 butterfly or two radix-2 butterflies. This automatic scheduling was necessary for making the main program kernel fit into one instruction.

CADD	
Inputs	`*t`
	`rx2`
Output	`res`
Latency	1

Table 4.3: Complex adder - ports

4.3.4 Complex Multiplier

Complex Multiplier (Table 4.4) performs a complex multiplication of two complex numbers according to the following principle:

$$\begin{aligned} \mathrm{Re}(res) &= \mathrm{Re}(in_1) \cdot \mathrm{Re}(in_2) - \mathrm{Im}(in_1) \cdot \mathrm{Im}(in_2) \\ \mathrm{Im}(res) &= \mathrm{Re}(in_1) \cdot \mathrm{Im}(in_2) + \mathrm{Im}(in_1) \cdot \mathrm{Re}(in_2) \end{aligned} \tag{4.1}$$

Each of the internal multiplication needs to be bit-shifted left by 15 bits to avoid an overflow.

CMUL	
Inputs	`in1`
	`*in2`
Output	`res`
Latency	3

Table 4.4: Complex multiplier - ports

4.3.5 Rotating Register

The rotating register is used to delay an input data by a specified number of clock cycles (10 in this case). Because of that, the unit is called DLY or DLY10 in the design files and this document. It stores the result addresses from AG and provide them in the end of the butterfly computation cycle to the LSUs' store operation. When using this unit it is no longer necessary to store the addresses in register files and load them back when needed. Instead the latency of this unit is set that the addresses pop out just at the time when they are necessary. This reduces the logic to maintain the register files addresses manually and thus making it possible to shrink the execution body into a single instruction.

DLY10	
Inputs	*in1
Output	out
Latency	10

Table 4.5: Rotating register - ports

4.4 Instruction Schedule

The program code consists of four parts. First a setup code configures the computation by setting a base memory address and FFT length. Number of iterations and parameters for FUs are computed and fed in the non-trigger inputs. These parameters are static and do not need to be computed again unless the FFT length changes. The setup code can also vary depending on how we load the FFT length. Generally, it can be read from a memory (slower) or fed in as an immediate number (faster).

After the initial setup the FFT computation starts. First let's explain the idea using only one butterfly (Fig. 4.4). Each numbered row corresponds to one instruction word where the number on the left is the instruction number. An instruction word consists of up to 11 instructions executed on buses „B1" – „B10" (32-bit) and „bool" (1-bit). The bus names are shown in the bottom row and correspond to the buses used of the architecture's IC network. In each column, only one instruction (data move) is repeated. On the top row the source and the destination in each move is shown. A detailed description of the notation is in the Tab. 4.6. It follows the TTA assembly syntax standard. The instructions do not change so in each column only one instruction is repeated over and over. If a field is grey, the instruction is executed. The white squares denote that no move was performed.

A linear counter is maintained by feeding the adder's output into its trigger ('1' was stored as its operand in the setup stage). The adder's output is fed into an address generator (AG) and a twiddle factor generator (TFG). The computed address is fed into a load/store unit (LSU) to fetch an input sample. At the same time the address is being delayed by 10 clock cycles using a rotating register (DLY10). When an operand is fetched from the LSU, it is moved into a complex multiplier (CMUL) together with a twiddle factor from (TFG). TFG also provides an information about a current radix to the complex adder (CADD) using the „bool" bus. The CMUL multiplies the twiddle factor with the from the LSU input sample.

The important thing here is that the results from LSU and TFG need to arrive in the same clock cycle. This constraint creates a requirement on the latencies (l_{FU})

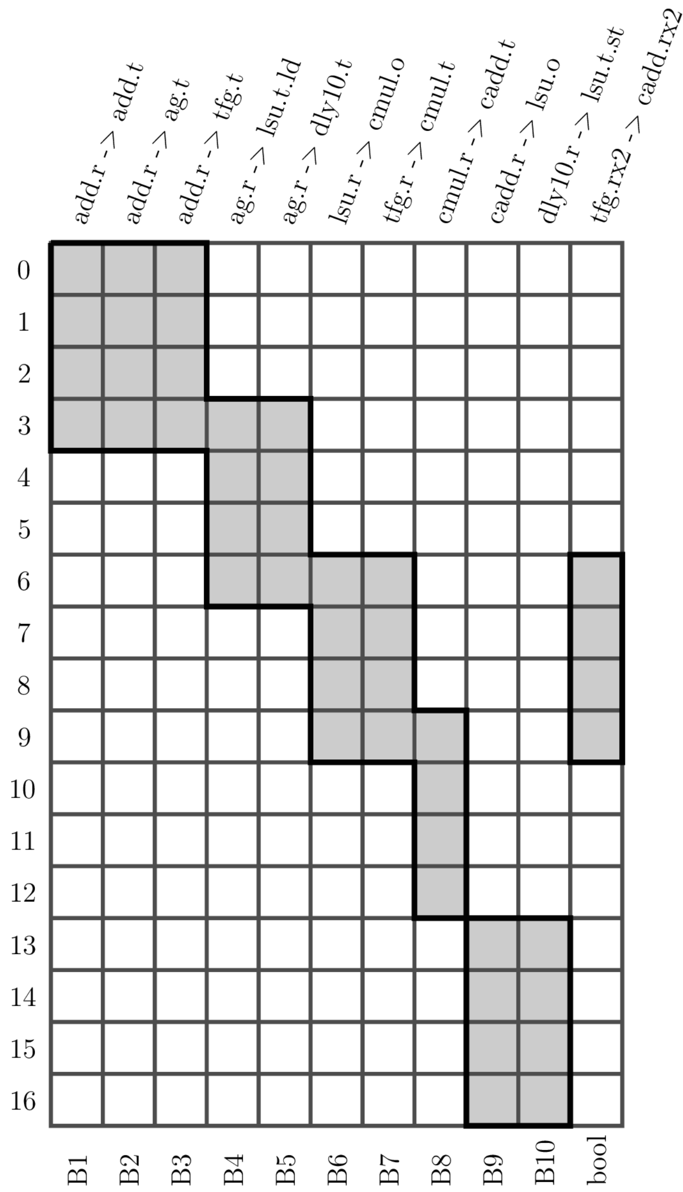


Fig. 4.4: Radix-4 butterfly instruction schedule and reservation table

FU names	
`add`	Adder
`ag`	Address generator
`tfg`	Twiddle factor generator
`lsu`	Load/store unit
`dly10`	Rotating register (10 clock cycles delay)
`cmul`	Complex multiplier
`cadd`	Complex adder
Port names	
`.o`	operand
`.t`	trigger
`.t.st`	trigger; executing a „store“ operation
`.t.ld`	trigger; executing a „load“ operation
`.rx2`	input for a current radix information
`.r`	result

Table 4.6: Instruction notation format

of AG, LSU and TFG:

$$l_{AG} + l_{LSU} = l_{TFG} \ . \tag{4.2}$$

If the LSU and TFG were not synchronized, it would be necessary to use additional registers to store the intermediate result.

With this approach, intermediate register transfers can be avoided and the data can be moved only between the FUs. The output from CMUL is moved into a complex adder (CADD). After four operands are fed in, CADD starts outputting four values from the butterfly operation, one in each clock cycle. The `.rx2` input of CADD configures the unit to use either one radix-4 butterfly or two radix-2 butterflies. The results from CADD need to be stored back to the memory in the same position from which they were read. This is done by feeding the delay address (from DLY10) and the value (from CADD) into both inputs of a LSU. One butterfly computation takes 17 clock cycles and is based on estimated latencies of the hardware FUs.

The computation is performed on complex numbers. They are 32 bits long, 16 bits for the real (LSB) and 16 bits for the imaginary part (MSB). The representation of both real and imaginary part is fixed point, Q15.

The instruction schedule was specifically designed to support software pipelining. We do not need to wait until one butterfly is computed before starting a new one.

Instead, the butterfly pattern can be repeated every 4 clock cycles. On Fig. 4.5 (middle) the software pipelining is demonstrated on a computation of a 16-point FFT. The computation is computed in two stages using 8 butterflies and takes 45 clock cycles to complete.

The computation can be divided into three parts: *prologue* (0–12), *kernel* (13–31) and *epilogue* (32–44). In both *prologue* and *epilogue*, the instructions keep changing every few clock cycles. However, during the *kernel* the instruction word is the same for the whole time. This is more noticeable in larger FFT sizes. Both *prologue* and *epilogue* keep the same length (13 clock cycles) and only the *kernel* grows. Obviously, in large FFTs the majority of the execution time is spent in the *kernel* part.

With only one instruction word repeating in the *kernel* it is possible to reduce the instruction schedule to only 27 instructions (Fig. 4.5 (right)). The instruction 13 is repeated a predefined number of times (the exact number is computed during the setup and is based on the FFT size). In order to avoid repeated reading (and thus consuming power) the same instructions from the instruction memory, a loop buffer is used to store the *kernel* instruction word and repeat it. Using a loop buffer has the advantage that it does not consume as much power as reading the instructions from a memory [Guzma et al., 2010]. Thus a more compact code and better power efficiency can be achieved.

4.5 Summary

The table 4.7 summarizes how many clock cycles are spent inside the kernel compared to the total number of cycles. The total number of cycles for a computation of FFT of size N can be computed as

$$total = N * nstages + setup + epilogue = N * nstages + 6 + 13 \tag{4.3}$$

where $nstages$ is the required number of stages for the current N. *Epilogue* and *setup* are clock cycles spent in those stages. The number of clock cycles of the *kernel* phase can be computed as the total number of clock cycles without the clock cycles spent in *setup*, *prologue* and *epilogue* phases:

$$kernel = total - setup - prologue - epilogue = total - 6 - 13 - 13\ . \tag{4.4}$$

The last column of table 4.7 shows a ratio $kernel/total$ as a percentage. It is apparent that the majority of the computation is spent inside the kernel. In case of larger FFTs almost all the computation is done in the kernel. The kernel consists of one instruction repeated in an instruction loop buffer. The power cost of reading from the buffer is comparable to reading from a register file which can be assumed to

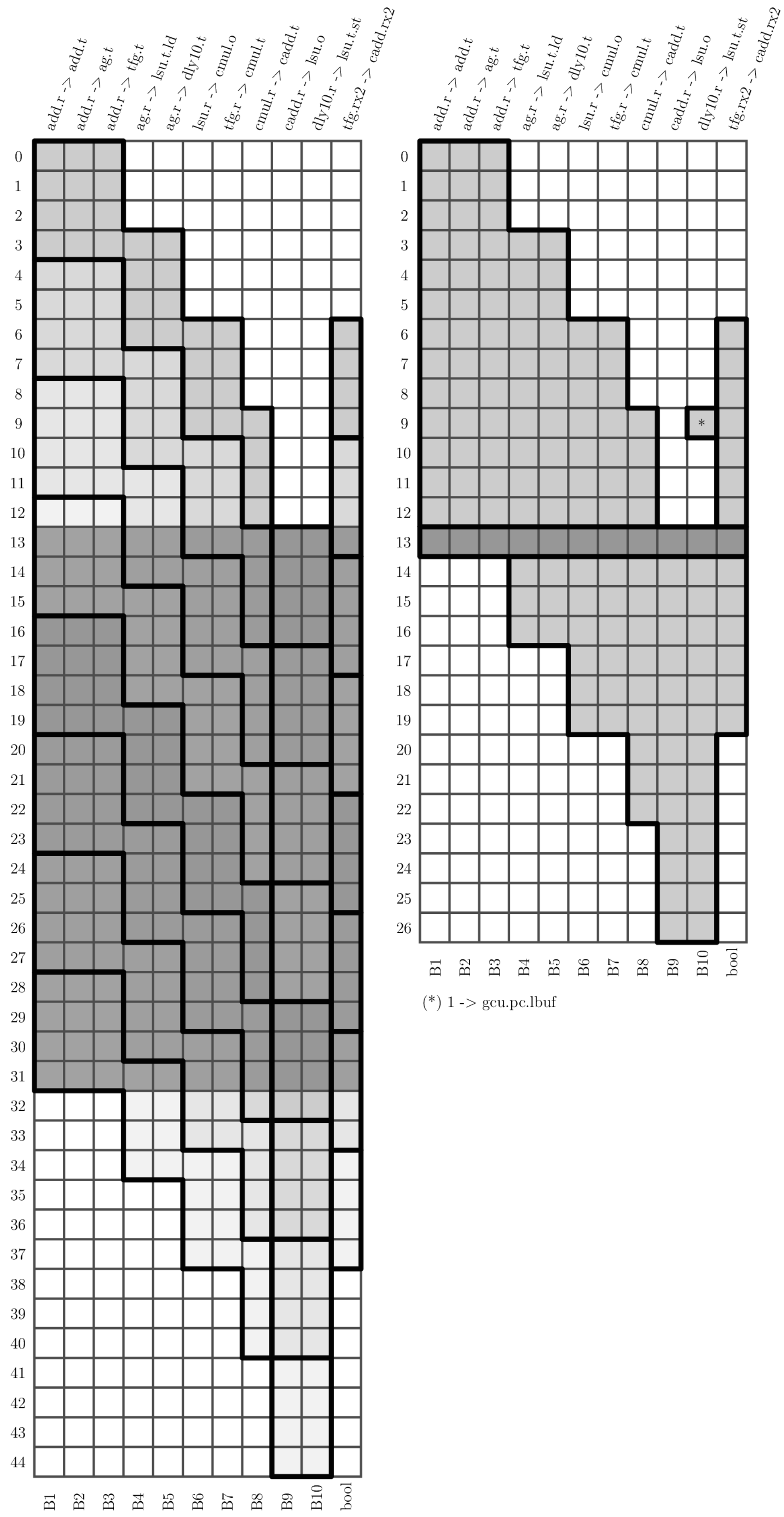


Fig. 4.5: Instruction schedule and reservation table of FFT of size 16.

be lower than reading from a memory [Guzma et al., 2010]. Therefore, a significant improvement in terms of a power efficiency is expected compared to the instruction memory approach.

FFT size	kernel	total	% of total
32	83	115	72.17
64	179	211	84.83
128	499	531	93.97
256	1011	1043	96.93
512	2547	2579	98.76
1024	5107	5139	99.38
2048	12275	12307	99.74
4096	24563	24595	99.87
8192	57331	57363	99.94
16384	114675	114707	99.97

Table 4.7: Number of required clock cycles for all supported FFT sizes.

The simulation currently performs with relatively low precision due to the complex multiplier. The error in a real or imaginary part of the result ranges between 10^{-5} and 10^{-3}. It is due to the fact that the model of a complex multiplier does not perform any rounding and overflows are avoided just by truncating the LSBs. This concern is addressed in the hardware implementation where a proper rounding and truncating is specified as necessary. Despite the precision loss, the simulation showed that the concept is feasible for a hardware implementation.

5 HARDWARE IMPLEMENTATION APPROACH

The processor is going to be synthesized on an ASIC technology in order to evaluate its power efficiency. However, the processor's description is universal and allows an implementation on an FPGA, too. The two implementations will differ only in details and a types of memories used. Before considering the ASIC synthesis, FUs need to be described and verified first. An FPGA device (ZYNQ XC7Z020-1CLG400C on a PYNQ-Z1 board) was used as a synthesis target for evaluating the FUs. At the time of writing this document, all custom FUs are described in a VHDL language and their verification is in the process. Because of that, this chapter gives an approach to the hardware design rather than definitive solutions.

Section 5.1 describes the process of designing a TTA processor with the aid of TCE. In section 5.2, designs of custom FUs are presented. Section 5.3 briefly describes memory types used and an access to them.

5.1 Design Workflow

The processor implementation workflow is depicted in Fig. 5.1. First step is to describe all the necessary FUs in a hardware definition language. The basic ones are already provided by TCE. Using HDB Editor, a hardware database file is created. This file is a library of all the possible HDL implementations for the required FUs. It maps the HDL description and parameters to the FUs used in the architecture description (ProDe). One FU can contain multiple HDL descriptions in the library (i.e. different latencies) so it is easy to switch between different implementations. All HDL descriptions use the same standardized interface which allows for a convenient mapping to the FUs ports.

An architecture definition file is also needed for the generation. It is obtained from ProDe during the timed simulation phase (section 4). Both the hardware database and the architecture definition are used as inputs to ProGe - Processor Generator. ProGe's GUI can also be accessed from ProDe. In the GUI, it is possible to specify a desired HDL implementation for each FU and register file as well as other parameters of the processor. The configuration can be saved into an implementation definition file to save time selecting the parameters in the future. ProGe generates a full HDL description of the processor, optionally a testbench and a compilation and simulation scripts (for GHDL and ModelSim).

In order to be able to simulate a program on the architecture, the program's instruction memory image has to be generated. This can be done in PIG (Program

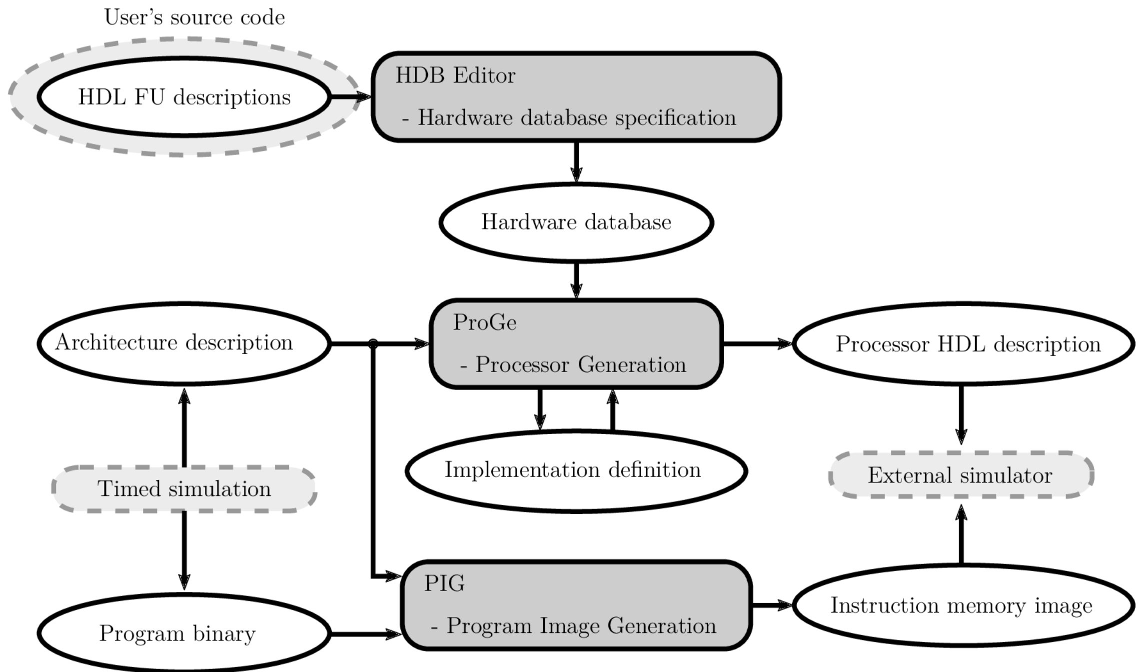


Fig. 5.1: Processor implementation workflow in TCE

Image Generator). It needs the program binary, also generated during the timed simulation phase. The program creates the instruction memory image which is then used by the processor instruction decoder (part of the gcu FU) to decode the instructions and execute them on the processor.

The generated processor can be then simulated and synthesized using third-party tools (such as GHDL, ModelSim, Xilinx Vivado, etc.).

5.2 Functional Units

In this section, a description of the hardware implementation of each custom FU is given. Some FUs contain multiple possible implementations (differing i.e. in the latency) as it is easy to swap them in the processor generation process. The latencies suggested in the timed simulation (chapter 4) are based on [Pitkänen, 2014] and serve as a starting point. However, they might be a subject of change depending on the final requirements and restrictions. The final FU's latency is a trade-off between speed on one side and a resource usage, power efficiency on the other. For example the complex multiplier can be implemented with a lower latency than 3 but it would require using four multipliers instead of only two (see 5.2.4).

The FUs also have a lock and reset mechanism but they are left out from the block diagrams for clarity. Generally, a unit accepts operands and computes values only when its trigger input port is triggered. In case of units with a delayed output

(such as the rotating register), the unit will keep outputting values every clock cycle even without triggering. The reset signal resets all registers and states to zeros.

The standardized FU interface ensures that all inputs are registered. Thus a minimal latency of a FU is one clock cycle (in case of a purely combinatorial FU).

5.2.1 Address Generator

The address generator (Fig. 5.2) is used to compute the address of an input/output sample for the in-place computation. First, a linear counter (lin_idx) is separated into two signals containing the information about the current stage (MSBs of lin_idx) and a sample index within the current stage (LSBs of lin_idx). The number of bits where lin_idx is separated into $stage$ and idx is determined by $nexp$ where $nexp$ is the current FFT size as a power of two: $N = 2^{nexp}$.

The address generation itself is computed by splitting idx into $left$ and $right$ parts and inserting two LSBs of idx between them (shifting the $right$ part two bits to the right). The position (pos) where to insert the $lsbs$ is computed by shifting $stage$ by one bit to the left and subtracting it from $nexp$. The address computation is finished by concatenating $left$, $lsbs$ and $right$ together (from MSB to LSB).

The final address is then multiplied by four (shift left by two positions). This is architecture-specific and given by the current bus width and a Minimum Addressable Unit (MAU). The proposed architecture's MAU is 8 bits but the numbers are stored as 32-bit words, therefore one number consists of four MAUs.

The $base_addr$ input signal adds a static offset to the computed memory address.

5.2.2 Twiddle Factor Generator

Twiddle factor generator (Fig. 5.3) is probably the most complicated FU used. In general, first, a twiddle factor coefficient k is computed. It is converted into an address of a twiddle factor in an LUT (see 1.6). The twiddle factor is then fetched from the LUT and manipulated in order to get the correct value.

As in the address generator, the input linear index is separated into $stage$ and idx. The base of k is generated as modulo 4 (in case of radix-4 being computed) or modulo 2 (in case of radix-2) of idx. The current radix is determined by the $rx2$ signal which is computed by evaluating the $nexp$ and $stage$. When the computation is in the last stage and the $nexp$ is odd, radix-2 computation is being used and the $rx2$ flag is set to one. The base of k is then multiplied by a weight (w) generated by reversing bit pairs of idx (in a total length of $nexp$). The multiplication is not using an actual multiplier. Instead, a constant multiplication principle is used since the base k can be only either 0, 1, 2, or 3. The weighted k is then scaled by shifting left according to the current computation stage and maximum allowed FFT size

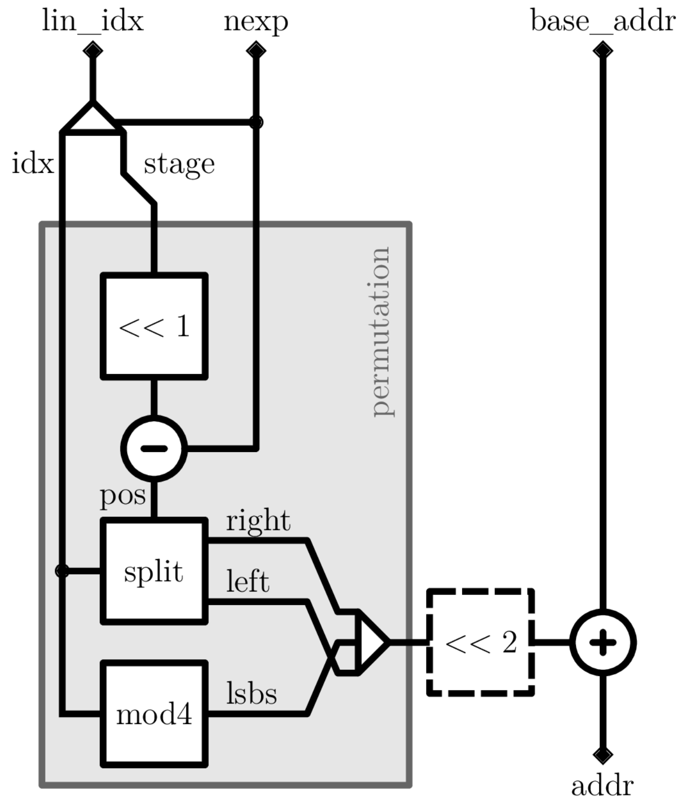


Fig. 5.2: Block diagram of an address generator

(currently 2^{14}). This scaling is necessary because an LUT of a maximum size (2049) is used which is the native for FFT of size 2^{14}. When computing smaller FFT, some coefficients of larger FFTs are not used and the k needs to be scaled in order to access the right values.

The scaled k is then converted into a read address for the LUT. As described in 1.6, only $N/8+1$ twiddle factors are stored in the LUT. The scaled k, however, spans a larger range and needs to be converted to the LUT address. The maximum allowed ranges for a scaled k and *addr* are in the „convert“ box in the block diagram. The converting unit also produces an opcode (*opc*) to specify a manipulation of the twiddle factor retrieved from the LUT. The manipulation consists of a combination of the following operations (imaginary part in the diagram on Fig. 5.3 is denoted by a dotted line):

1. Negate the real part
2. Negate the imaginary part
3. Swap real and imaginary parts

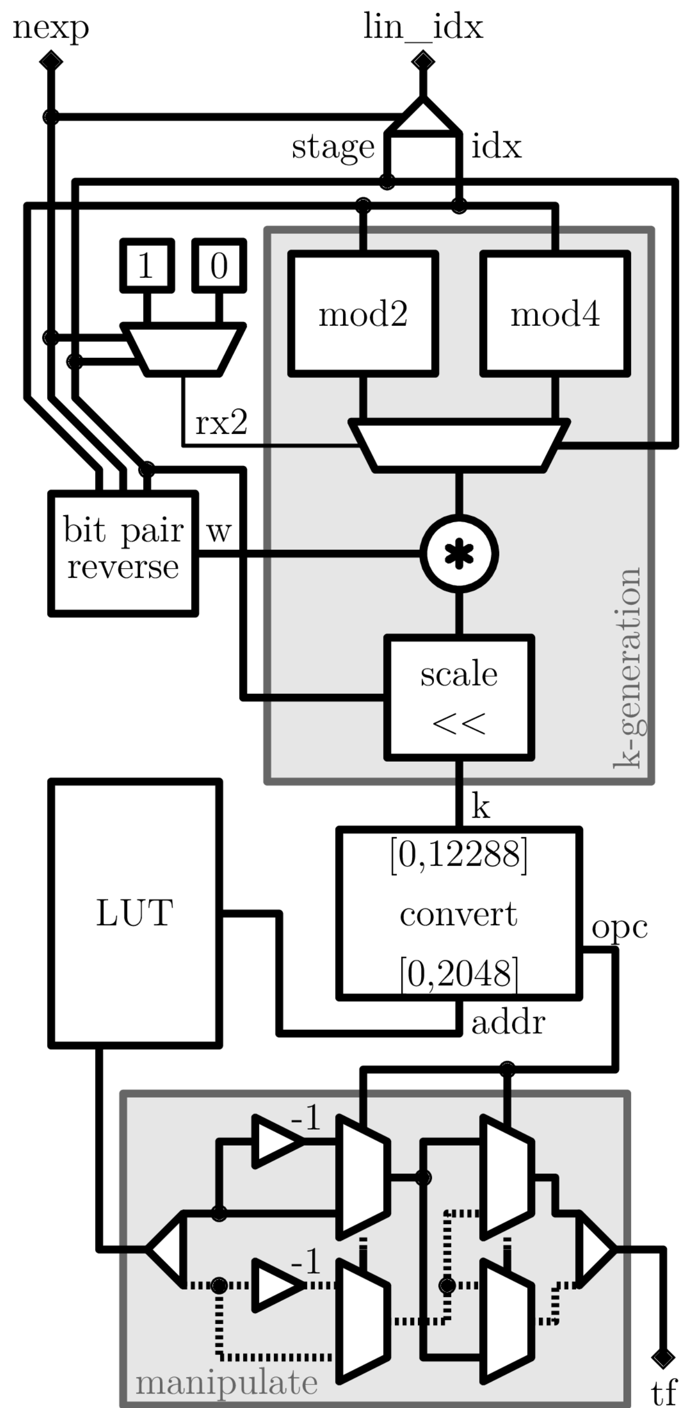


Fig. 5.3: Block diagram of a twiddle factor generator

5.2.3 Complex Adder

The complex adder performs the butterfly operation with four operands (a, b, c and d). The unit consists of two parts: a sequential register and a combinatorial logic part (Fig. 5.4). The sequential part basically serves as a serial/parallel converter for a four-input combinatory complex adder. The sequential part behaves like a state machine of four states. The state is generated as a 2-bit linear counter. It is incremented only when a new value is loaded into the t port (every new load is indicated by the *load* input signal).

Let's assume an input sequence of four numbers a, b, c and d, loaded after each other in the t input port, each time triggering the *load* port. The state machine's behaviour can be described according to the value of the counter (*cnt*) as follows:

- 0 : Store a and *rx2* into registers
- 1 : Store b into a register
- 2 : Store c into a register
- 3 : Load all register contents into the combinatory part (d is directly transferred - it is not registered)

The combinatory part computes four results in a sequence with the same operands (a, b, c and d). After four clock cycles a new set of four operands is loaded.

On a diagram in a Fig. 5.4, dotted lines represent imaginary parts of the input numbers. Solid lines are the real parts. An *rx2* input signal provides the information about the current radix and together with *cnt* they provide an opcode (*opc*) for the combinatory part. The opcode selects inputs of multiplexors, the negation of multiplexors' outputs and an add/subtract mode of the adders.

5.2.4 Complex Multiplier

Complex multiplier performs a multiplication of two complex numbers. A pipelined version is shown on a block diagram on Fig. 5.5. The operation corresponds to the behaviour described by equation 4.1. However, the number of multipliers has been reduced to only two and the computation takes two clock cycles.

The unit's inputs are a and b. First, the real part of the result is computed and after rounding and truncating it is delayed by a register. In the next clock cycle, a select (*sel*, one bit) signal is inverted and an imaginary part is computed. The delayed real part and the immediate imaginary part are concatenated into a result (p).

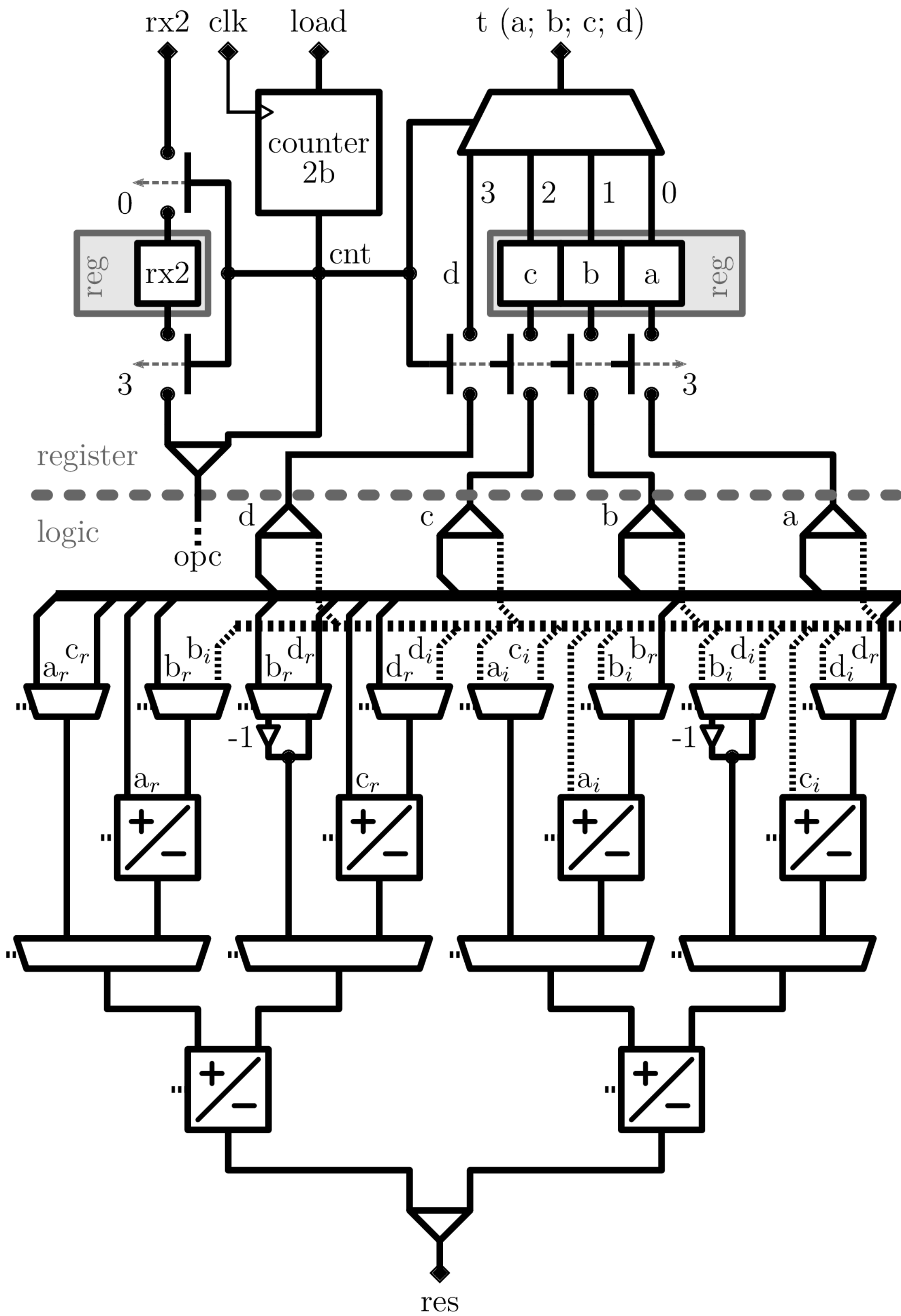


Fig. 5.4: Block diagram of a complex adder

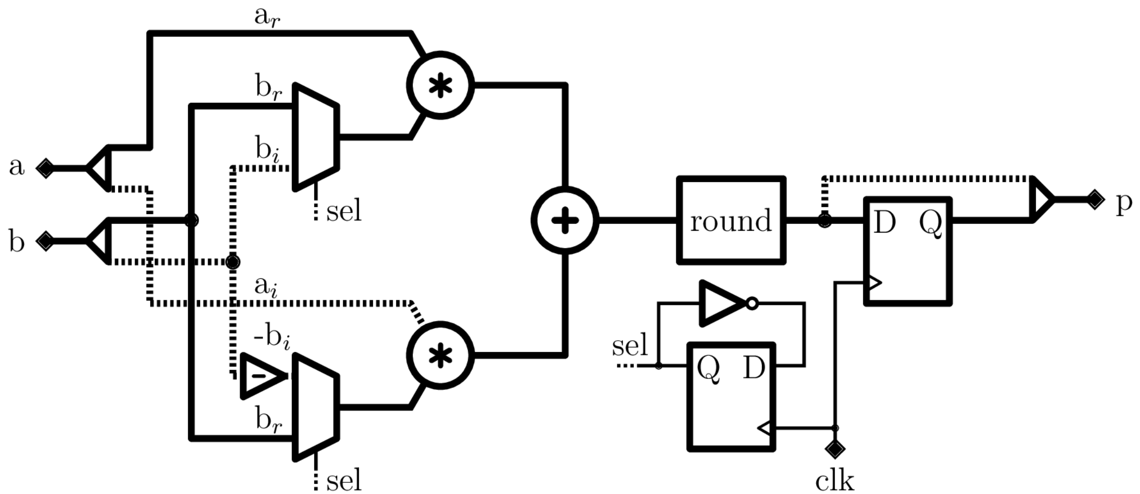


Fig. 5.5: Block diagram of a complex multiplier

5.2.5 Rotating Register

This FU serves as a delay unit, delaying an input by 10 clock cycles. The rotating register was chosen instead of a shift register because of a lower power consumption. In a shift register, all values in all registers are replaced with the next values. This constant replacing is inefficient. In our case (length of 10), it would mean writing 10 values simultaneously every clock cycle. Rotating register keeps values in their addresses. An address counter is determining where a new value will be stored and from where an output value will be read in the current clock cycle. Thus only one write and one read operations are performed per clock cycle. Compared to a shift register there is the overhead of the modulo 10 counter. However, when working with 32-bit numbers its influence on a power consumption is smaller than switching ten 32-bit values every clock cycle.

The unit's block diagram is on Fig. 5.6. The address counter computes the write address. The read address is one clock cycle before the write address. This means that the read address pointer will be on the position of the current write address pointer 10 clock cycles later.

5.3 Memories

There are three kinds of memories in the proposed architecture (Fig. 4.2):

1. Instruction memory
2. Data memory
3. Twiddle factor lookup table

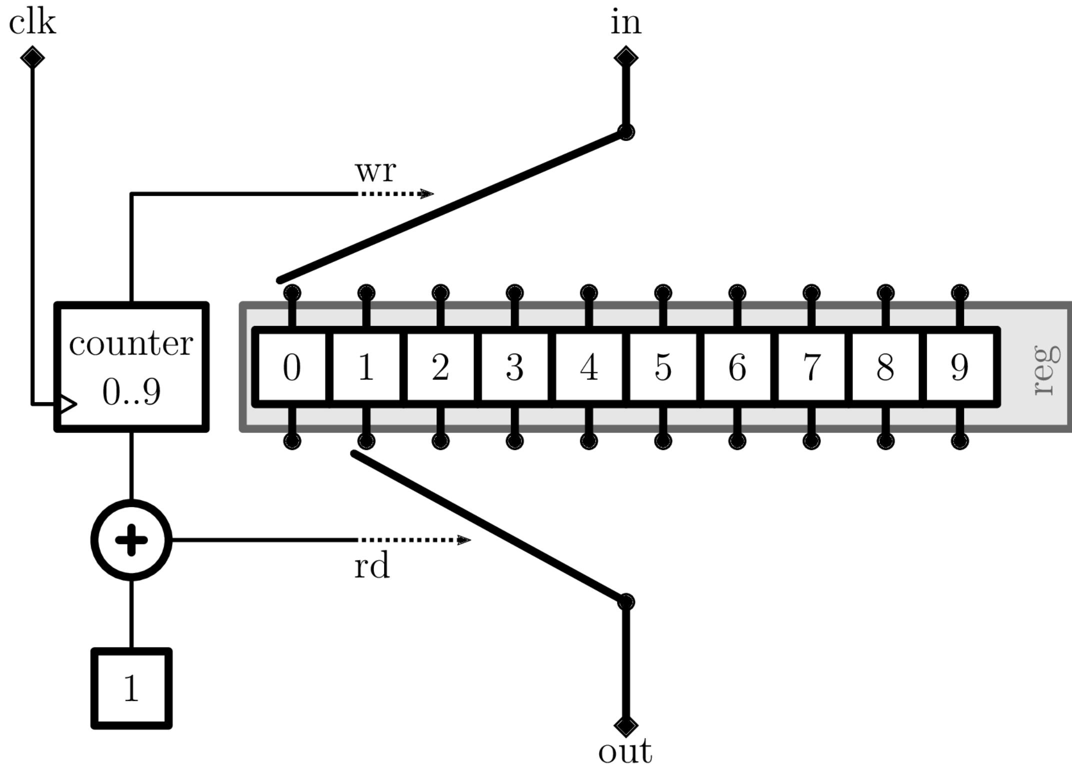


Fig. 5.6: Block diagram of a rotating register

Their implementation can vary depending on the target platform and it is up to the user to specify them. TCE can generate the data and instruction memories for a testbench as synchronous SRAM (Static Random-Access Memory) memories. Otherwise, TCE can provide the data and instruction memory contents in several formats. It can be in a form raw data to be put into a user's memory module, or as a premade VHDL array package. The data and instruction memory were not in a consideration so far since the focus is, at the time of writing this, on implementing and verifying the custom FUs. The twiddle factor LUT is a ROM (Read Only Memory). At the time of the writing, it is implemented as a synchronous ROM using two FPGA's BRAM (Block RAM). However, this will be changed into a distributed LUT implementation for a better power efficiency and future ASIC implementation.

During the kernel of the instruction schedule, two parallel memory accesses are performed each clock cycle (one read and one write). Therefore, two load/store units need to be used. [Pitkänen et al., 2009] suggests using a parallel dual-port memory as the memory organization for the twiddle factor LUT. It was compared to a single dual-port memory and showed a smaller area and power consumption while maintaining the same performance. Since the memory is divided into two separate memory modules acting like single-port memories, an address assigning mechanism needs to be used. Therefore, an additional permutation logic needs to be inserted

between the load/store units and memory modules. This logic was provided by TUT and will be used in the ASIC synthesis.

6 CONCLUSION

This work introduces an efficient FFT processor based on a TTA architecture template. As a basis of this work, a dissertation project of Teemu Pitkänen [Pitkänen, 2014] was used. Several modifications were made compared to the previous implementation. Mainly, some of the FUs were modified to support the new instruction schedule and the usage of registers has been reduced to only one rotating register.

These modifications allow for a more optimized instruction schedule. While the kernel of the previous project consists of 16 instruction words, the kernel proposed in this thesis consists of only one instruction word. This instruction word is then placed in an instruction loop buffer and executed from there. It has been shown that majority of the computation is spent in the kernel loop. Since reading from the loop buffer consumes less power than reading from an instruction memory, power savings are expected compared to the Pitkänen's approach.

A timed model for the architecture was developed and simulated using a TCE toolset with an addition of a several custom FUs. The work on a hardware implementation has begun by describing the custom FUs and verifying them. However, the final implementation of the processor has not been completed yet. At the time of writing the thesis, the work on this project is being continued under as a contract job at TUT with the aim to complete the synthesis of the processor on an ASIC technology to compare the results against the Pitkänen's results. The results from the simulation also suggest to use rounding and truncating techniques when computing arithmetic operations.

As side products of the thesis project, two Python programs were developed. Namely `refftta` [Žádník, 2017a] - for a generation of reference values and `reftest` [Žádník, 2017b] - for automatic compilation and simulation of TTA programs and their evaluation against the reference values. All the project files and programs were version controlled using Git and placed online for a public access (with the exception of the loop buffer which is a property of the CPC research group at TUT). The repositories were grouped together under a „FFT on TTA“ group available at `https://gitlab.com/fft-on-tta`.

BIBLIOGRAPHY

Chang, W.-H. and Nguyen, T. Q. (2008). On the fixed-point accuracy analysis of FFT algorithms. In *IEEE Transactions on Signal Processing*, volume 56. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.650.1613&rep=rep1&type=pdf`.

Cooley, J. W. and Turkey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. In *Mathematics of Computation*, volume 19, pages 297–301. `http://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf`.

Corporal, H. (1998). *Microprocessor Architectures from VLIW to TTA*. John Wiley & Sons Ltd., Chichester.

CPC Group (2017). TTA-based co-design environment. `http://tce.cs.tut.fi`.

Esko, O., Jääskeläinen, P., Huerta, P., de La Lama, C. S., Takala, J., and Martinez, J. I. (2010). Customized exposed datapath soft-core design flow with compiler support. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 217–222, Washington, DC, USA. IEEE Computer Society. `https://tutcris.tut.fi/portal/files/1970608/esko_customized_exposed_datapath_soft_core_design_flow.pdf`.

Fisher, J. A. (1983). Very long instruction word architectures and the ELI-512. In *ISCA '83 Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150. Association for Computing Machinery (ACM).

Guzma, V., Pitkänen, T., and Takala, J. (2010). Reducing instruction memory energy consumption by using instruction buffer and after scheduling analysis. In Nurmi, J., Takala, J., Vainio, O., and Salmela, P., editors, *2010 International Symposium on System-on-Chip Proceedings, 29-30 September 2010, Tampere, Finland*, pages 99–102. Contribution: organisation=tkt,FACT1=1.

Oliphant, T. and Community (2017). NumPy. `https://www.numpy.org`.

Oppenheim, A. V. and Schafer, R. W. (1989). *Discrete-Time Signal Processing*. Prentice Hall, New Jersey.

Philips Semiconductors (2011). An introduction to very-long instruction word (VLIW) computer architecture. `https://web.archive.org/web/20110929113559/http://www.nxp.com/acrobat_download2/other/vliw-wp.pdf`.

Pitkänen, T. (2014). *Fast Fourier Transforms on Energy-Efficient Application-Specific Processors*. PhD thesis, Tampere University of Technology. https://tutcris.tut.fi/portal/files/1429016/pitkanen.pdf.

Pitkänen, T., Partanen, T., and Takala, J. (2007). Low-power twiddle factor unit for FFT computation. In Vassiliadis, S., Bereković, M., and Hämäläinen, T. D., editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation: Proc. 7th Int. Workshop SAMOS VII*, pages 233–240, Berlin, Germany. Springer-Verlag. https://tce.cs.tut.fi/papers/samos_twiddle.pdf.

Pitkänen, T. and Takala, J. (2011). Low-power application-specific processor for fft computations. In *Journal of Signal Processing Systems*, volume 63, pages 165–176. Springer.

Pitkänen, T., Tanskanen, J., Mäkinen, R., and Takala, J. (2009). Parallel memory architecture for application-specific instruction-set processors. *Journal of Signal Processing Systems*, 57(1):21–32. Contribution: organisation=tkt,FACT1=1.

Žádník, J. (2017a). Refftta - reference FFT computation following the TTA dataflow. https://gitlab.com/fft-on-tta/refftta.

Žádník, J. (2017b). Reftest - an automatic compilation, simulation an evaluation tool for TTA programs and processors. https://gitlab.com/fft-on-tta/reftest.