BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS
ÚSTAV RADIOELEKTRONIKY

PARALLELISM IN DIGITAL SIGNAL PROCESSING
PARALELISMUS V ČÍSLICOVÉM ZPRACOVÁNÍ SIGNÁLŮ

DOCTORAL THESIS
DIZERTAČNÍ PRÁCE

AUTHOR — **Ing. Roman Mego**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Tomáš Frýza, Ph.D.**
ŠKOLITEL

BRNO 2020

Abstract

The doctoral thesis is focused on the systems for digital signal processing, its architecture and possibilities of software development. The text discussed the basic classification of computer systems from the view of parallel processing. It also demonstrates the behavior of the low-level and high-level programming languages on the multicore digital signal processors based on VLIW architecture. The aim of the dissertation thesis is to develop a tool that can be used to implement any DSP algorithm on the any VLIW processor with efficiency of the low-level programming languages, but with the advantages of the high-level programming languages. Result is the software that uses a signal-flow graph approach to describe an algorithm, and generates the low-level assembly code.

Keywords

digital signal processing, VLIW architecture, software development, signal-flow graph

Abstrakt

Dizertační práce je zaměřena na systémy pro číslicové zpracování signálů, jejich architekturu a možnosti vývoje softwaru. Text pojednává o základním rozdělení počítačových systémů z hlediska paralelního zpracování dat. Rovněž demonstruje chování nízkoúrovňových a vysokoúrovňových programovacích jazyků na vícejadrovém signálovém procesoru založeném na architektuře VLIW. Cílem dizertační práce je vytvořit nástroj, který může být použitý při implementaci DSP algoritmů na VLIW procesory s efektivností nízkoúrovňových programovacích jazyků, ale s výhodami vysokoúrovňových programovacích jazyků. Výsledkem je software, který využívá pro popis algoritmů graf signálových toků a generuje kód v jazyce symbolických adres.

Klíčová slova

digitální zpracování signálů, VLIW architektura, vývoj softwaru, graf signálových toků

Mego, Roman. Parallelism in digital signal processing: doctoral thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Radio Electronics. 2020. Supervised by doc. Ing. Tomáš Frýza, Ph.D.

Declaration

I declare that I have written my doctoral thesis on the theme of Parallelism in digital signal processing independently, under the guidance of the doctoral thesis supervisor and using the technical literature and other sources of information which are all quoted in the thesis and detailed in the list of literature at the end of the thesis.

As the author of the doctoral thesis I furthermore declare that, as regards the creation of this doctoral thesis, I have not infringed any copyright. In particular, I have not unlawfully encroached on anyone's personal and/or ownership rights and I am fully aware of the consequences in the case of breaking Regulation § 11 and the following of the Copyright Act No. 121/2000 Sb., and of the rights related to intellectual property right and changes in some Acts (Intellectual Property Act) and formulated in later regulations, inclusive of the possible consequences resulting from the provisions of Criminal Act No. 40/2009 Sb., Section 2, Head VI, Part 4.

Brno, 19. August 2020 ……..

Author's signature

Contents

Introduction....1

1 State of the art....2

1.1 System classification....2
1.1.1 Single instruction, single data (SISD)....2
1.1.2 Single instruction, multiple data (SIMD)....3
1.1.3 Multiple instructions, single data (MISD)....3
1.1.4 Multiple instructions, multiple data (MIMD)....4
1.2 Individual cases of processor architectures....4
1.2.1 Scalar central processing units (CPU) and digital signal processors (DSP)....4
1.2.2 Graphics processing units (GPU)....6
1.2.3 Very long instruction word (VLIW)....6
1.2.4 Multicore systems with shared memory....7
1.2.5 Multicore systems with distributed memory....8
1.2.6 Multicore systems with hybrid distributed-shared memory....8
1.3 Programming methods....9
1.3.1 Low-level languages....9
1.3.2 High-level languages....10
1.3.2.1 Difference between compiled and interpreted languages....11
1.4 Standard optimization methods....11
1.4.1 Redundancy elimination....12
1.4.2 Constant propagation optimization....12
1.4.3 Useless code elimination....13
1.4.4 Inline expansion....13

2 The objectives of the dissertation thesis....15

3 Effectiveness of software development tools....16

3.1 Multicore DSP TMS320C6678....16
3.1.1 TMDSEVM6678LE Development Board....17
3.2 Test cases....18
3.2.1 Data and thread parallelism using OpenMP....18
3.2.1.1 Section work-sharing....18
3.2.1.2 Loop work-sharing....19
3.2.2 Algorithm parallelization in OpenMP....19
3.2.2.1 FIR filter....20
3.2.2.2 Discrete Fourier transform....20
3.2.2.3 Fast Fourier transform....20
3.2.3 Measured performance of OpenMP....21

3.2.4 Low-level optimizations of the algorithms on the VLIW architecture.....24
3.2.5 High-level and low-level comparison.....25
3.2.5.1 Low-level assembly.....25
3.2.5.2 Linear assembly.....26
3.2.5.3 High-level language.....27
3.2.6 Comparison of the libraries with different structure.....28
3.3 Chapter summary.....29
4 Impact of the software efficiency to the power consumption.....30
4.1 Theoretical power consumption increase on multi-unit systems.....30
4.2 Practical test cases.....31
4.2.1 Case 1: Empty loop.....31
4.2.2 Case 2: Load/Store operations.....32
4.2.3 Case 3: Fixed-point operations.....32
4.2.4 Case 4: Floating-point operations.....33
4.2.5 Case 5: FFT routines.....33
4.3 Experimental Results.....34
4.4 Chapter summary.....37
5 Instruction mapping tool for DSPs.....38
5.1 The idea of signal-flow graph approach.....38
5.2 Input files.....39
5.2.1 Architecture definition.....39
5.2.1.1 Hardware resources.....40
5.2.1.2 Instruction set.....44
5.2.2 Algorithm description.....47
5.3 Algorithm mapping.....48
5.3.1 Input files parsing.....49
5.3.1.1 Parsing signals.....49
5.3.1.2 Parsing nodes.....50
5.3.1.3 Multi-operation nodes.....52
5.3.2 Finding relations and validation.....53
5.3.2.1 Extending nodes and signals information.....53
5.3.2.2 Determining execution order.....53
5.3.3 Node sorting.....55
5.3.3.1 Sorting according to execution level.....56
5.3.3.2 Sorting according to the total CPU cycles of the operation.....56
5.3.3.3 Sorting according to number of supported functional units.....56
5.3.3.4 Handling constant loading.....57
5.3.4 Functional unit allocation.....57
5.3.4.1 Finding start cycle of the execution.....57
5.3.4.2 Allocation.....58

5.3.5 Signal allocation....59
5.4 Implementation....60
5.4.1 Build environment....62
5.5 Chapter summary....62
6 Experimental results....64
6.1 Basic behavior of algorithm mapping....64
6.1.1 Values prepared in registers....64
6.1.1.1 Fast Fourier Transform....64
6.1.1.2 Matrix multiplication....71
6.1.2 Values stored in memory....74
6.1.2.1 Fast Fourier Transform....74
6.1.2.2 Matrix multiplication....77
6.2 Optimization impact....79
6.2.1 Node priority....79
6.2.2 Functional unit priority....80
6.3 Comparison to other methods....82
6.4 Chapter summary....82
7 Conclusion....84

List of figures

Figure 1.1: SISD arrangement....2
Figure 1.2: SIMD arrangement....3
Figure 1.3: MISD arrangement....3
Figure 1.4: MIMD arrangement....4
Figure 1.5: CPU (left) and GPU (right) difference....6
Figure 1.6: Superscalar (top) and VLIW (bottom) difference....7
Figure 1.7: Shared memory system....8
Figure 1.8: Distributed memory system....8
Figure 1.9: Hybrid distributed-shared memory system....9
Figure 1.10: Levels of the programming languages....9
Figure 1.11: Example of common subexpression elimination....12
Figure 1.12: Example of copy propagation transform....12
Figure 1.13: Example of constant folding....13
Figure 1.14: Example of useless code elimination....13
Figure 1.15: Example of dead code elimination....13
Figure 1.16: Example of inline expansion....14
Figure 1.17: Example of loop unrolling....14
Figure 3.1: Example of section work sharing....19
Figure 3.2: For-loop parallel execution....19
Figure 3.3: FFT radix-2 with highlighted loop iterations....21
Figure 3.4: Relative speedup of FIR filter....22
Figure 3.5: Relative speedup of DFT....22
Figure 3.6: Relative speedup of FFT....23
Figure 3.7: Hand-written assembly code....26
Figure 3.8: Example of linear assembly code....26
Figure 3.9: Disassembly of the algorithm written in linear assembly....27
Figure 3.10: Disassembly of the FFT algorithm written in C....27
Figure 4.1: Functional unit utilization for the case 1....31
Figure 4.2: Functional unit utilization for the case 2....32
Figure 4.3: Functional unit utilization for the case 3....32
Figure 4.4: Functional unit utilization for the case 4 without data loading/storing....33
Figure 4.5: Functional unit utilization for the case 4 with data loading/storing....33
Figure 4.6: Functional unit utilization for the case 5....34
Figure 4.7: Workplace for the measuring the power consumption....35
Figure 4.8: Power consumption of theoretical test cases at data path A....35
Figure 4.9: Power consumption of theoretical test cases at data paths A and B....36
Figure 4.10: Power consumption of FFT routines at data paths A and B....36
Figure 5.1: Structure of the TMS320C6678....39

Figure 5.2: Basic structure of the JSON architecture file....40
Figure 5.3: Structure of data path in JSON file....41
Figure 5.4: Structure of cross-path in JSON file....42
Figure 5.5: Example of the cross-path connection to the functional units....43
Figure 5.6: Creating register groups from the physical registers....44
Figure 5.7: Structure of instruction in JSON file....45
Figure 5.8: Execution progress of ADDDP instruction....45
Figure 5.9: Signal-flow diagram from example algorithm....48
Figure 5.10: Mapping process....49
Figure 5.11: Signal definition format....49
Figure 5.12: Arithmetic operation format....51
Figure 5.13: Function definition format....51
Figure 5.14: Constant definition format....52
Figure 5.15: Signal alias definition format....52
Figure 5.16: Memory operation format....52
Figure 5.17: Determining execution level using input signals....54
Figure 5.18: Determining execution level using previous nodes....55
Figure 5.19: Determining execution level of constant loading....55
Figure 5.20: Instruction execution order based on CPU cycles....56
Figure 5.21: Instruction execution order based on number of supported functional units...57
Figure 5.22: Determining first possible CPU cycle for execution....58
Figure 5.23: Determining signal lifetime....59
Figure 5.24: Mapping tool structure....60
Figure 5.25: Architecture editor running under Linux and Windows system....62
Figure 6.1: 4-point FFT algorithm....65
Figure 6.2: Part of signal definition in the 4-point FFT implementation....65
Figure 6.3: Source code of the 4-point FFT (without signal definition)....65
Figure 6.4: Graphical representation of the 4-point FFT....66
Figure 6.5: Generated source code for the 4-point FFT with fixed-point representation....66
Figure 6.6: Functional unit usage in FFT4 (32-bit integer)....67
Figure 6.7: Assignment of signals in FFT4 (32-bit integer)....67
Figure 6.8: Functional unit usage in FFT4 (single precision floating-point)....68
Figure 6.9: Assignment of signals in FFT4 (single precision floating-point)....68
Figure 6.10: Resource utilization of the 4-point FFT....69
Figure 6.11: Graphical representation of 8-point FFT....70
Figure 6.12: Usage of functional units in FFT8 (32-bit integer)....70
Figure 6.13: Assignment of signals in FFT8 (32-bit integer)....71
Figure 6.14: Resource utilization of the 8-point FFT....71
Figure 6.15: Graphical representation of matrix multiplication 2x2....72
Figure 6.16: Resource utilization for the matrix 2x2 multiplication....72

Figure 6.17: Graphical representation of matrix multiplication 3x3.....73
Figure 6.18: Resource utilization for the matrix 3x3 multiplication.....73
Figure 6.19: Difference of the input/output definition.....75
Figure 6.20: Graphical representation of the 4-point FFT with memory operations.....75
Figure 6.21: Functional unit usage in FFT4 (32-bit integer, data in memory).....76
Figure 6.22: Resource utilization of the 4-point FFT with memory operations.....77
Figure 6.23: Graphical representation of the 2x2 matrix multiplication (data in memory). 77
Figure 6.24: Resource utilization for the matrix 2x2 multiplication (data in memory).....79

List of tables

Table 3.1: Basic parameters of the TMS320C6678.....17
Table 3.2: Measured reference time.....21
Table 3.3: Time needed to create parallel region.....23
Table 3.4: C implementation FFT performance.....24
Table 3.5: Low-level implementation FFT performance.....24
Table 3.6: Relative speedup of the low-level FFT implementation.....25
Table 3.7: Performance comparison of the different approach of the C libraries for FFT…28
Table 5.1: Arithmetic instruction supported operations.....46
Table 5.2: Arithmetic instruction supported data types.....46
Table 5.3: Memory instruction supported operations.....47
Table 5.4: Signal definition roles.....50
Table 5.5: Signal definition data types.....50
Table 5.6: Operators for arithmetic operations.....51
Table 6.1: Average hardware resources usage on selected algorithms.....74
Table 6.2: Average hardware resources usage on selected algorithms (data in memory)….78
Table 6.3: Node priority mapping improvements (data in memory).....80
Table 6.4: Functional unit priority mapping improvements (data in memory).....81
Table 6.5: Comparison of tool results with the standard methods.....82

Introduction

The signal processing is the field of electrical engineering which is used for acquiring, modifying and evaluating signals using mathematics operations. In these days, it is used practically in every type of applications around us, such as multimedia, communication, medicine or industrial control. In the beginnings of the electronics, the signal processing was performed only with analogue circuits such as active or passive filters, additive mixers, integrators, derivators, voltage-controlled oscillators, phase-locked loops and so on. These circuits were able to provide enough resources to implement such complex systems like radars and television broadcasting.

Later in 1960s, the digital signal processing became the next field of electrical engineering and computer science. It was caused by availability of required hardware components. But this did not lead to the massive deployment of the applications, because the price of computers was quite limiting. The digital signal processing was used mainly in military, medical and research applications. In the 2000s, the hardware became inexpensive, so the digital signal processing replaced analogue circuits in the applications of everyday life.

Digital signal processing is the application of mathematics operations on discrete quantized signal. The algorithms can be implemented in general computer, digital signal processors or on specialized hardware based on field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). The system parameters are highly dependent on application purpose. The main advantages of the digital signal processing on programmable circuits over its analog equivalent are high accuracy, cheaper implementation of complex algorithms, wide offer of interfaces for data recording and its easy modification without touching the electrical connection. The last advantage leads to the software which is one of the key aspects of the final performance.

This dissertation thesis is focused on software part of the digital signal processing applications, especially on parallel architectures. The result will be a tool, that help to optimize the software with generated parts in the assembly language. The first part of thesis shows the overview of the architectures that can be used on data processing and methods of the programming. The second part demonstrates the behavior of various methods of creating software, especially on multicore very long instruction word (VLIW) processor, and its impact on the application performance. The last part introduces the tool for instruction mapping suitable for creating cores of digital signal processing algorithm cores.

1 State of the art

There are many options how to realize digital processing in these days. Every realization is made of the hardware part and the software part. This chapter is dealing with the hardware resources for digital processing and the possibilities of creating the software.

1.1 System classification

One of the most known classifications of the computer architectures is the Flynn's taxonomy [1]. This classification is based on the number of concurrent instructions and data streams. The processors can be divided according to Flynn's taxonomy into the following groups:

- Single instruction, single data (SISD)
- Single instruction, multiple data (SIMD)
- Multiple instructions, single data (MISD)
- Multiple instructions, multiple data (MIMD)

1.1.1 Single instruction, single data (SISD)

The first group of the Flynn's taxonomy is SISD. Systems belonging to this group are the simplest. They can process only one instruction in one instruction cycle. They also are not able to process multiple data at once, so there is no parallelism (Figure 1.1) [2]. This group might include classic scalar architectures such as complex instruction set computers (CISC) [3] or reduced instruction set computers (RISC) [4]. The advantage is the simplicity of implementation, which requires only one functional unit (FU), and low requirements in software design.

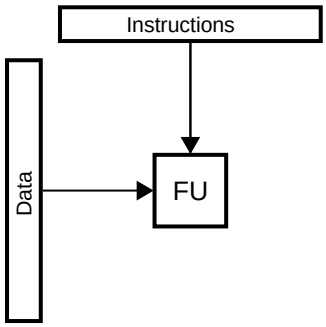

Figure 1.1: SISD arrangement

1.1.2 Single instruction, multiple data (SIMD)

The next group of the Flynn's taxonomy is SIMD. These systems are able to handle larger amount of data with a single instruction (Figure 1.2) [2]. Vector and matrix operations are typical for this group, so the processors are sometimes called the vector processors. The example could be a processor based on the x86 architecture, which is SISD, but extended with the MMX instruction set [5]. The special subset if formed by graphics processing units (GPU). They are used in the homogeneous processing of large amount of data. The disadvantage is that the classic high-level programming languages, such as ANSI C, are not able to utilize the full potential. For this reason, the optimized libraries, special macros or the unusual programming languages are used.

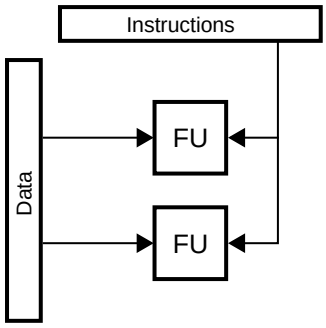

Figure 1.2: SIMD arrangement

1.1.3 Multiple instructions, single data (MISD)

The systems from the MISD group are quite unusual. They are commonly used in special fault-tolerant applications. Data are processed on independent functional units and the results are compared. Data and instruction streams are shown in Figure 1.3 [2]. It reduces the chance of the errors. Except this feature, it provides no benefit like the increase of the computing power. Specific example from MISD group is IBM System/88 [6].

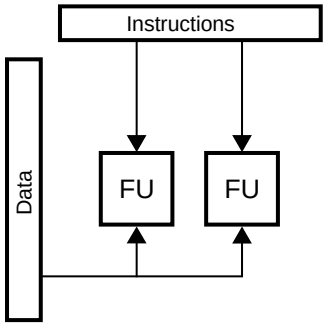

Figure 1.3: MISD arrangement

1.1.4 Multiple instructions, multiple data (MIMD)

MIMD systems use several mutually independent functional units, which can handle different data (Figure 1.4) [2]. In practice, the majority of systems are made of multi-core processors with shared or distributed memory. In this case, every processing unit has its own thread, which is not dependent on the others. It offers flexibility in the parallel processing of the data. This category also includes processors based on very long instruction word (VLIW). Core of the VLIW architecture consist of the multiple functional units, so it can execute multiple instructions in one instruction cycle.

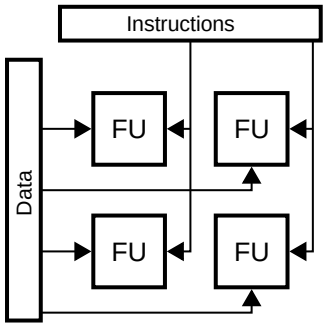

Figure 1.4: MIMD arrangement

1.2 Individual cases of processor architectures

Some specific processor and computer architectures were mentioned in the description of Flynn's taxonomy, which can be used for the digital processing. The next text deals with these architectures.

1.2.1 Scalar central processing units (CPU) and digital signal processors (DSP)

Scalar processors have been used since the birth of the first computers until now. The program is executed sequentially in the order of instructions in the memory. The only options how to change its execution order are the branch instructions or the instructions for calling the subroutines. Over the time, there were made various requirements during its development. This has to led to expanding of the instruction set and thus to the increasing of the arithmetic logic unit (ALU). After some time, it was found that most of the applications can be created with use of only a small number of instructions with comparable performance relative to the original solution. This gave the opportunity to create the RISC. Thanks to the reduced instruction set, the ALU could be smaller, the execution of instructions was faster, and the compilers could be better optimized [7].

RISC [4] is characterized by the following properties:

- a large register file,

- emphasis on operations that use registers,
- instructions are executed in one instruction cycle,
- simple instructions for memory access,
- simplified addressing modes,
- uniform length of the instruction word
- and others.

For comparison CISC [3] characterized with:

- a small register file,
- a large number of instructions,
- instructions oriented for memory access,
- a non-uniform length of instruction word,
- a different time of instruction execution.

Classic processors CISC and RISC are adjusted mainly for control applications. Average application of this type performs branch operation on every 7^{th} instruction [8]. In addition, branches are often unpredictable. Digital signal processing algorithms are different. They are characterized mainly by regular running in loops and periodic memory access. Digital signal processing applications also includes many algebraic operations. Typical operation is a multiply and accumulate (MAC), fused multiply-add (FMA), vector operations or saturated arithmetic [9][10][11]. For this reason, digital signal processors (DSP) were created. Their architecture is similar to the RISC processors in some ways. The first step was to implement the previously mentioned MAC function. Then the idea of separate buses was taken from Harvard architecture. In this case, the memory has not been divided into program and data parts, but the buses are used to read instruction and all operands in one instruction cycle, what increases throughput [12].

Nowadays the typical representative of the CISC architectures are IA-32 (known as Intel x86) and AMD64 (IA-64, x86-64) [13][14][15], which are currently used on most of the personal computers. There are also processors derived from the 8-bit Intel MCS-51 core, which are used in embedded devices. The example of the derivate is 8051 [16]. The RISC processors are more common in embedded devices. They are contained in wide spectrum of the variants from the 8-bit microcontrollers such as PIC [17] or AVR [18], through the 16-bit mixed signal microcontrollers like MSP430 [19] or PIC24, to the 32 and 64-bit processors with the ARM core [20]. The DSPs are also available from the lower performance variants like dsPIC [21] or C2000 [22] to the high-performance processor cores like C6000 [23] or StarCore [24].

1.2.2 Graphics processing units (GPU)

Classic CPUs are oriented to the complex controlling of application and data processing in one thread, sometimes with use of cache memory. GPUs are oriented to parallel data processing with high throughput. It is achieved with the high number of computing cores [25]. One GPU can contain hundreds of them. This number is achieved at the cost of their simplicity, so they are not suitable for control applications. GPUs are therefore used in combination with CPUs as the coprocessor [26]. The difference between CPU and GPU is shown in Figure 1.5 [27].

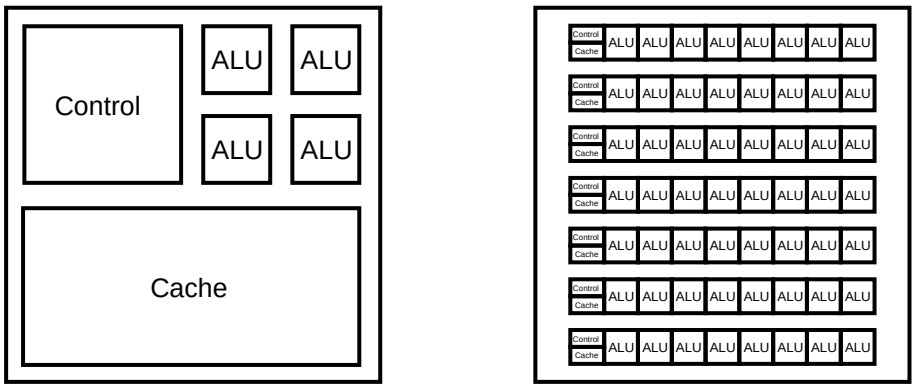

Figure 1.5: CPU (left) and GPU (right) difference

From the graphical comparison of the CPU and GPU can be seen, that the big part of the classic CPU is made of the memory, which can be used for data and instructions. Also due to its complexity, the control logic and ALUs needs more logic elements. On the other side is the GPU with the minimal cache memory or control logic. The biggest part is created by the ALUs, so it makes GPUs suitable for data processing with high throughput as it was mentioned, not for control application.

First video cards stared with the IBM Monochrome Display Adapter in 1981 with only text support. Later, the video cards supported 2D and 3D graphic acceleration [28]. In 1999 Nvidia introduced the first GPU for the personal computer (PC) industry with the definition that a GPU is “a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second” [29]. ATi introduced Radeon R100 as Nvidias rival and later with R300, ATi used term Visual Processing Unit (VPU) [30]. Nowadays, the GPUs are not used only for graphic processing, but there are also models dedicated for high performance computing (HPS), such as Nvidia Tesla [31]. These cards are also known as general-purpose graphics processing units (GPGPU).

1.2.3 Very long instruction word (VLIW)

Core of the processor based on VLIW [32][33] architecture contains multiple functional units with ability to execute multiple instructions at once. It is the instruction-level

parallelism like in the superscalar processors, but with one difference. Superscalar processor maps the instruction dynamically from the stream of the single instructions (Figure 1.6) [34]. Software for VLIW is made of instruction packets, which are created statically during the software compilation. Thanks to this, the VLIW core structure can be simplified. This makes the space for the additional functional units, its functionality or the increase of the clock frequency. The VLIW processors usually find its place in signal processing or multimedia applications. The instruction-level parallelism is used mainly in the implementation of DSP algorithm cores.

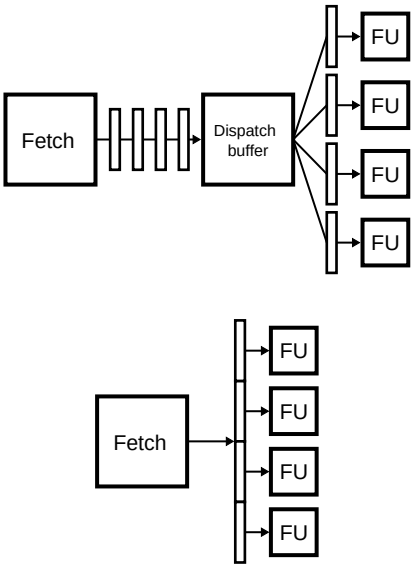

Figure 1.6: Superscalar (top) and VLIW (bottom) difference

1.2.4 Multicore systems with shared memory

Multicore systems with shared memory contain several independent CPUs with direct access to the local memory, which is usually RAM (Figure 1.7). This model could be applied to various architectures such as CISC, RISC, DSP or their combination, so the system could be homogeneous or heterogeneous. The most known systems from this group are multicore PCs, but they are also used in embedded systems for medical systems, radar systems etc. The parallelism is created through threads. During the processing, the input signal is divided into several parts, which are processed separately. The iterations must be independent on each other, so not all algorithms can be parallelized in this way.

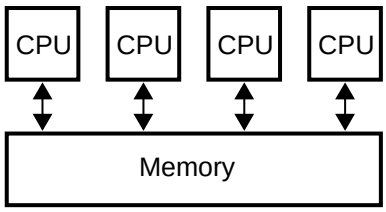

Figure 1.7: Shared memory system

The advantage of these systems is the fast data sharing between the tasks. Also, the single address space provides simple perspective on memory during the software developing. The main disadvantage is the scalability of the systems. With the increase of the number of processors also increases the traffic on the memory to CPUs bus. The next disadvantage is connected with the previous one. The software developer should be aware about the correct access to the memory [35].

1.2.5 Multicore systems with distributed memory

Multicore systems with distributed memory are similar to the systems with shared memory from the parallelism principle point of view. The difference is that every processor has its own address space. When access to the different memory space is needed, data are transmitted in the message through the communication network (Figure 1.8). These systems are used in the HPC typically for simulation of the physical effects such as fluid flow or electromagnetic fields with very detailed models.

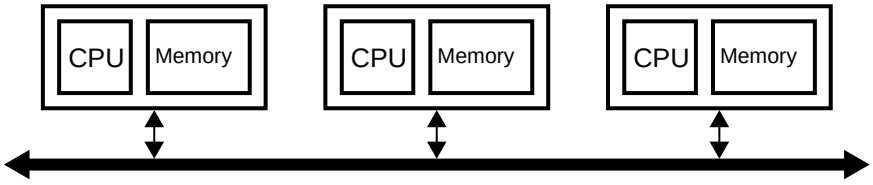

Figure 1.8: Distributed memory system

The advantage of this systems is the scalability. With the increasing the number of processors the memory also expands. In addition, each processor can access to its own memory without interference of the other CPUs. The disadvantage is the non-uniform data access, because the data can be placed in different node. This also makes difficult to work with the global data, because the software needs to process the data exchange between the nodes [35].

1.2.6 Multicore systems with hybrid distributed-shared memory

These systems combine previously mentioned systems. The shared memory systems with multiple CPUs or GPUs with its own memory space are interconnected with

network like system with distributed memory (Figure 1.9). These systems can be scaled to the desired application respecting the advantages and disadvantages of the combined systems.

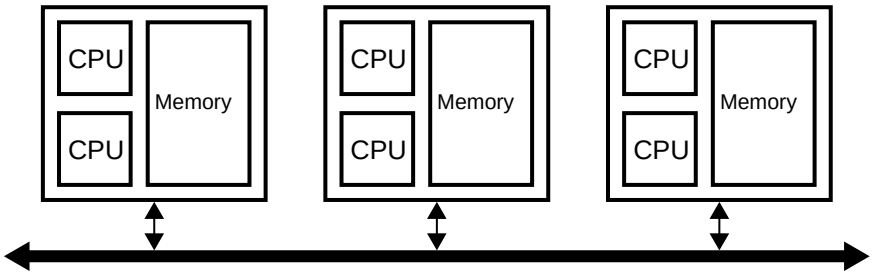

Figure 1.9: Hybrid distributed-shared memory system

1.3 Programming methods

The performance of the final application is not only dependent on the device, but also on the software. It is really important part of the application, because the well optimized code could make better performance on the low-cost hardware than the bad written code running on the high-priced device. There are several methods of creating the final code which has its pros and cons (Figure 1.10). This subsection will introduce some methods of creating software.

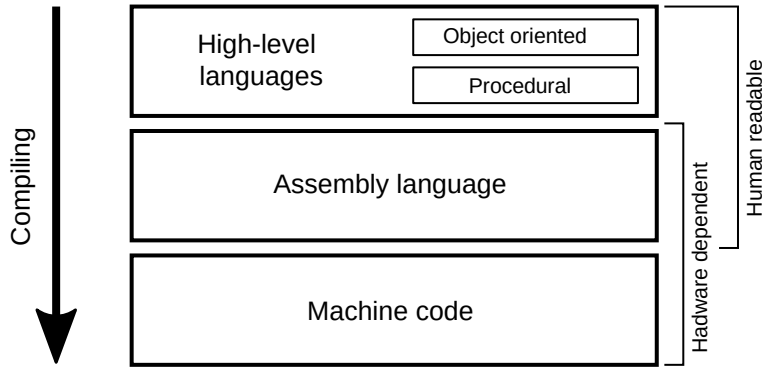

Figure 1.10: Levels of the programming languages

1.3.1 Low-level languages

The low-level programming languages provide only little abstraction from processor instruction set. Low-level code could be converted directly to the machine code without using a compiler. The software written in low-level language could be really fast and the result binary code could be small. This kind of programming was common in the past because of lack of high-level language compilers, but nowadays is used only for:

- embedded systems with small resources

- optimizing of the critical part of the software
- creating hardware drivers and system code

The next reason, why it is not used, is the economical aspect. The software development takes a long time and the code is highly dependent on the processor architecture and instruction set, so it is not easy portable between different devices [36] [37].

There are several ways how to write a low-level code:

- machine code
- low-level assembly
- linear assembly

Writing an application in machine code is unusual and in common practice is not used at all, because it is requires lot of concentration, the code is not human readable, so there could be easy to make mistake that is really hard to find. Instead of this, the assembly language is used. It is the text interpretation of the processor instructions. The difference between low-level assembly and linear assembly is that the code in linear assembly does not include the information about the used registers and functional units. The compiler maps the required resources itself automatically [38].

1.3.2 High-level languages

The high-level languages provide strong abstraction from the hardware. Instead of dealing with the instructions, registers and memory addressing, the high-level languages deal with the variables and arithmetic expressions. The code is better readable than the assembly code. Thanks to the strong abstraction, it is also easy portable. High-level languages include for example the FORTRAN [39], BASIC [40][41], C [42], C++ [43], C# [44] or Java [45]. After the compilation, some of them could be executed directly on the machine, but some of them needs interpreter. The price for possibility to easy write complex code, which is also portable, is a smaller efficiency and the larger size of the final binary program. This is caused by the inability of the direct translation of the elements into the machine code. Even if the compilers are still being developed to generate more optimized code [46], they are not able to handle some special cases. The following examples refer to the standard C/C++ expressions:

- inability to express special DSP operation such as addition, subtraction and multiplication with saturation
- inability to express vector operations
- inability to mark the independent part of programs which can be run in parallel due to sequential character of notation

- inability to process data on parallel functional units/cores (split iterations of loops)

These deficiencies are removed using the special optimized libraries provided by processor manufacturers [47][48][49] or by the third party [50], compiler extensions, such OpenMP [51] for program execution on shared memory system or MPI [52] for distributed memory system or with special programming languages like CUDA [53] for general-purpose processing on GPU. There are also some projects such as [54] that are able to handle the instruction level parallelism more effective.

1.3.2.1 Difference between compiled and interpreted languages

The typical approach of the translating source code of the program written in high-level languages such C/C++ is compiling it to the machine code. This result can be decoded directly by the compatible hardware and the CPU can execute the instructions. In some cases, it is not necessary to translate source code directly to the machine code [55]. These languages can be divided into:

- pure interpretation,
- hybrid implementation systems.

The pure interpreted languages are also called scripting languages. These languages need the interpreter, i.e. a software for fetching the high-level language statements and parsing it into the operations at runtime. The main disadvantage of this system is the slowness of the execution.

This disadvantage is reduced with the hybrid implementation system, where the source code is compiled into the intermediate code, sometimes called byte code. This code contains only instruction for the virtual machine, which only translate the byte code into the machine code [56].

1.4 Standard optimization methods

Optimizations are set of analyze and transform operations performed on source code achieving to run it faster or consume less hardware resources. These operations finds and replaces parts of code with more efficient alternatives. The compilers use two main techniques to determine the code parts to optimize [57]:

- control flow analysis
- data flow analysis

Control flow analysis is based on the examination of the control statements which can cause branch in the program such as loops, conditions and function calls. In this case, the optimizations are applied on the possible paths of program execution.

Data flow analysis is another type of optimization, which analyzes the usage of data in the program. This can be used for reducing number of variables, optimize loading of constants and data transfer.

Several optimization techniques are described in [46] and [57]. Following text will shortly introduce some of these common methods.

1.4.1 Redundancy elimination

The code can be marked as redundant when the same expression has been previously evaluated without modification of its variables [46]. The redundancy elimination includes common subexpression elimination and copy propagation transformation [57].

The common subexpression elimination reduces number of executed instructions by removing expressions which were already computed. The result value is used instead of the evaluate expression again (Figure 1.11).

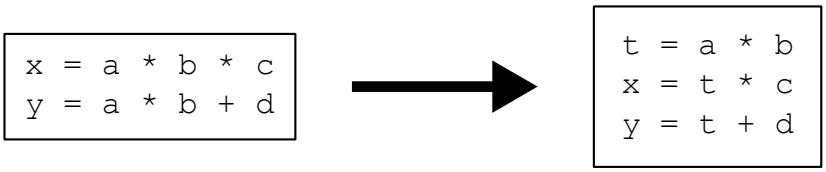

Figure 1.11: Example of common subexpression elimination

The copy propagation transform reduces cases when variables are copied from one to another. Instead of copying variables and accessing to target and source location, the source variable is used in next expressions (Figure 1.12).

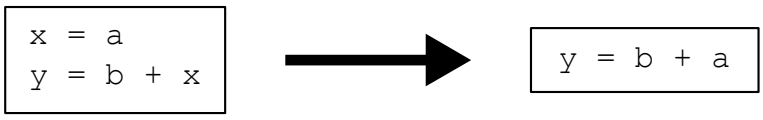

Figure 1.12: Example of copy propagation transform

1.4.2 Constant propagation optimization

Constant propagation optimization [46] also known as constant folding [57] tracks the known variable values propagation in the call graph. In cases where the value of the expression can be determined at compile time, this expression is substituted with the evaluated constant (Figure 1.13).

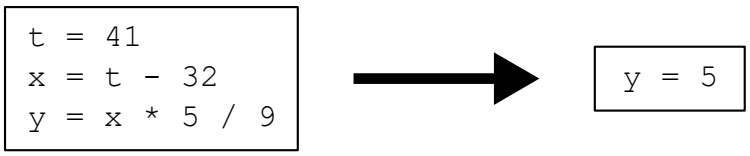

Figure 1.13: Example of constant folding

1.4.3 Useless code elimination

Some parts of code can have no effect to the program results. These parts of code can be formed by assignment operations to unused variables (Figure 1.14) and unrealizable conditions (Figure 1.15). These operations and parts of code can be removed without program functionality affection. Special case of this optimization type is also known as dead code elimination [46][57].

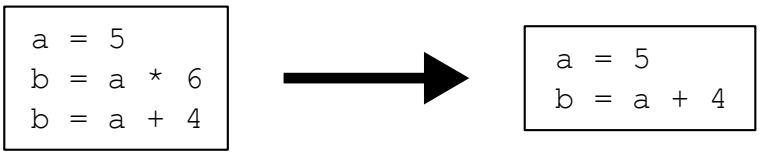

Figure 1.14: Example of useless code elimination

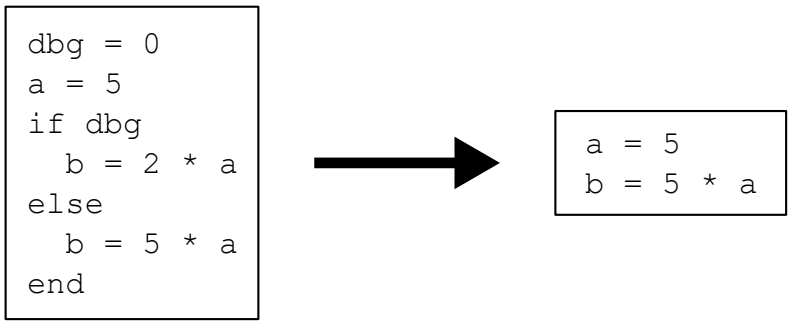

Figure 1.15: Example of dead code elimination

1.4.4 Inline expansion

Inline expansion, or inlining, is used to increase the code performance [46][57]. It replaces complex parts of code with its inline representation. Typical optimization is the function inlining where function call is replaced directly by function content (Figure1.16). Inlining also includes loop unrolling, where the loop content is replaced by the series of operations from loop (Figure 1.17).

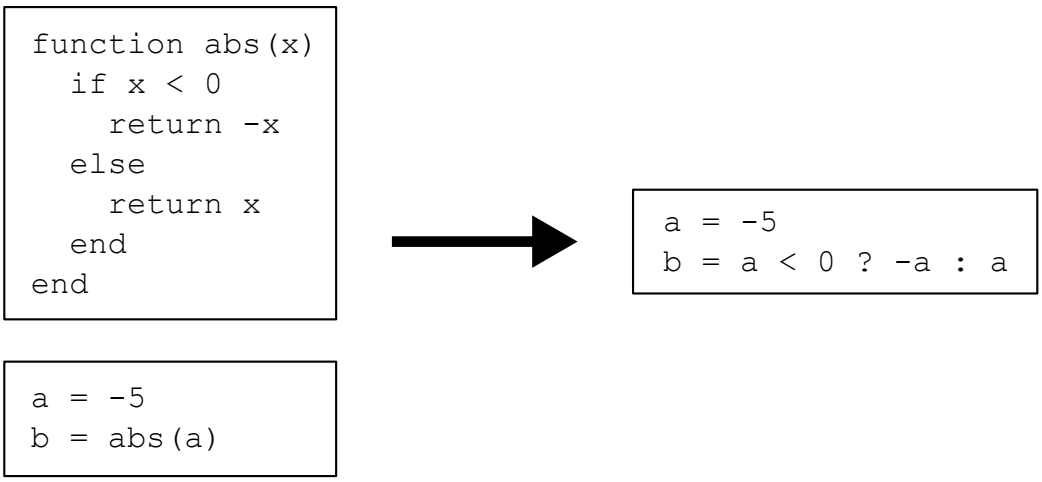

Figure 1.16: Example of inline expansion

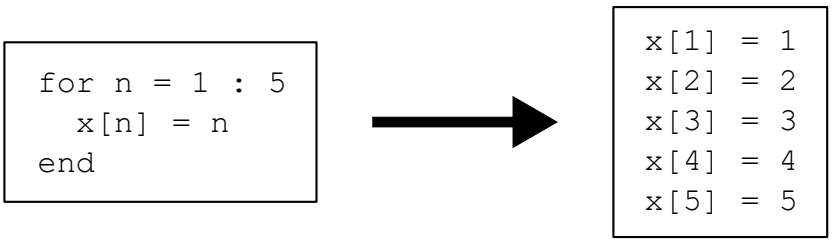

Figure 1.17: Example of loop unrolling

2 The objectives of the dissertation thesis

The previous chapter described possible realization options of digital signal processing applications. It does not matter if the signal processing is performed on the scalar processor or the multicore system, the software is still the most critical part that specifies the final efficiency. The modern compilers could produce quite effective code, usually on scalar architectures, because these compilers were developed for a long time and they are frequently used. But there are other architectures which are not commonly used in applications and they are using some enhanced type of parallelism, not only pipelining, so the compilers could be less effective. The VLIW architecture meets this condition, because its instruction parallelism must be specified at compile time.

For this reason, the dissertation thesis will be focused on the software part of the signal processing systems, mainly the parallelism. The objectives are as follows:

- Prove that the software development tools for instruction-level parallelism are less effective than the tool for data parallelism or task parallelism.
- Create the effective tool for the software developing of digital signal processing application suitable for architectures using instruction-level parallelism, especially VLIW processors.

The second objective consists of the followed points:

- Create the general model of VLIW processor or any general-purpose processor which will be used by the tool to final assembly code.
- Create an algorithm for DSP algorithm assignment to the available hardware resources.
- Implement an optimization method to effective mapping of the functional units and registers.

3 Effectiveness of software development tools

The software plays the key role in the whole signal processing system based on DSP. This chapter will show the effectiveness of the widely used programming approaches focused on parallelism. The dissertation thesis is aimed on the instruction parallelism when the software execution is determined at compilation time. Also, the instruction level parallelism should be compared with the data parallelism. For that reason, the multicore VLIW based DSP will be used in the next benchmarks.

This chapter will demonstrate the programming methods of signal processing applications from higher-level to low-level. The high-level approach will include data processing in multiple threads to show the suitability on computations in different areas. The next high-level approach will be pure single threaded execution of the algorithms to be compared with the low-level approach when VLIW architecture is used. This high-level case will be compared with the low-level assembly language and linear assembly language, which is not available for all architectures.

There are not so many silicon manufacturers producing VLIW DSPs which meets the requirements and are also easily available. Texas Instruments (TI) offers DSPs from C6000 family, which are based on VLIW architecture and they are also made in multicore variants. There ale also multiple development kits based on these DSPs. The most of them are with the C64x [58] cores, which is older series supporting only fixed-point arithmetic, and with the C66x [59] cores with floating-point support. The choose will be decided from the newer C66x, because it will show also the handling of the floating-point arithmetic. From the availability of the evaluation boards, the TMS320C6678 [60] was chosen. This DSP fits perfectly, because it is multicore fixed-point VLIW based DSP allowing wide demonstration cases in fields of instruction-level and threading parallelism. The processor and the development board will be described in detail later in this chapter.

The first part of chapter describes the structure of used processor, its features and properties, and the used development board as well. The second part is evaluating the DSP algorithms created with the high-level and low-level languages in instruction-level parallelism point of view. The high-level language also demonstrates the thread level parallelism using OpenMP.

3.1 Multicore DSP TMS320C6678

The TMS320C6678 is a multicore fixed/floating-point digital signal processor and it is containing of eight C66x DSP cores [59]. Each core consists of two data paths (titled A and B), two sets of thirty-two 32-bit registers (A0, ..., A31 in data path A and B0, ...,

B31 in data path B, respectively), and two sets of four functional units (.L1, .S1, .M1, .D1 in data path A and .L2, .S2, .M2, .D2 in data path B). Each functional unit is primary used for a different type of operations. The .Dx units are used for loading/storing data between a general-purpose register file and a memory space. The .Lx and .Sx functional units perform general fixed and floating-point arithmetic operations, next the logic operations, and finally the branch functions. The .Mx units perform all multiply operations with the single/double precision floating-point numbers as well as with the fixed-point values. In addition, the DSP is capable to execute SIMD instructions for fixed-point and floating-point instructions, where 8 and 16-bit operands are packed into the single 32-bit word, or single precision floating-point values are packed into the register pairs. These SIMD instructions are especially for additions and multiplications (`DADD2`, `MPY2`, `DADDSP`, `DMPYSP`, `QMPYSP`, etc.) [23]. This is useful for the signal processing algorithms such as Fast Fourier Transform, Discrete Cosine Transform, etc. The DSP can also perform complex multiplication or multiplication of complex vectors by the complex matrices. Detailed description of the DSP functionality can be found in [60]. The basic parameters of TMS320C6678 DSP are shown in Table 3.1.

Table 3.1: Basic parameters of the TMS320C6678

Parameter	**Value**
Clock speed	1.4 GHz
L1P memory	32 kB/Core
L1D memory	32 kB/Core
L2 memory	512 kB/Core
Shared L2 memory	4 MB
External memory interface	64-bit DDR3
GFLOPS	128
Thermal design power	17 W

3.1.1 TMDSEVM6678LE Development Board

The 8-core DSP is assembled on a development board TMDSEVM6678LE [61]. It is a stand-alone development board with 512 MB of DDR3 memory, 64 MB of NAND Flash, 16 MB SPI NOR Flash, Gigabit Ethernet, PCIe, and other typical peripherals. Some of them are routed to the AMC B+ edge connector. It makes the board ideal for developing of media gateways, and/or video servers of video recognition applications. The board also contains an embedded JTAG emulator, so it can be connected with TI's software development tool: Code Composer Studio without need of any external emulator. Nevertheless, it is possible to connect a different emulator through a 60-pin TI

JTAG connector. In this case, the XDS560v2 is providing the real-time debugging. The disadvantage of the board is the lack of accessible testing points, so the power consumption can be measured for the whole board and not for the components separately.

3.2 Test cases

Testing of the software behavior is divided into 2 groups. The first group explores the performance of the code from the data and thread parallelism, the second examines the performance from the instruction level parallelism. All of the evaluations were performed on the real hardware which was previously described.

3.2.1 Data and thread parallelism using OpenMP

The data parallelism is achieved when the multiple functional units (or the cores) perform the same operation on different data. It could be realized on the SIMD or multiprocessor system. For comparison, the thread parallelism can be achieved only on the multiprocessor system. One of the solutions how to make code to run on multicore processor is to use OpenMP [51].

OpenMP is an application program interface (API), which provides a portable, scalable model for shared-memory programming. First specification of OpenMP was defined in 1997 for FORTRAN by major hardware and software vendors. One year later OpenMP was defined for C/C++.

OpenMP uses thread based parallelism with fork-join model. This means, that application start in one thread and if it come to parallel section, it creates another thread. When this team of threads completes their work, they synchronize and terminate except master thread. These threads can be section work-sharing and loop work-sharing [62].

3.2.1.1 Section work-sharing

This type of work-sharing can be used for independent pieces of code which can run in parallel. Parallelism of this type is similar to creating threads through standard libraries provided by operating system. It can be used to pipeline the processing.

Figure 3.1 shows the example of section work-sharing. The original algorithm consists of 4 steps and is performed sequentially. Steps 2 and 3 are independent and can be performed in different order or in parallel.

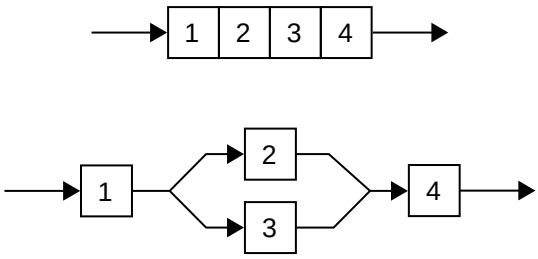

Figure 3.1: Example of section work sharing

3.2.1.2 Loop work-sharing

The loop work-sharing is the common way how to increase the performance of the application. For-loops are primary targets in parallelization. They can be used if iterations have no dependencies between each other.

Figure 3.2 shows a for-loop parallel execution, which is processing an array with length of 16. It is divided into 4 threads, where each of them processes 4 values. Parts of array which are processed are marked in gray color.

Figure 3.2: For-loop parallel execution

In this case the processed data are shared among all cores. If the algorithm uses some auxiliary variables and they are used by all threads, the code will possibly return wrong result. These variables must be defined as private, which means that there is created local copy in memory for every core.

3.2.2 Algorithm parallelization in OpenMP

This part is dealing with a parallelization of selected signal processing algorithms. It is especially finite impulse response (FIR) filter, discrete Fourier transform (DFT) and Fast Fourier transform (FFT). These algorithms allow easy parallelization on the loop. Each of them has different character comparing the others. The FIR filter process relatively small part of input signal to generate one output sample. The DFT needs the whole input signal for each output sample. These two algorithms process the output signal directly sample by sample, but the FFT needs to compute intermediate data from all input samples and then the final signal. Parallelization of code is realized with OpenMP directives. During this process, it is important to take care of which variable is shared between threads and which must be created as private for each thread.

3.2.2.1 FIR filter

FIR filter is implemented according to

$$y_n = \sum_{k=0}^{N-1} x_{n-k} h_k \tag{3.1}$$

from [63], where x is input signal, y is the output signal and h is impulse response if the filter with the length of N.

This type of filter was selected, because it does not require feedback, which could not be simply parallelized. Final code contains 2 nested for-loops, but only outer loop is parallel. However, OpenMP support nested parallelism, inner loop is performed sequentially. It is because the number of physical cores is less than number of signal samples and there is no space where to execute other threads.

3.2.2.2 Discrete Fourier transform

Structure of the DFT implementation (3.2) is similar to the FIR filtration (3.1). The output sample is given by the sum of products of input signal and another variable. It consists of 2 nested for-loops. The difference is that there are complex calculations and the inner loop goes through full length of the signal. This means, that the amount of processed data is much higher in compared to the FIR filter. According to [63], DFT is given by

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \tag{3.2}$$

where x is input signal with length of N in time domain. The X is output signal in frequency domain also with the length of N.

3.2.2.3 Fast Fourier transform

For the demonstration of FFT, the Cooley-Tukey algorithm [64] was chosen. This algorithm is one of the most used in the practical implementations of the signal processing algorithms. The structure is different from the previous implementations. Figure 3.3 schematically shows progress of used loops in algorithm. The outer loop iterations, which represent the stage in FFT, obviously depend on each other, so it cannot be executed in parallel. The middle, representing group of butterflies, and the inner loop, representing butterfly processing, are independent in each of its iteration. For simplicity, only middle loop was chosen for parallelism even if last stages will not benefit from this. The final implemented FFT algorithm is the radix-2 decimated in time (DIT).

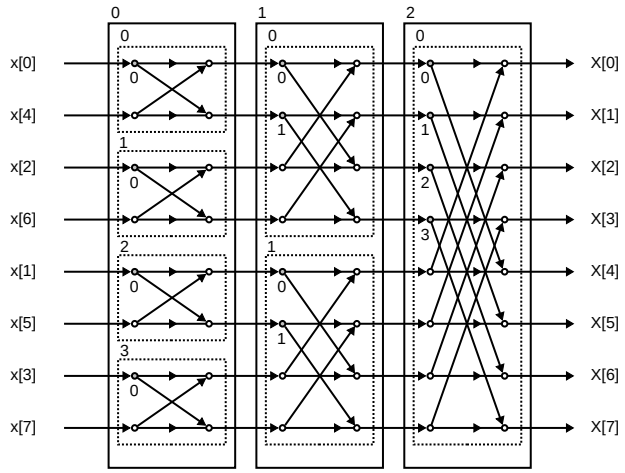

Figure 3.3: FFT radix-2 with highlighted loop iterations

Final parallel code cannot run without operating system, which controls threads. TI provides real-time kernel called SYS/BIOS [65] or DSP/BIOS [66]. It is designed to use in embedded applications which requires real-time scheduling.

3.2.3 Measured performance of OpenMP

The execution time of whole function call represents the performance of implemented algorithms. Dependence of execution time on number of created threads and length of input signal was measured. For determining how the performance of algorithms was influenced with changing of these parameters and by the OpenMP runtime, the execution time of sequential versions (without OpenMP pragmas) of algorithms was chosen as reference (Table 3.2).

Table 3.2: Measured reference time

Length of the signal	FIR	DFT	FFT
16	2 µs	157 µs	16 µs
32	4 µs	605 µs	39 µs
64	8 µs	2457 µs	93 µs
128	16 µs	9911 µs	215 µs
256	32 µs	39832 µs	491 µs
512	63 µs	159782 µs	1105 µs
1024	126 µs	640102 µs	2456 µs
2048	252 µs	-	5405 µs
4096	507 µs	-	11810 µs
8192	1025 µs	-	25791 µs

Figures 3.4 to 3.6 show the relative increase of performance. The *X* axis represents number of cores processing the signal, the *Y* axis carries the length of the processed signal and the *Z* axis shows the speedup relative to the reference time from Table 3.2. From graphs can be seen, that performance of all algorithms with OpenMP directives are slower when there is only master thread. It is because the process of thread creating is still active, even if the maximum number of threads is set to 1. It is the same reason why the relative speedup is not the same as the number of created threads. In addition, threads are communicating with each other and accessing to the same memory, because inputs and outputs are defined as shared variables.

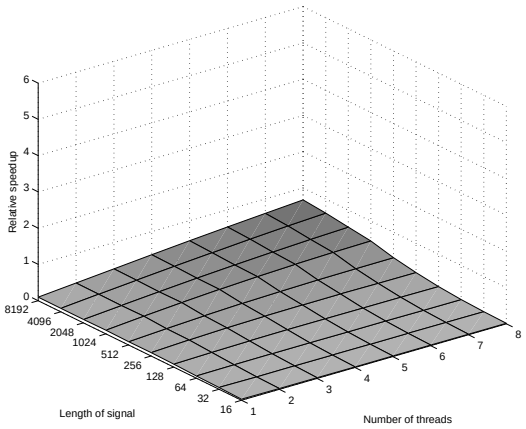

Figure 3.4: Relative speedup of FIR filter

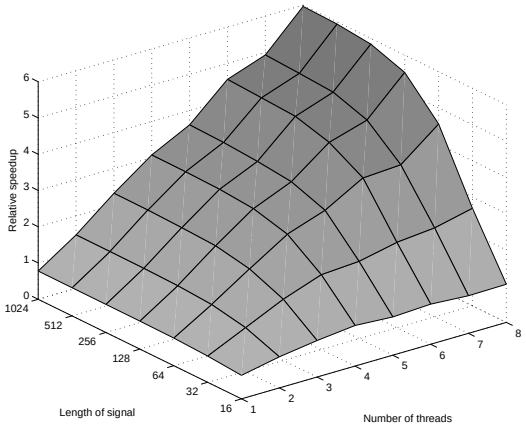

Figure 3.5: Relative speedup of DFT

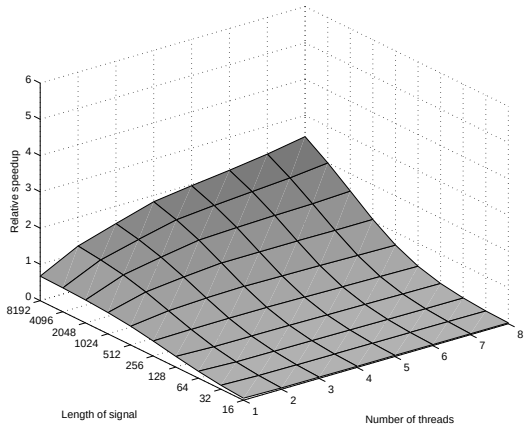

Figure 3.6: Relative speedup of FFT

Table 3.3 shows the measured times that are needed to create the new threads. On FIR filter and DFT algorithm, it is created only once. When program compute FFT, the parallel region is created regularly depended on length of input array.

Table 3.3: Time needed to create parallel region

Number of threads	**Time**
1	17 µs
2	34 µs
3	36 µs
4	39 µs
5	42 µs
6	45 µs
7	48 µs
8	52 µs

If the processing is made of the small number of instructions or the length of processed data is short, it does not worth it to parallelize the loops. It is because the time required for creating threads and time while these threads communicate with each other can be approximately the same or bigger than the execution time of the actual time of calculation. In addition, the behavior of the hyper-thread enabled processor could be found in [67]. This makes threading parallelism suitable to apply on processed data with the same algorithm core, not for its creation. The algorithm core creation should be performed by optimization on the low-level, which will be shown in next part of this chapter.

3.2.4 Low-level optimizations of the algorithms on the VLIW architecture

The low-level programming approach allows the programmer to utilize the functional units of the VLIW processor as much as possible. For the next examination, the FFT was chosen again. Now, the algorithm is not written to work in loops with variable-length input signal, but it is written to process fixed vectors with 4, 8 and 16 samples. The function's computing performance was measured in CPU cycles. All measurements were evaluated for a single-core DSP version only.

Table 3.4: C implementation FFT performance

Function	**Input**	**Data path**	**CPU cycles**
FFT4R	4-point real	A+B	46
FFT4C	4-point complex	A+B	80
FFT8R	8-point real	A+B	123
FFT8C	8-point complex	A+B	205
FFT16R	16-point real	A+B	425
FFT16C	16-point complex	A+B	642

Table 3.4 summarizes the computing demands of functions written in C language, where notation FFT4R represents a function for real FFT with $N = 4$, FFT16C is the function for complex FFT with $N = 16$, etc. The code was compiled by commercially available compiler for C6000 Optimizing Compiler v7.3.1 from TI. By exploring the disassembly code, the usage of both DSP data path A and B was affirmed. It can be seen, for a single FFT calculation between 46 CPU cycles (for $N = 4$ real values) and 642 CPU cycles (for $N = 16$ complex values) is needed. The computing performance of functions written in low-level source code is shown in Table 3.5.

Table 3.5: Low-level implementation FFT performance

Function	**Input**	**Data path**	**CPU cycles**
FFT4R	4-point real	A	19
FFT4C	4-point complex	A	24
FFT8R	8-point real	A	34
FFT8C	8-point complex	A	42
FFT16R	16-point real	A	88
FFT16C	16-point complex	A	100

Low-level implementation of the previous functions takes from 19 (for N = 4 real values) to 100 CPU cycles (for N = 16 complex values). The relative speedup (Table 3.6) is from 2.4 (for N = 4 real values) up to 6.4 (for N = 16 complex values). The next improvement is the utilization of only one data path. It means that if there is need to compute multiple transforms in row, the speedup can be twice as it is now achieved only with copying the code into the data path B.

Table 3.6: Relative speedup of the low-level FFT implementation

Function	**Input**	**Relative speedup**
FFT4R	4-point real	2.42
FFT4C	4-point complex	3.33
FFT8R	8-point real	3.62
FFT8C	8-point complex	4.88
FFT16R	16-point real	4.83
FFT16C	16-point complex	6.42

3.2.5 High-level and low-level comparison

Previous parts are exploring the speed of execution of low-level and high-level implementation. Now, the text will show the difference in the structure of the compiled code. It will be shown on the 4-point FFT with complex inputs. The code is based on the FFT4C function from previous demonstration. The low-level code was rewritten into the linear assembly and C language respecting the same order of the operations. The optimizations were disabled for better recognition of the disassembled parts.

3.2.5.1 Low-level assembly

The low-level language offers the most accurate way to optimize the code. The software developer has full control over the processor functionality and timing. It makes this method suitable for creating time critical parts of software, such as the DSP cores. Developing software in the low-level assembly requires more time and the final code can be used only on the specific architecture. For these reasons, the low-level assembly is not used for creating the complex software or the libraries. For example, the FFTW library [50] is fully implemented in C, but some parts have multiple implementations, which include assembly routines as well.

```
    LDDW  .D1 *A4++[2], A17:A16
    LDDW  .D1 *A4--[1], A19:A18
    ADDSP .L1 A6, A8, A6
||  SUBSP .S1 A6, A8, A8
||  LDDW  .D1 *A4++[2], A21:A20
    ADDSP .L1 A7, A9, A7
||  SUBSP .S1 A7, A9, A9
||  LDDW  .D1 *A4++[1], A23:A22
```

Figure 3.7: Hand-written assembly code

The part of low-level implementation of the FFT is shown in Figure 3.7. The first ADDSP (single precision floating-point addition, see [23]) operation is the equivalent of the first addition operation of the C code from Figure 3.10. This part of the code is preceded by the other 4 LDDW (load double word from memory) instructions for loading data into the registers and the loading process continues during the processing of previously loaded data. In this case, the whole computation of the 4-point FFT with complex input takes only 24 instruction cycles for TMS320C6678. The code uses only data path A, but the level of parallelism is much higher than the result of the C compiler.

3.2.5.2 Linear assembly

Linear assembly language is very similar as the classic assembly language, where the developer uses specific instructions, but does not care about timing and usage of functional units and registers. This method is alternative for the TMS320C6000 architecture family DSPs [68]. This feature should help to reduce developing time [69].

The FFT algorithm from the previous case in the linear assembly language contains instructions in the same order as in the low-level assembly code, but the register names were replaced by the symbolic titles. The functional units were removed as well. The part of the linear assembly code is shown in Figure 3.8.

```
ldw     *pX[6], in6
ldw     *pX[7], in7
addsp   in0, in4, m0
addsp   in1, in5, m1
subsp   in0, in4, m2
```

Figure 3.8: Example of linear assembly code

In the disassembly form of the example code (Figure 3.9) can be seen one data path A is used, similar tot the low-level assembly, but the instructions are executed sequential, even if there is a possibility to combine them into one instruction packets. The example is the instructions ADDSP and SUBSP, which use already loaded independent data, but SUBSP waits for the completion of the ADDSP instruction. The

addition and subtraction of two floating-point numbers can be performed by the functional units .L and .S [23]. The arguments of the operations are also different. In addition, the compiler waits for the result with NOP (no operation) instruction before executing the following operation.

```
   LDW.D1T1  *+A4[6],A19
   LDW.D1T1  *+A4[7],A18
   ADDSP.L1  A7,A9,A17
   .fphead   p, l, W, BU, nobr, nosat, 0000011b
   NOP       3
   ADDSP.L1  A6,A8,A16
   NOP       3
   SUBSP.L1  A7,A9,A9
   NOP       3
```

Figure 3.9: Disassembly of the algorithm written in linear assembly

3.2.5.3 High-level language

The high-level programming languages are useful for creating complex software, because it reduces developing time. They are also suitable for creating the libraries for the multiple platforms, because the source code is portable to different architectures.

Tested algorithm is made as separate function in the C language, with one input pointer to signal samples vector. The temporary results are stored into the local variables. The code contains only 16 arithmetic operations and the part of final disassembled code from TIs C6000 compiler v7.3.1 with the default optimization level is shown in Figure 3.10. The disassembled code shows the first operation of the algorithm.

```
fft4_dit_c:
00008340:   07FFEC52          ADDK.S2       -40,B15
00008344:   AC45              STW.D2T1      A4,*B15[1]
18            A6 = pX[0] + pX[4];
00008346:   6246              MV.L1         A4,A3
00008348:   9247     ||       MV.L2X        A4,B4
0000834a:   904D              LDW.D2T2      *B4[4],B4
0000834c:   018C0264 ||       LDW.D1T1      *+A3[0],A3
00008350:   020C979A          FADDSP.L2X      B4,A3,B4
00008354:   2C6E              NOP           2
00008356:   DC45              STW.D2T2      B4,*B15[2]
```

Figure 3.10: Disassembly of the FFT algorithm written in C

The execution of the function takes 195 instruction cycles, including function call and return. There can be seen, that the compiler is using both data paths A and B. It could be a good idea to use all possible resources, but in this cases with similar range it

is not effective because the data transfer between data paths must be realized through the cross-path, which is limited on single value per cycle. The next think to notice is the instruction parallelism. The || sign means, that the instruction is executed at the same time with the instruction above. Here, the code is executed mostly sequentially, one instruction after the other.

The other issue is the frequent access to the memory. The function also does not use access to values through the pointer, but it uses separate variables where the values were copied. In this example the result was similar, because the variables were allocated on the stack.

The function was also rewritten to not using access to values through the pointer, but it uses separate variables where the values were copied. The result was similar, because the variables were allocated on the stack. Other information about usage of the functional units can be found in [70].

3.2.6 Comparison of the libraries with different structure

The method for implementing DSP algorithm should be considered for the application. It is typically compromise between the effort and code portability on one side and the code performance on the other.

Table 3.7: Performance comparison of the different approach of the C libraries for FFT

Size	**Cycles**		
	Non-optimized	**FFTW**	**TI-DspLib**
8	5 909	893	145
16	10 520	2 080	171
32	35 628	4 862	244
64	60 804	15 400	373
128	193 058	33 990	818
256	321 088	77 314	1 483

Table 3.7 shows the performance, given in CPU cycles, of three FFT libraries on the TMS320C6678. The first non-optimized library was implemented only for the testing purposes. Everything is computed during the runtime, including the twiddle factors. The second is the FFTW [50], which was configured for the general C compiler, because it does not have any support of the special instructions for the target DSP processor. The twiddle factors and other parameters are precomputed before the FFT execution. The last one is the TI's DSP library for C6000 [47]. The FFT parameters are also precomputed, but it is optimize using the low-level assembly parts. The disadvantage is

that this code cannot be used on different architectures. The difference of the libraries performance is significant. The optimized FFTW library is about 6.5 times faster than unoptimized library for small vectors and about 4 times faster for larger vectors. The low-level library (TI-DspLib) is about 6.5 times faster than optimized C library for small vector and about 53 times faster for larger vector.

3.3 Chapter summary

This chapter showed the difference between low-level and high-level programming languages. The demonstration was performed on the multicore DSP TMS320C6678. The results are described below.

The high-level programming languages is the fast and easy way how to write DSP algorithms offering the possibility of compile the code on the other platform. But on the VLIW architectures, it is not very effective. The compiled code contains no or little parallelism on instruction level. This could be caused by the processes of optimization where compiler tries to find similar parts of the code end reuse them. This works on scalar processors, but on VLIW architectures, where the detected code could contain different parallel operations which cannot be changed at run-time, it makes the target processor behaves as it has only one functional unit.

The thread parallelism can be helpful for processing a large amount of data. On smaller inputs, the cost of creating parallel regions by the operating system could be much bigger than the data process itself. This makes thread parallelism inappropriate for creating the cores of the DSP algorithms. For this purpose, the low-level programming languages can produce highly optimized code, especially on VLIW architectures. The disadvantages of the low-level languages are the longer development time and the fact, that the produced code could be used only on the specific platform. The results were published in [67], [70] and [71].

4 Impact of the software efficiency to the power consumption

The previous chapter showed how the different approaches of software creation affect the final performance of the application. This has an influence on the final time of data processing. But there is also another aspect which is affected. It is the amount of energy which is consumed while the application is running. This chapter will show the behavior of the real systems from the view of the power consumption when the program is executed on different number functional units and cores.

4.1 Theoretical power consumption increase on multi-unit systems

As it was mentioned, the software performance could have also impact on the power consumption of the system. In case of the scalar systems, the relation between the total energy and time is clear. The energy is given by

$$E = P \cdot t \tag{4.1}$$

but only under assumption that the power requirements are the same for every operation. The input power P contains the static power of the processor P_S, dynamic power of the ALU P_D and the background power P_B, which includes the other circuits in the system.

The situation in parallel systems is slightly different. In case that the total input power P changes only with the dynamic power P_D of the functional units. The total energy in this case is given by

$$E = (N \cdot P_D + P_S + P_B) \cdot t \ . \tag{4.2}$$

In simply case when the N units will compute the result in time t and the same algorithm will be computed in time $N \cdot t$ with single unit the system with single unit will be more efficient when

$$(N \cdot P_D + P_S + P_B) t > (P_D + P_S + P_B) \cdot t \ . \tag{4.3}$$

The equation (4.3) has the solution only when

$$N < 1 \tag{4.4}$$

what means that it cannot happen, because the real systems have at least one functional unit. So, even when the multicore system is fully loaded and its power consumption is at

its maximum value, its final consumed energy is less than the same result is achieved on the system with single ALU.

4.2 Practical test cases

The previous theoretical power consumption assumes the linear increase of the input power with the number of working functional units and some background power input for additional circuits. At this point, the ratio between static and dynamic power is unknown. This part will identify the real impact of the software optimization.

Several functions were proposed for measuring the difference of the DSP power consumption. The functions combine usage of all functional units for fixed or floating-point operations and data loading or storage as well. The power consumption was measured when one (A) or both data paths (A+B) were used for processing. The dependence on number of running DSPs cores was observed, as well. All functions were programmed in low-level assembly language to reach the requested operations and the codes were executed from the L2 cache memory of each core. All proposed test cases are described below.

4.2.1 Case 1: Empty loop

The first function is the empty loop. The function contains only one branch function to itself, executed by the .S1 unit (Figure 4.1). This instruction takes exactly 6 clock cycles, so the processor could execute other 5 instructions before the actual branch. For this reason, 5 no-operation instructions (`NOP`) were inserted after the branch code. This case should have the lowest power consumption from all test cases, because it uses only one functional unit, next it does not manipulate with any registers (except the jump address), and it does not modify the memory space.

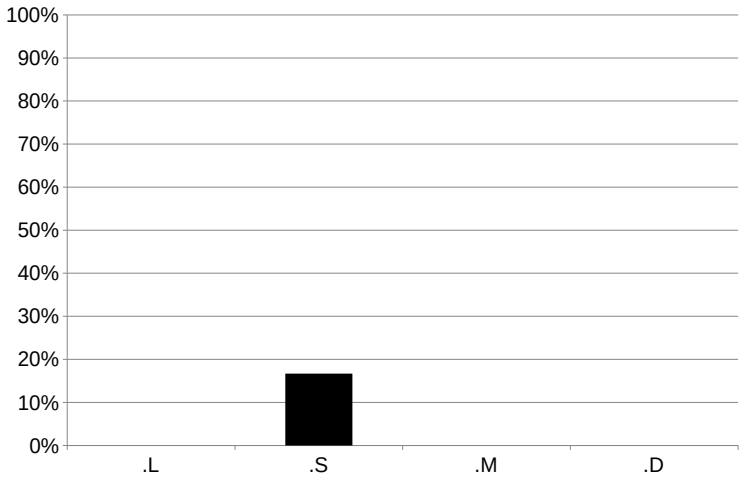

Figure 4.1: Functional unit utilization for the case 1

4.2.2 Case 2: Load/Store operations

The load and store operations use the .Dx units and perform data transfer between memory space and general-purpose registers (Figure 4.2). The instructions are pipelined, so the no-operation instructions are not required, and the next load/store operation can follow immediately after the previous one. The same applies to the branch instruction, creating the infinite loop. The data are loaded/stored from/in the shared on-chip memory.

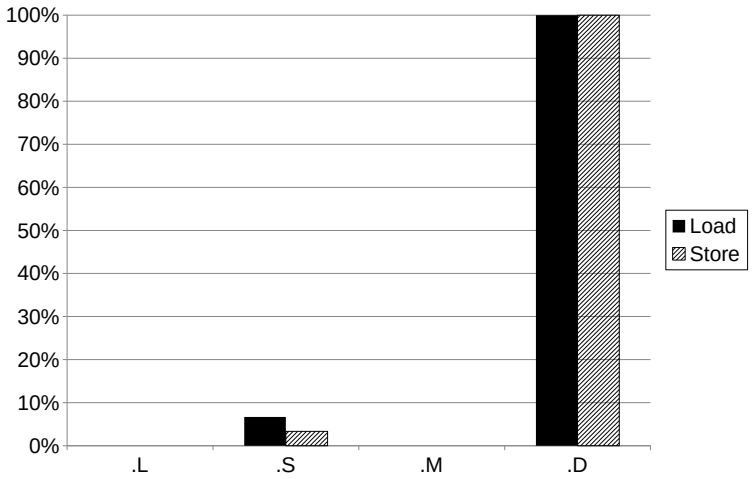

Figure 4.2: Functional unit utilization for the case 2

4.2.3 Case 3: Fixed-point operations

All DSP's functional units can perform fixed-point operations (Figure 4.3). The proposed test function contains addition, subtraction and multiplication of register values. All used instructions are also pipelined, similar to the previous case.

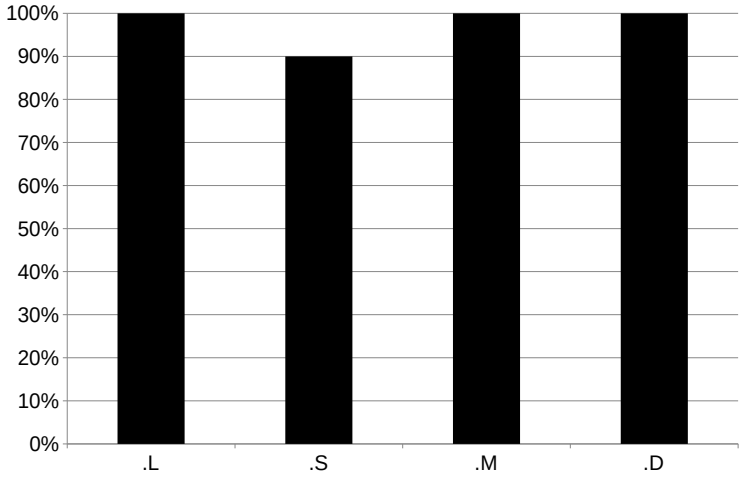

Figure 4.3: Functional unit utilization for the case 3

4.2.4 Case 4: Floating-point operations

Floating-point operations are supported only by .Lx, .Sx and .Mx units (Figure 4.4). The .Dx units are not suitable for these operations. Units perform addition, subtraction and multiplication of floating-point values stored in general-purpose registers.

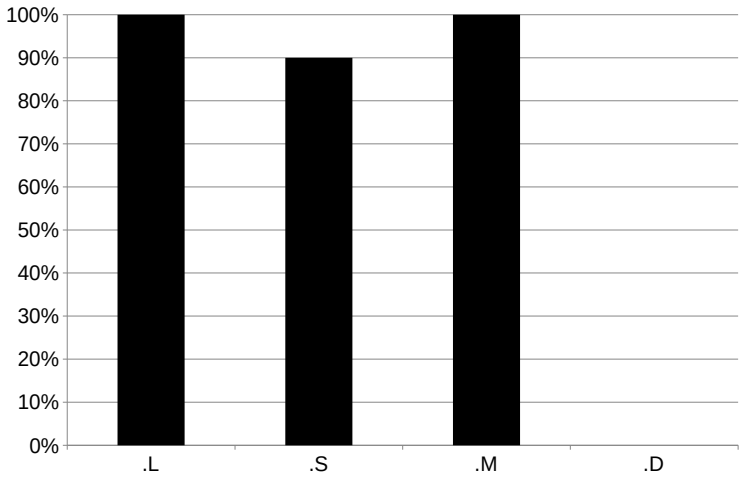

Figure 4.4: Functional unit utilization for the case 4 without data loading/storing

To reach the maximum DSP core exploitation, the proposed floating-point test cases were combined with the loading/storing operations as well (Figure 4.5).

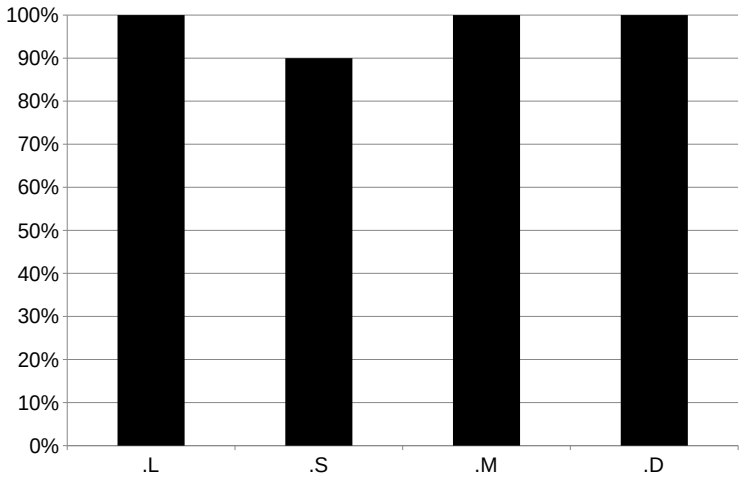

Figure 4.5: Functional unit utilization for the case 4 with data loading/storing

4.2.5 Case 5: FFT routines

Above mentioned theoretical test cases were appended by a real application from the digital signal processing domain. The real and complex FFT routines with length of $N = 8$ were tested. The functions adopted single precision floating-point representation and contain needed arithmetical and loading/storing operations for decimation-in-time

FFT algorithm. The routines were executed in data paths A and B (Figure 4.6), and combine four independent calculations in a single routine call. In average, every 1.28 CPU cycles one FFT transform with a real input vector is calculated, and every 1.97 CPU cycles one FFT transforms with a complex data is calculated with C6678 DSP.

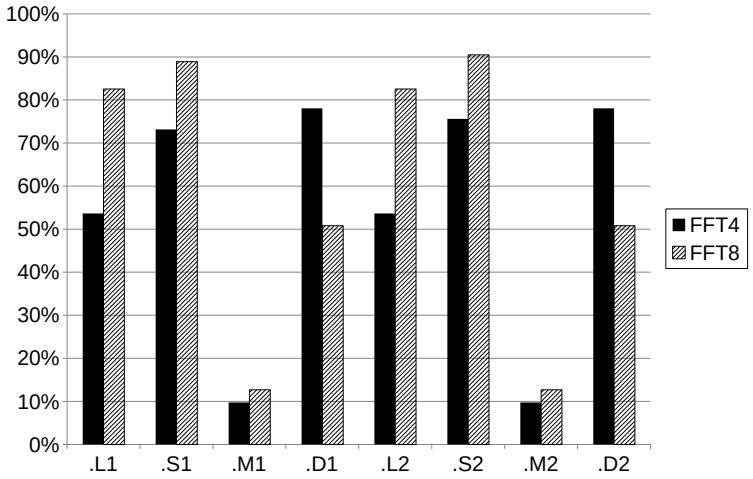

Figure 4.6: Functional unit utilization for the case 5

4.3 Experimental Results

As mentioned in section 3.1.1, the evaluation board has no possibility to measure power consumption of individual parts. But the power consumption can be measured relatively from the idle power level. For the measuring reasons, the power supply adapter was replaced by the regulated laboratory power supply unit Diametral P230R51D and power consumption was measured with two multimeters Agilent 34405A (for current and voltage). The multimeters can communicate with PC through the USB, so the samples can be captured in a synchronous way and the final power consumption can be calculated. For data capturing, the simple application using .NET and VISA drivers was programmed. Each measurement was done 10-times with frequency of $f_{measure}$ = 2 Hz and the final value were determined as the mean function from the samples. The experimental workplace is shown in Figure 4.7.

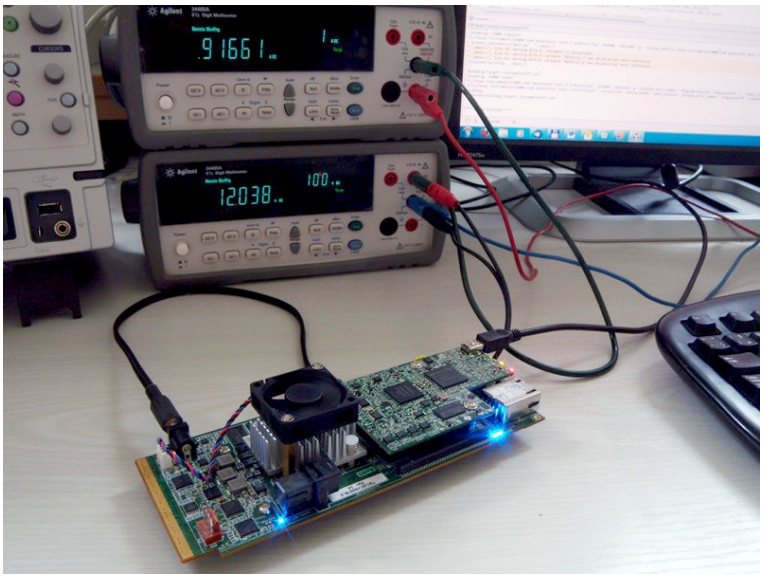

Figure 4.7: Workplace for the measuring the power consumption

The results for routines executed at data path A (half of DSP core), data paths A and B, and the real FFT functions are shown in Figures 4.8, 4.9 and 4.10 respectively. Remark: the idle power consumption of the development board was measured when all DSP cores were stopped. This value is representing the background consumption of the board (FPGA, clock generators, memory, emulator…) and the static power of the DSP; the value was 10.93 W.

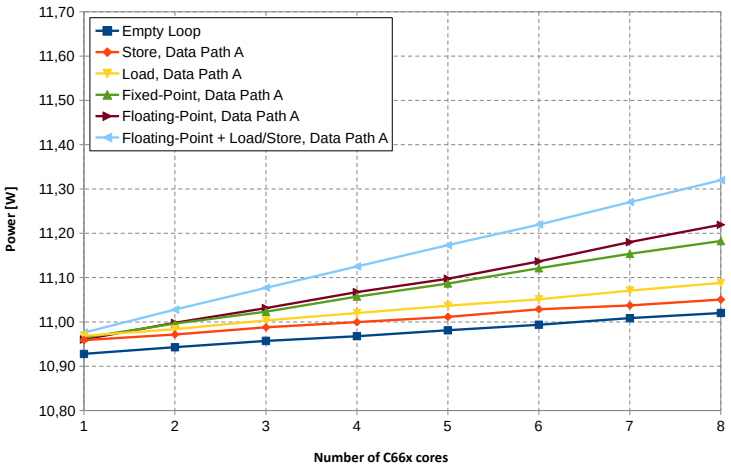

Figure 4.8: Power consumption of theoretical test cases at data path A

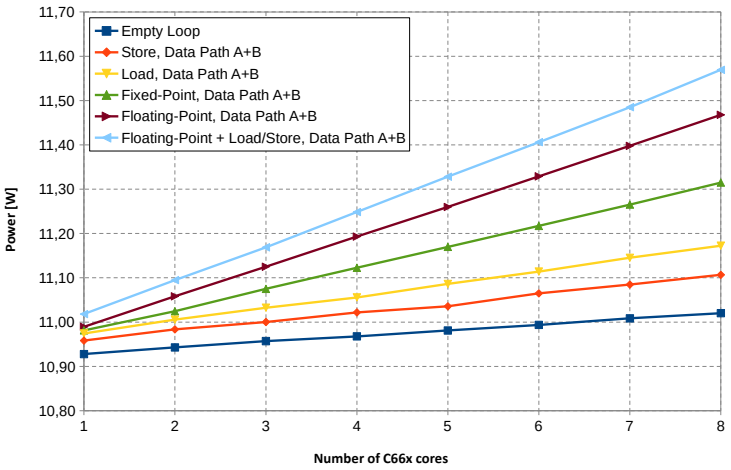

Figure 4.9: Power consumption of theoretical test cases at data paths A and B

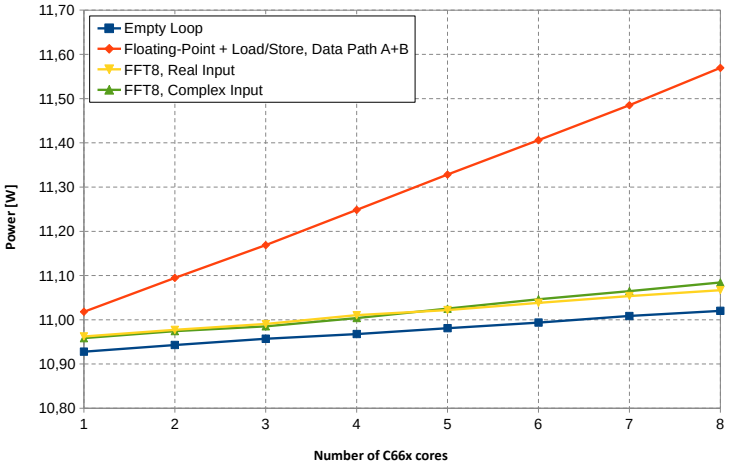

Figure 4.10: Power consumption of FFT routines at data paths A and B

A few experimental conclusions can be observed. First, the loading and storing operations do not have the same complexity; the loading data into the register file is more power demanding then the storing operation. It relates with the operations' duration - i.e. instruction for loading double words (`LDDW`) needs 5 CPU cycles and instruction for storing double words (`STDW`) is a single-cycle instruction only. Second obvious result is the bigger power demanding of floating-point operations then the consumption of the fixed-point instructions. Finally, in spite of average function units' loads of real FFT routines (54 % for real and 59 % for complex version, respectively), the average power consumption is closed to the simplest test case titled "Empty Loop".

4.4 Chapter summary

The software of the application does not have impact only on computation time, but also on the power consumption of the system and the amount of the spent energy to achieve the result. The utilization rate of the functional unit and processors cores have linear influence on the power consumption increase, but the static power input is significantly higher. It means that when the processor is fully utilized on all functional units and cores, its power input is comparable with the state when only single functional unit is processing data. The final energy to achieve the result is then given by time, so the way to reach the best power efficiency is to make software which uses all hardware resources as much as possible. Some related information was published in [72].

5 Instruction mapping tool for DSPs

The previous chapters showed the impact of the software on parallel architectures to the final performance of the DSP application. The thread parallelism is suitable for applying the DSP algorithms on larger amount of data like splitting the signals into the smaller parts and process them separately. This method is inappropriate for creating DSP core functions. For this purpose, it is better to create parallelism on the instruction level. Previous test cases were aimed on VLIW architectures, where the parallel execution of instructions must be desired during the compilation time. The easiest and fastest way to implement the DSP algorithm is to use high-level language. This process produces relatively optimized code for scalar architectures, but not for VLIW cores. The low-level approach using the assembly language is still significantly better for optimization in this case. This process demands lots of time and mistakes can be quickly made.

To ease the process of the DSP core functions optimization in low-level languages, the tool for instruction mapping will be made. The goal of this tool will be to process some easy writable mid-level notation of the algorithm core and generate the assembly language code. The result should be comparable to the hand-written implementation. The tool also must be adapted to fact, that the code can be ported into the different architecture without intervention in the tools code.

The first part of chapter will introduce the idea of signal-flow graph approach. Next part shows the definition of target architecture and algorithm to map. Then the text contains the details about the mapping process. The last part will describe implementation of the proposed tool.

5.1 The idea of signal-flow graph approach

As we can see from the benchmarks, the main problem of the standard compiler on the VLIW architecture is that it is not searching for the possible parallel operations. For example the main optimizations in GCC are based on control flow analysis, data flow analysis respectively, leading to reduce the call of code with no impact on the result or reduce the calling of jump functions, such as dead code elimination or redundancy elimination [57]. Some of them were described in chapter 1.4. These methods lead to the faster execution of program. Then there are methods for code size optimizations, which are trying to find similar code parts and reuse them as function calls. But these methods are aimed more likely for the control applications, where the scalar architectures are dominant.

The VLIW architectures are intended for use in DSP or generally in data processing applications. This leads the idea of the new tool to not using the sequential notation in the algorithm description, but the using of signal flow graph. The name can be similar to

the data flow analysis in the GCC, but it has completely different meaning. While the data flow analysis traces the variables in the code to find out if they are used and there are not performed useless operations, the signal-flow graph method will be used for finding relations between operations and subsequently the possible parallel operations will be searched. So, it will be similar to the hardware description languages (HDL).

5.2 Input files

Because the instruction mapping tool should be independent on the architecture, the input of the tool is not only the algorithm to map, but also the description of the target. Next text will describe the format of the input files.

5.2.1 Architecture definition

The hardware architecture is defined in text file in the JavaScript Object Notation (JSON) format [73]. This format is alternative to the Extensible Markup Language (XML) [74]. The only advantage of JSON over XML format is the smaller data representation and better human readability, because JSON does not use tag pairs. This is useful in case, when the architecture description file is edited by the hand.

The architecture description model structure is based on the TMS320C6678, which was chosen to be used in the previous benchmarks, but when some resources are omitted in the definition, some other architecture such as ARM can be defined. Its simplified structure is shown in Figure 5.1.

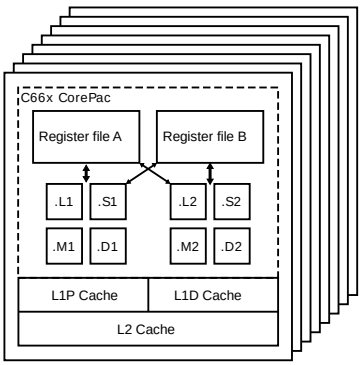

Figure 5.1: Structure of the TMS320C6678

At first sight, it may seem that the core has quite large amount of the resources for parallel operation, but it has its limitation.

The first is that the functional units are not equal. They are not capable to execute the same instructions. Functional units are marked .L1, .L2, .S1, .S2, .D1, .D2 and .M1, .M2. The .D units are primary used for loading and storing data into the memory. The .L and .S units are designed for the general arithmetic, logic and branch

operations as well. The last, .M units, are able to perform multiply operations with single and double precision floating-point values. All of the units are also able to execute other types of instructions, but not with all data types.

The second limitation is caused by the division of the previously mentioned hardware resources into 2 identical data paths. These data paths are marked as data path A and data path B. Because of this, it is not possible to directly access registers from data path A with functional unit from data path B. It can be done only through the register file cross-paths marked 1x and 2x. The single cross-path in the C66x is capable to transfer 64-bit operand in the instruction. In addition, this operand can be used in multiple instructions in the same execute packed, which was not allowed in the older C64x core.

The model itself is aimed only on the description of the processor core, not the processor as the entire unit. Its structure is shown in Figure 5.2. The main parts of the model are:

- hardware resources of the core (data paths, cross-paths);
- instruction set.

```
{
  "name":          "TMS320C6678",
  "regwidth":      4,
  "dataPath":      [
                     { ... },
                     { ... }
                   ],
  "crossPath":     [
                     { ... },
                     { ... }
                   ],
  "instruction":   [
                     { ... },
                     { ... }
                   ]
}
```

Figure 5.2: Basic structure of the JSON architecture file

5.2.1.1 Hardware resources

The topology of the model is based on the VLIW architecture with the multiple data path.

Data paths

From the external point of view, the data path is the top level element, which contains all basic hardware resources. For this reason, the part of the model with the hardware resources is set of structures describing the data path.

The selected TMS320C6678 has 2 practically identical data paths, so the model in this case can contain only the template of one data path and information about the number of the data paths in given architecture. But in general, the processor may consist of several different data paths, so every element in the model has its own definition.

```
{
  "name": "A",
  "unit": [
    {
      "name":       ".S1",
      "crosspath": ["NONE", "X2"]
    },
    { ... }
  ],
  "register": ["A0", "A1", ...],
  "reggroup": [
    {
      "name": "pair",
      "assign": [
        {
          "label": "A1:A0",
          "regs":  ["A0", "A1"]
        },
        { ... }
      ],
    },
    { ... }
  ],
  "datatype":
  {
    "int32": "register",
    "int64": "pair",
    ...
  }
}
```

Figure 5.3: Structure of data path in JSON file

Each data path is defined by:

- list of functional units;
- list of registers;

- register groups definition;
- data types representation.

The structure of the data path description is shown in Figure 5.3. Data path is identified in the system by its name in string format. The functional units and registers are stored in the arrays of objects. The register groups and data types are virtual (or logical) objects for specifying the data representation in registers. These features will be explained later.

Cross-paths

As it was mentioned in the TMS320C6678 description, the data paths work as the separated units. The data cannot be directly moved between the register files and the functional units cannot read the register value. For this purpose, the model is able to define cross-paths (Figure 5.4).

```
{
  "name":     "X1",
  "source":   "A",
  "width":    2,
  "operands": 4
}
```

Figure 5.4: Structure of cross-path in JSON file

Each cross-path is defined by the following parameters:

- source data path with register file;
- maximum width of the transferred data;
- maximum number of operands where the value can be used.

The meaning of the source data path is clean. The target data path is not defined at this point, because the functional units in the TMS320C6678 are not handling the operands in the same way. The .D, .M and .S units can read only the second operand through the cross-path and the .L units can access to the different register file for both operands (Figure 5.5) [59]. For this reason, the destination of the cross-paths is defined individually on the functional units.

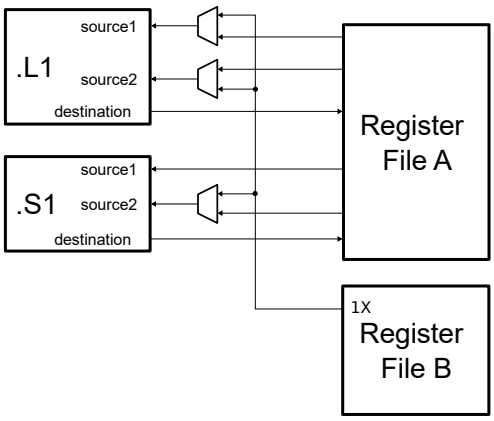

Figure 5.5: Example of the cross-path connection to the functional units

The maximum width of transferred data is given by the bus width, which is 64-bit in the selected processor despite the fact, that the register size is 32-bit. There is no need to define this parameter to different value than the multiply of register width, so the model keeps only the number of possible transferred registers.

The requirement of parameter which can tell if it is possible to use the operand transferred by the cross-path in the multiple operations is given by the difference between the C66x and C64x cores. In the C64x, it is possible to use the data from the cross-path only in one functional unit at once in compare with the C66x where this limitation does not exist.

Functional units

Each data path includes the set of functional units. The only one parameter, except the name, of the functional unit is the identification of the operand input connection to the cross-path. The referenced C66x and also the older C64x are composed of the 2 data paths, so in this case the parameter could only be with the meaning connected or disconnected. But in general, the processor could have more than 2 data paths and therefore it is needed to identify which cross-path is connected into the functional unit input.

Registers

The last physical hardware resources in the presented model are the general-purpose register files. Each data path has one register file defined by the set of the registers. The registers are identified only by their names. Even the width of the registers is not mentioned in the model. To determine how many and which registers to represent data type, virtual resources are used. They will be described in the next chapter parts.

Register groups and data types

Register groups are only logical definitions for the tool, to determine which registers can be used together as a single value (Figure 5.6). As it was mentioned, the model is not working with the physical width with the registers. Also, the registers can handle different number of bits on different architectures, so the decision which group to use as given data type cannot be made. For this reason, the data types supported by the tool are assigned to the created register groups.

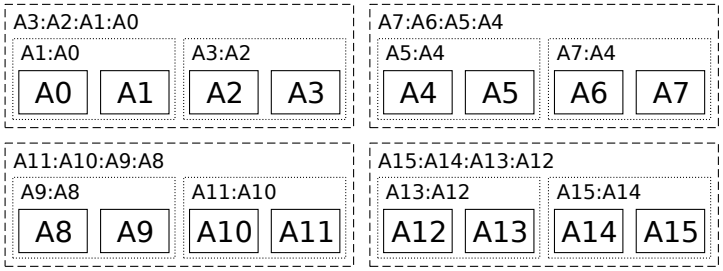

Figure 5.6: Creating register groups from the physical registers

5.2.1.2 Instruction set

The instruction set is the next part of the processor model (Figure 5.7). It is not divided into other segments as the hardware resources, but it is only the list of the instructions that can fit into the operation abstraction of the tool. The model includes 3 types of the instructions which are:

- arithmetic instructions;
- memory instructions;
- general functions.

Each instruction type is derived from virtual class, which is used in mapping process to simplify the algorithm. This virtual class contains information about:

- name of the instruction;
- instruction format;
- number of cycles needed to the instruction and operands;
- number of cycles needed to write result to the registers;
- total number of cycles.

The meaning of the instruction name is clear. Its purpose is only the identification by the user in architecture definition.

```
{
  "name":   "ADD",
  "format": "ADD unit src1, src2, dst1",
  "type":   "add",
  "data":   ["int32", "pointer"],
  "units":  [".D1", ".D2", ".L1"],
  "read":   1,
  "write":  1,
  "total":  1
}
```

Figure 5.7: Structure of instruction in JSON file

The instruction format gives the position of the parameters in the final notation of the generated code. For example, the `ADD` instruction format for C66x is `ADD unit src1, src2, dst1` [23].

Some of the instructions are able to process data with different number representation. For example, the `ABS` instruction in the C66x is able to process 32-bit integers and 64-bit integers as well [23]. That is why this parameter is list of the data types.

Functional units are another list acting as the instruction parameter. This list contains the functional units from all data paths. They are not divided into smaller groups.

The last group of parameters defines the timing of the instruction. The full instruction cycle was reduced into 3 stages. During the read stage, a functional unit is fetching instruction and the input value must be prepared in registers. After this stage, the functional unit can be used for other purpose and the input register can be overwritten. The write stage moves the result of the operation into the destination registers. At this stage, the register must be prepared to receive new data to prevent overwrite the valid values for other operations. The instruction is executed between these stages and the resources can be freely used without limitations. Figure 5.8 shows the timing of the `ADDDP` [23] instruction as the example.

Pipeline stage	1	2	3	4	5	6	7
Read	src1_l src2_l	src1_h src2_h					
Write						dst_l	dst_h
Unit in use	.L / .S	.L / .S					

Figure 5.8: Execution progress of ADDDP instruction

During the execution of the ADDDP (double precision floating-point addition) instruction, the functional unit .L (or .S) is completely utilized in the first 2 CPU cycles. Due to pipelining, it can be used to execute other instructions. The input arguments are stored in 2 register pairs (src1_h:l, src2_h:l). After the registers are read, they can be used for another purpose. The result of the operation is stored into the register pair dst_h:l in cycles 6 and 7. In cycle 8,the value can be used in the next CPU cycle.

Arithmetic instructions

The first derived instruction type is arithmetic instruction. It extends the base instruction class with the following parameters:

- operation;
- data type.

The operation defines arithmetic function of the instruction. The recognized values by the tool are specified in Table 5.1. The second parameter is the list of data types supported by the function. The data types keywords are shown in Table 5.2.

Table 5.1: Arithmetic instruction supported operations

Operation	**Description**
ADD	Addition
SUB	Subtraction
MPY	Multiplication
DIV	Division

Table 5.2: Arithmetic instruction supported data types

Data type	**Description**
INT8	8-bit integer or fixed-point
INT16	16-bit integer or fixed-point
IN32	32-bit integer or fixed-point
INT64	64-bit integer or fixed-point
FLOAT	Single-precision floating-point
DOUBLE	Double-precision floating-point
POINTER	Pointer (usually size of register)

Memory instructions

Memory instructions are dedicated to loading registers and storing their values back to the memory. This type extends the base instruction type with the operation, which is shown in Table 5.3. There is nothing such as data type like in arithmetic instruction, because these instructions work only with registers.

Table 5.3: Memory instruction supported operations

Operation	**Description**
LOAD	Load from memory through pointer
STORE	Store to memory through the pointer
CONST	Load constant to the register
CONSTH	Load constant to upper half of the register
CONSTL	Load constant to lower part of the register

Functional instruction

The functional instructions are for general-purpose. They are used when the operation in the algorithm does not fall under the previous categories. In this case the base instruction structure is extended like in arithmetic instruction, but the difference is the operation. It is not represented by the enumerated value, but with string which is later compared with the functions in the algorithm.

5.2.2 Algorithm description

The algorithm notation uses the signal-flow graph description based on HDL and the tools syntax uses two base elements: signals and nodes. The signals are equivalent for the variables, but there is a limitation for their use. In classic sequential programming languages, like ANSI C, the variable can change its value during runtime many times. In the case of this tool, the value of the signal can be assigned only once. There are three types of signals: input signals which are allocated at the beginning of the runtime, the output signal must be valid until the end of execution, and temporary signals could be created and expired when required.

The other elements in the syntax are nodes. A node represents the elementary operation, while the nodes are architecture and data type independent. These parameters are assigned during the process of code generation. An example of the algorithm description and its graphical representation is shown in Figure 5.9. The graphical representation of the algorithm is on the left side. The right side contains text representation.

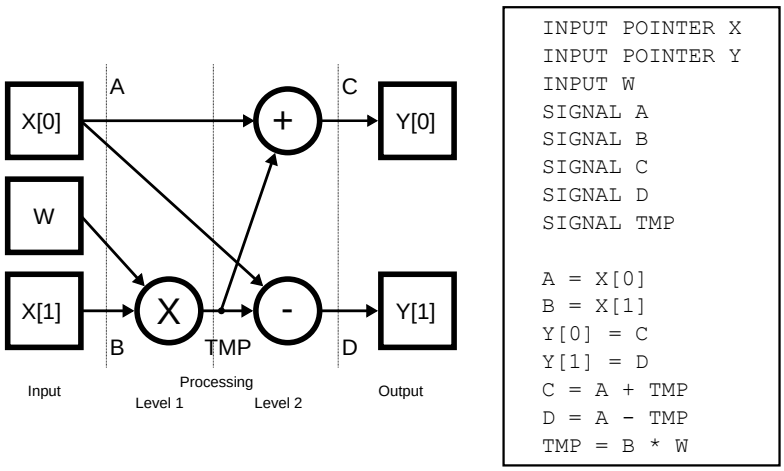

Figure 5.9: Signal-flow diagram from example algorithm

The example represents a simplified butterfly of FFT, without complex numbers. It has 3 input signals `X`, `Y` and `W`, which can be compared to the arguments of the function in ANSI C language. Signal `X` is the pointer to array with input samples, signal `Y` is the pointer to array with output samples and signal `W` represents the twiddle factor, which could be also replaced by constant loading. There are also internal signals `A`, `B`, `C`, `D` and `TMP`. The `TMP`, `C` and `D` signals are created by the arithmetic nodes. Signals `A` and `B` are loaded from memory using pointer `X`. Signals `C` and `D` stores the result of the operations which are passed out of the algorithm by pointer `Y`. The operations on signals do not have to be written in the same order as they should be processed. The full syntax will be described later.

5.3 Algorithm mapping

The mapping process leads to the semi-ideal low-level assembly code of the algorithm for the target architecture and uses both architecture and algorithm description. The process can be split into the following steps:

- input files parsing
- assignment of node instructions
- sorting nodes and signals
- mapping nodes and signals
- generating output files

The order of the mapping steps is also shown on Figure 5.10, where is also small description. The details are given in the next subchapters.

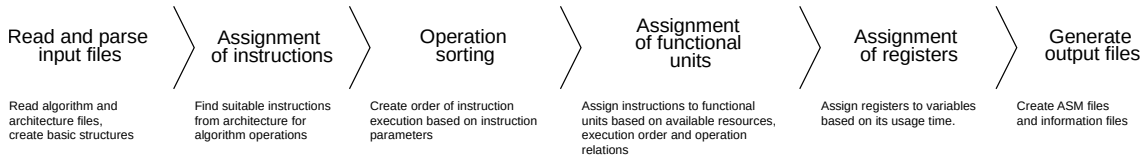

Figure 5.10: Mapping process

5.3.1 Input files parsing

First of all, the information about the target architecture and the algorithm needs to be retrieved from the input files. The architecture description has the same structure as the processor model described in chapter 5.2.1, which is in JSON format, and contains the information about

- data paths
- functional units
- registers
- instructions

The second parsed file is the algorithm. The processing of the algorithm description file creates structure which includes the list of

- nodes
- signals

5.3.1.1 Parsing signals

All signals must be defined before the operations. This means that the signal definition is located at the beginning of the file. Each signal is represented by its name, data type and role in the algorithm. The example of the definition format is shown on Figure 5.11.

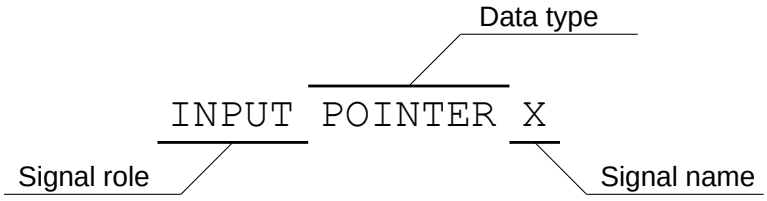

Figure 5.11: Signal definition format

The order of the keywords cannot be changed, but not all fields are mandatory. The data type of the signal is only optional for special cases, when the signal is not the part of the algorithm processing chain. In this case, it is the pointer to input data. The supported values are listed in Table 5.4 and 5.5.

Table 5.4: Signal definition roles

Signal role	Description
INPUT	The signal carries the input data. It is located at the beginning of the algorithm and data is filled outside of the block to the registers.
OUTPUT	Output signal usually carries the result of the algorithm. Once the signal is allocated during the processing it is not destroyed.
SIGNAL	This is the internal signal it could be result of the operation or it can be only the alias of another signal.

Table 5.5: Signal definition data types

Data type	Description
INT8	8-bit integer or fixed-point
INT16	16-bit integer or fixed-point
IN32	32-bit integer or fixed-point
INT64	64-bit integer or fixed-point
FLOAT	Single-precision floating-point
DOUBLE	Double-precision floating-point
POINTER	Pointer (usually size of register)

5.3.1.2 Parsing nodes

The nodes are practically every operation with the signal, but the tool recognizes 5 types of the operations, which have different definition format. These operations are:

- arithmetic operation;
- function;
- constant assignment;
- signal assignment;
- memory operation.

Arithmetic operation

The arithmetic operation contains basic mathematical operations with two input arguments, which produces one result. The definition is similar to other programming languages, but there can be only one operation per line. The format is shown on Figure 5.12.

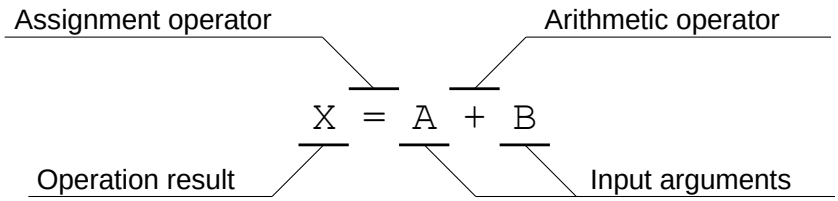

Figure 5.12: Arithmetic operation format

There are 4 types of operations which are supported in arithmetic nodes. The list of recognized operators is in Table 5.6.

Table 5.6: Operators for arithmetic operations

Operator	**Operation**
+	Addition
-	Subtraction
*	Multiplication
/	Division

Function

Next type of node is the function. It is generic operation with variable number of inputs and outputs. It is usually used for operations that does not fall into the other categories. The format is shown on Figure 5.13. The function is recognized by its name. The name usually corresponds to the processor instruction name that will be executed. The items in the result and input argument are separated with comma. Additionally, the input arguments must be placed into the parenthesis.

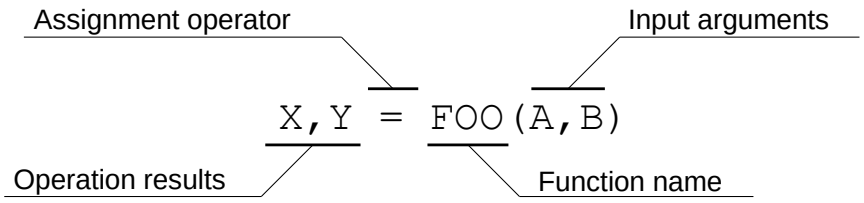

Figure 5.13: Function definition format

Constant

Signal can be loaded directly by the constant. The format of this operation is simple assignment of the constant number into the signal, what is shown on Figure 5.14. There is no need to follow numerical format of the constant for floating-point or integer values, because it will be automatically converted into the proper data type according to the target signal.

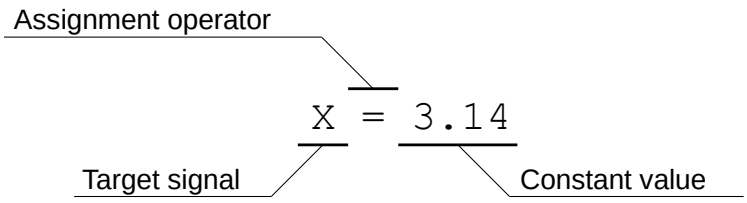

Figure 5.14: Constant definition format

Signal alias

Signal aliases are used to code clarity when it is constructed from multiple blocks. They are not actually new signals, only creates new names for existing signals. When the alias is created from another alias, the new one also references the original signal. The definition format is shown on Figure 5.15.

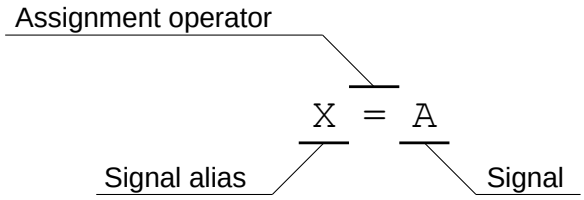

Figure 5.15: Signal alias definition format

Memory operation

The last node definition is the memory operation. This node can store or load value from the memory into the signal.

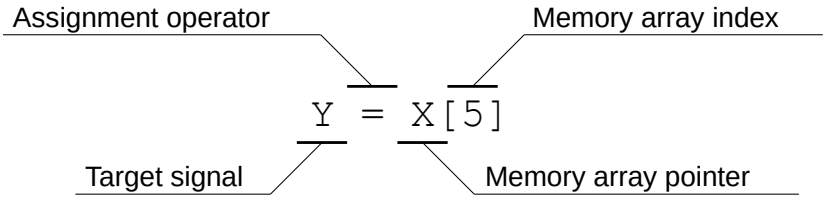

Figure 5.16: Memory operation format

Figure 5.16 shows memory load variant. The left and right sides can be changed to achieve memory storing. The pointer to the memory must be defined as type pointer (see chapter 5.3.1.1). The memory array index is zero-based integer constant. This index gives final memory address according to the target signal type width, so it is not representing shift in bytes.

5.3.1.3 Multi-operation nodes

Some of the nodes in the algorithm description requires multiple operations for achieving the result. It is typically memory operation mentioned before.

The first case is a constant loading. Even if processors support operations with wider data than its registers, they usually support only register loading, or even worst, only loading its upper or lower part. For this reason, the constant loading is divided into the multiple register loading operation according to the size of stored data type. These registers loadings are independent on each other, but it must be taken into account that the loading of the upper or lower part of the register can overwrite the whole register. This is not the problem of the C66x.

The second case is the loading value from the memory or storing value into the memory. First of all, there is the same problem as the constant loading, which is the different width of the registers and processed value data types. The second problem is the identification of the memory address, which is given by pointer at the beginning of the data array and the value index in that array. So, the mapping algorithm loads constant which corresponds to the index, then modifies the initial pointer and after that it can load or store the value.

5.3.2 Finding relations and validation

The next step after the file parsing is relation creation between nodes using the signals as a connection. The goal is mainly to determine the possible execution order of the nodes, but also to extend nodes description with the additional information. This process is also used to validate the algorithm connections.

5.3.2.1 Extending nodes and signals information

The nodes description can be in this step extended with the specific instruction, which can be executed in the final code. The instruction selection is based on the operation of the node and the data type of the signals connected to the node. In case that the instruction set contains multiple instructions with the same function, the one with shorter execution time is chose.

The signals description can be also extended. Because the signal can be represented in different data type, it could require different number of registers. According to the data type of the algorithm, the register group from the architecture structure is chosen what will lead to physical registers which can be used.

5.3.2.2 Determining execution order

The mapping process, as it could be seen later, is based on the first-fit method. For that reason, the processing order of the nodes must be considered before the allocation of functional units.

The first and the most important parameter is the execution level of the nodes. The execution level value is based on the node relations. There are two rules for defining the execution level of the node:

- The execution level of the node is zero if all its input signals are the input signals of the algorithm.
- The execution level of the node must be higher than the highest execution level of the nodes which creates its input signals.

Figure 5.17 shows case where the execution level is given by input signals of the algorithm. Node 1 has two input signal and both of them are input signals of the algorithm. This means that the node has execution level equal to zero and it is possible to execute it in the first instruction cycle. Node 2 also uses the algorithm input signal, but the second signal is created by the node 1. In this case the second rule is applied, and the execution level needs to be set to the higher value than the execution level of the node 1.

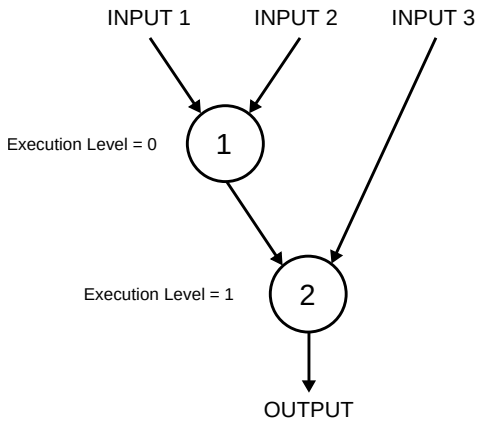

Figure 5.17: Determining execution level using input signals

Figure 5.18 shows case with nodes in the middle of the algorithm. The examined subject is the node 4, where the process level is given by the nodes 1 to 3. The maximum execution level of these nodes is *N* so the execution level of the node 4 is *N*+1.

A special case of the order determination is a constant loading. Because nodes loading constants are not dependent on input signals from other operations, so it can be executed in the beginning of the program. With the first execution level it is assured that the nodes which uses constants as arguments are not blocked and can be mapped as soon as possible.

This solution could be ineffective, because the generated signals allocates registers from the beginning of the execution. This could be solved by execution level update on

constant loading nodes after execution level assignment of all nodes according to lowest one execution level of all nodes which uses specific constant.

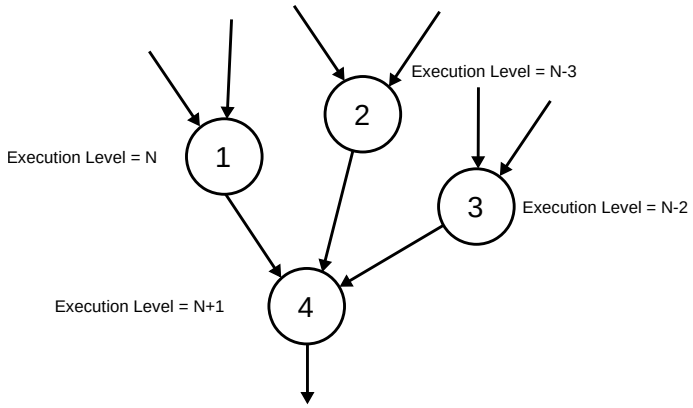

Figure 5.18: Determining execution level using previous nodes

Figure 5.19 shows the situation where node 3 uses constant *C* as input. Firstly, this constant is set to beginning of the execution. After the execution level assignment of all nodes, the constant *C* changed its level to one up to reduce the usage of allocated registers.

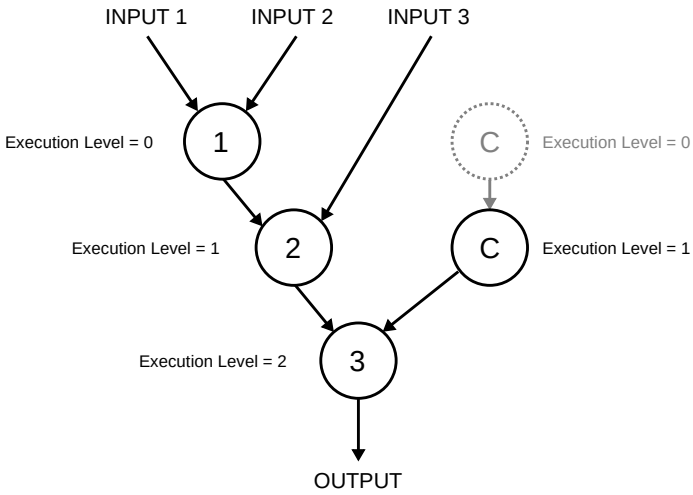

Figure 5.19: Determining execution level of constant loading

5.3.3 Node sorting

Node sorting is the last step before allocation process. This operation creates the order in which the nodes will be allocated into the functional units. The main purpose is to reduce the demands on computing power on the allocation process. The sorting can also be used to optimize the resulted code, when the operations that can use only few available resources will be allocated first and these resources will not be blocked by another operations.

5.3.3.1 Sorting according to execution level

The execution level is the only parameter which is considered in in all sorting methods. When the execution level is ascending in the list of the nodes, the mapping process can go through only with one iteration and all nodes are mapped to available resources. This results in the state that the nodes which operates only with input signals are mapped first and the nodes which creates output signals are mapped last.

5.3.3.2 Sorting according to the total CPU cycles of the operation

To achieve better performance of the generated code, other parameters should be considered in determination of the order of node mapping. These parameters will affect the order of the nodes in the list only within the same execution level.

The first parameter can be a number of total CPU cycles to execute an instruction assigned to the node. This could reduce execution time of the algorithm. Figure 5.20 shows three pipelined instructions executed on the same functional unit. The left case is the ideal order, when the first executed instruction takes 5 CPU cycles and the last 3 cycles. The result is written to registers at the same time. The case on the right is the worst case, when the instructions are executed in the reverse order. The execution of all instructions takes 7 CPU cycles instead of 5.

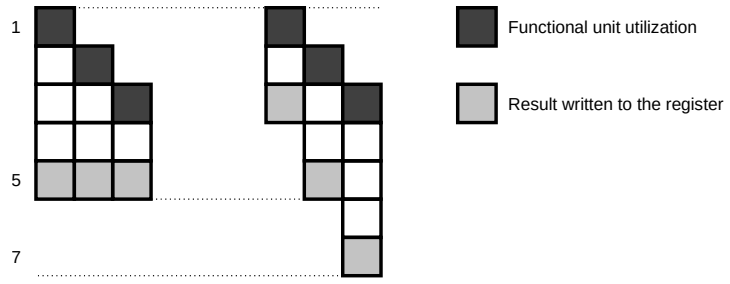

Figure 5.20: Instruction execution order based on CPU cycles

5.3.3.3 Sorting according to number of supported functional units

Second parameter could be the number of supported functional units where the instruction can be executed. Figure 5.21 shows the situation on two functional units A and B and five instructions. The shorter instructions (3 CPU cycles) can be executed on both functional units. The longer instructions (4 CPU cycles) can be executed only on functional unit A. The case on the left side is the worst case, when the short instructions are allocated first and then are allocated longer instruction (the order is indicated by numbers on top). The result is that the functional unit B is executing only one instruction and the rest is executed on the functional unit A. The execution of all instructions takes 7 CPU cycles. The situation on the right is ideal, because the longer instructions were allocated first, so they are not blocked by the shorter instructions.

Short instructions can be allocated on the functional unit B. The execution now takes 5 CPU cycles.

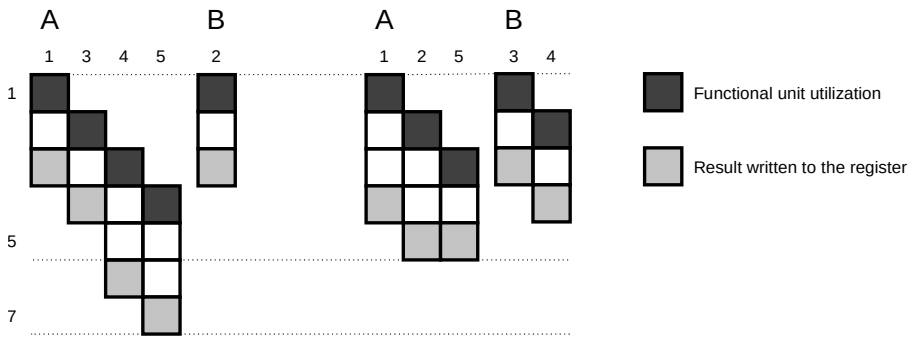

Figure 5.21: Instruction execution order based on number of supported functional units

5.3.3.4 Handling constant loading

Also, the type of operation can play a role in the execution order. As it was previously mentioned, the constant loading operations could be executed at algorithm beginning, but it will pointlessly allocate registers holding the constant value for a long time. For this reason, its execution level was modified to reduce lifetime of the generated signal. But even after this modification the signal could be generated relatively long time before it will be used. For this reason, the constant loading operations are placed into the end of its process level, so they will be allocated as last in the group.

5.3.4 Functional unit allocation

Functional unit allocation can be performed after the execution order of the nodes is determined from the previous steps (chapters 5.3.2, 5.3.3). This order is also the same as the order of the allocation process. The allocation process finding the free functional unit which can be used to execute instruction assigned to the algorithm node.

5.3.4.1 Finding start cycle of the execution

Before the allocating functional unit for instruction, the node needs to have defined the minimal start cycle, when the instruction can be executed. This cycle can be determined when the instructions from the previous execution level (chapter 5.3.2.2) are mapped. These operations create the signals which are processed by currently mapped node. The only special cases are nodes processing the input signals. These nodes have the execution level equal to zero and therefore can be executed immediately at the beginning of the algorithm, the others depend on the previous nodes.

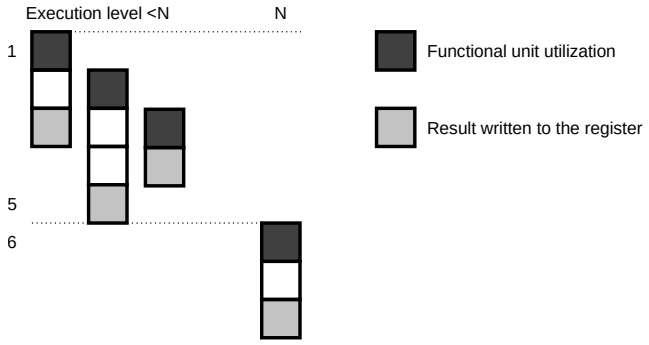

Figure 5.22: Determining first possible CPU cycle for execution

Figure 5.22 shows the instruction on execution level N which depends on the results from the three instructions on lower levels. The last result from these instructions is written on 5th CPU cycle, so the examined instruction could be executed on 6th CPU cycle. If there will be another instruction on the lower level which gives result after 5th CPU cycle, but it is not used in examined instruction, this information is irrelevant and the minimal possible execution start of the examined instruction is not changed.

5.3.4.2 Allocation

The instruction mapping into the functional unit is similar to the first-fit method in the memory management which means that the instruction is mapped into the first suitable position. The difference is that this allocation process must consider two dimensions (functional unit and time), not only single dimension like in memory management. The tool can be set to take priority on the functional units.

Without priority

This is the simplest method, which is actually equivalent of the first-fit. The tool starts examining the functional unit usage map from the instruction cycle, which is the first possible when the operation can be executed (chapter 5.3.4.1). When it finds that any of the functional units is unused, it places the node into the map. When there is no free functional unit, it moves on the next instruction cycle and repeats the process.

Global priority on number of supported instructions

The previous method of finding free unit does not consider any parameter. The order of the unit examination is given by its definition in the architecture. This method prefers the functional units that supports the least number of instructions, so there is a bigger chance that the allocated node will not block the next operations. The order of the functional unit examination is fixed through the process.

Priority on remaining number of supported instructions

This method is similar to the previous one. The difference is that the order of the functional unit examination is dynamic according to the instructions in the remaining unallocated nodes. In each node allocation step, it finds number of possible upcoming nodes which can be possibly executed on each functional unit. The highest priority has the functional unit with the smaller number as in the previous method.

5.3.5 Signal allocation

Signal can be allocated to registers only when all nodes are mapped, because there is relation between the node's execution time and the signals lifetime. The lifetime of the signal means the time, when the registers hold the value from the given node which created the signal and other nodes cannot rewrite this value. The registers are not allocated during the whole algorithm process, but only for the necessary time. Generally, the lifetime of the signal starts with the value write and ends with the last read of the target nodes. Special cases are input and output signals of the algorithm. The input signal registers are allocated from the first instruction cycle and the output signal registers keeps their values until the end of the algorithm.

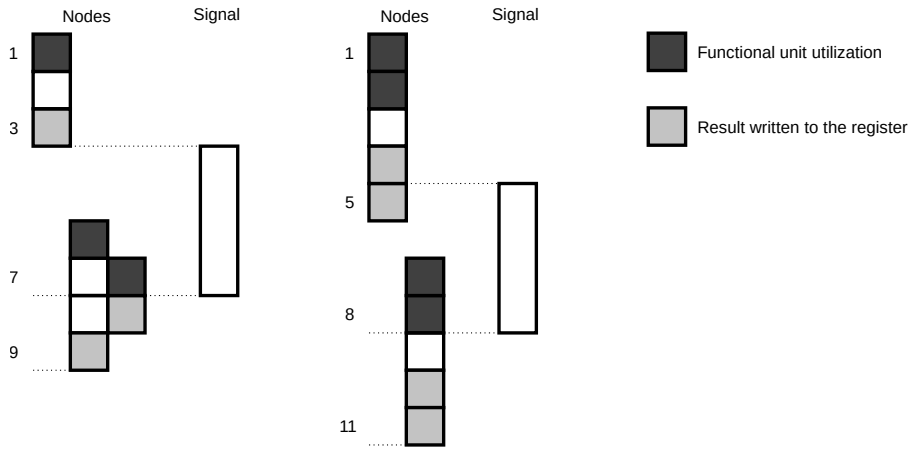

Figure 5.23: Determining signal lifetime

Figure 5.23 shows two cases of the signal lifetime determining. The first (left) shows the situation when the signal is used by two nodes. Signal lifetime starts one CPU cycle after instruction value write. This one cycle delay is caused by the possibility of using the same register for input and output by single cycle instructions. The signal lifetime ends after the last instruction read of the second target node.

The second case shows the situation with instructions which needs more than one CPU cycle for reading and writing. The lifetime end is after the read like in the previous case. The difference is in the lifetime start, which is not after writing as it may seem

from the previous situation, but it is after the first CPU cycle of the write. The behavior of determining the lifetime start and end is technically the same in both cases.

When the signals have given its lifetime, they are allocated to the registers in similar way as the nodes. The two-dimensional map of the register usage in time is created and the registers are placed into the map like first-fit method.

After this procedure, the final low-level assembly code can be generated, or the others information files as well.

5.4 Implementation

The tool is implemented purely in C/C++ language, mostly using standard system libraries. The only 3rd party library is JSON parser [75]. The tool is divided into two separated applications, the algorithm editor and the mapping tool itself. The structure of the code is shown on Figure 5.24.

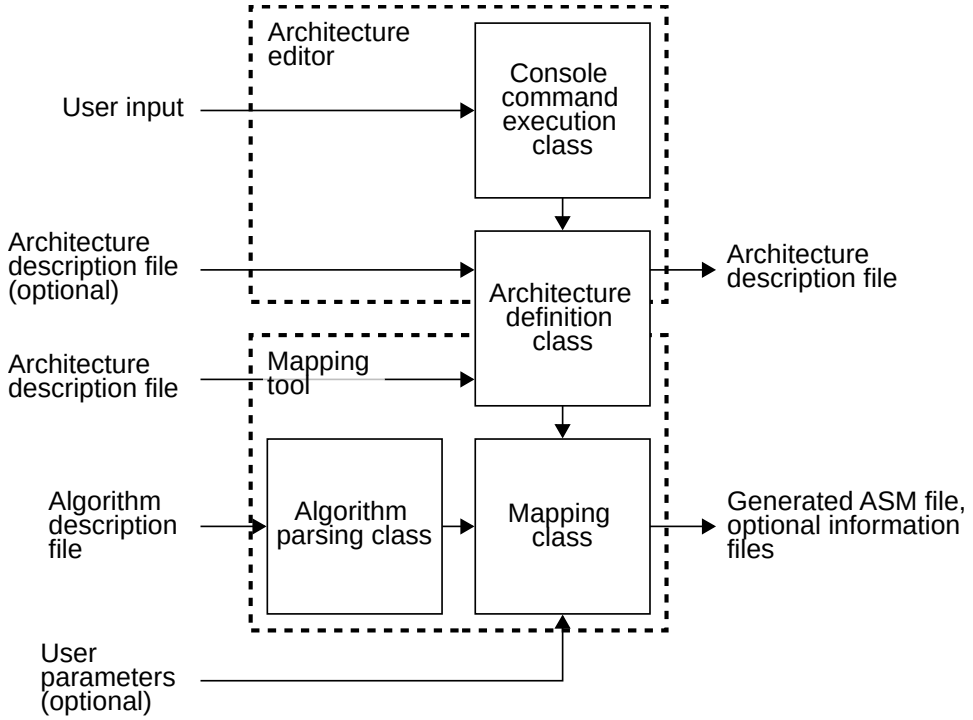

Figure 5.24: Mapping tool structure

The architecture editor is the command line application for defining the target architecture, so it is based on the architecture and command line classes. The purpose of this application is creating and editing architecture files which are used in algorithm mapping. The console command execution class is only parsing the user input and calls the architecture definition methods.

The mapping tool itself has no user interaction during the data processing in contrast with the architecture editor. It only takes the arguments at process start. From these arguments are taken paths to the input files to architecture and algorithm definitions which are parsed and passed to the mapping class. The mapping class can take additional arguments such as the data type and custom path to the output files. Thanks to this modular structure, the application can be extended with graphical user interface (GUI) using some of the multiplatform frameworks such as GTK+ [76] or Qt [77].

As it can be seen, these applications are sharing a considerable part of common code. It is mainly the architecture class, but also the data types and some of the console parsing procedures for getting application input arguments. The following text will describe the tools structure according to its separations into the libraries and executables in the project.

cJSON library

This library is the only external part of the project, which can be downloaded from [75]. It provides C functions for reading and saving the JSON formatted files. With cJSON, the application can work with basic types as booleans, numbers and strings. It also supports arrays with these types and the objects as well.

Command line library

Command line library is used for parsing input from the user in command line interface. It could be used for executing the functions according to its paired string name. It is also parsing the input arguments of the application and provides the help description for the usage.

Common functions library

This library provides common functions and data types for all parts of the project. It mainly contains types for the architecture and algorithm objects, but also functions for message printing or program version information.

Algorithm library

Algorithm library provides class for storing the algorithm to map. It contains lists with the algorithm signals and nodes. It also provides its own parser for reading algorithm description as it was shown in the chapter 5.3.1.

Architecture library

Architecture library provides the C++ class which describes the target architecture of the generated code. The class corresponds to its model described in the chapter 5.2.1. It

is dependent on the cJSON library, which is used for parsing and saving the architecture description.

Architecture editor

This part creates command line executable for defining and editing the architecture object by the user. It mainly uses the architecture and command line libraries, where the methods from architecture objects are paired with the commands in the user interface.

Mapping application

This application depends on all previously mentioned libraries. It also adds definitions of the other classes for creating the maps, which are used to generate the output files with the final assembly code. The mapping process uses process described in the chapter 5.3.

5.4.1 Build environment

The build process of the instruction mapping tool is controlled by the CMake. This lets the user to build the tool for the Unix based systems or for the Windows. The only requirement is the compiler supporting C++11 standard. The tool was developed and tested under Ubuntu 18.04 LTS and Windows 10 with Visual Studio 2017 Community Edition (Figure 5.25).

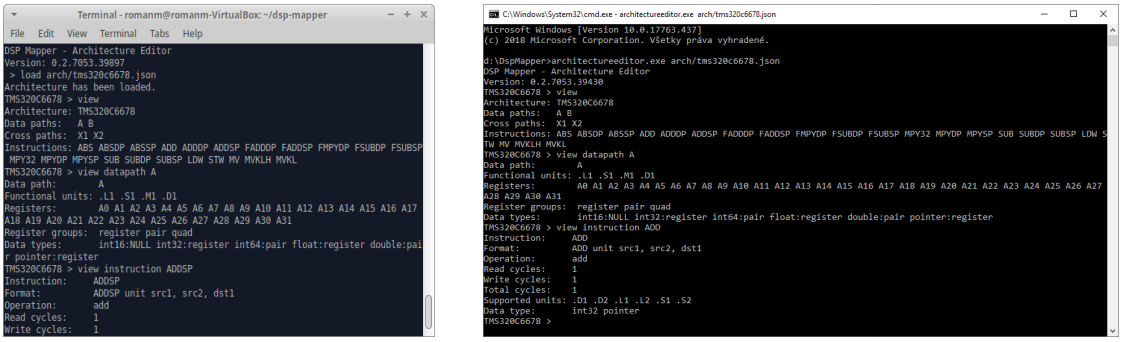

Figure 5.25: Architecture editor running under Linux and Windows system

5.5 Chapter summary

The aim of the thesis is to create tool for ease the process of optimizing DSP algorithms on VLIW architecture. This chapter described such a tool that was created. This tool is designed to generate the low-level assembly language from the abstracted code, which could be used as optimized part of the DSP core functions. The tool is not tied to the single type of architecture, but the target processor can be specified without the

modifying source code only with JSON formatted text file containing the architecture description.

The proposed tool uses signal-flow chart idea for finding the relations between operations in the algorithm. This approach is suitable for searching parallel operations, which can be mapped to be simultaneously executed on different functional units. For achieving better performance of the generated code, the operations mapping into the functional units are done with priority given by the statistics which reduces the occupation of the functional units which could be used by other operations which cannot be mapped anywhere else.

The tool is realized as two console applications. The first is for architecture definition and the second for generating assembly language code. These applications are written in the native C++ and can be compiled on Windows or Unix based systems. The tools introduction was published in [78], [79] and [80].

6 Experimental results

The proposed mapping tool was verified on several basic algorithms with the aim of observing the efficiency of the processor's functional units and general-purpose registers. The efficiency is determined from the usage of the resources. The goal of the tool is to use the potential of the parallel architecture, so the best efficiency is evaluated when all functional units are used during the all instruction cycles of data processing. The opposite state is when the VLIW architecture is using only single functional unit in single moment. At that time, the processor behaves simply as scalar architectures and takes no benefit from its structure. But the performance is not evaluated only from the number of used functional units. The architecture can benefit from instruction pipelining. When it is not used properly, the no-operation gaps start appearing in code. Because of this, the evaluation will also examine the ration between used and unused time slots for each functional unit.

6.1 Basic behavior of algorithm mapping

Two algorithms were chosen for their high potential of parallelization and for their indispensability in signal processing and communication domain: FFT and matrix multiplication. This part will show it's the behavior without any allocation priority. It will also show the difference with and without memory operations.

6.1.1 Values prepared in registers

The first case will test the algorithms in situation, when the input values are prepared in the registers. Also, the results of the algorithm will be stored in registers to pass them away from the function.

6.1.1.1 Fast Fourier Transform

To demonstrate all steps during the mapping process, the 4-point FFT radix-2 with time decimated complex input (equivalent to Figure 6.1) was implemented. For simplification, the twiddle factors were substituted by adding and subtracting operations. This simplification is achieved by twiddle factor

$$W_N^n = e^{\frac{-j \cdot 2 \cdot \pi \cdot n}{N}} \tag{6.1}$$

for number of samples $N = 4$, when it can reach only values

$$W_4^n \in \{1, -1, j, -j\} . \tag{6.2}$$

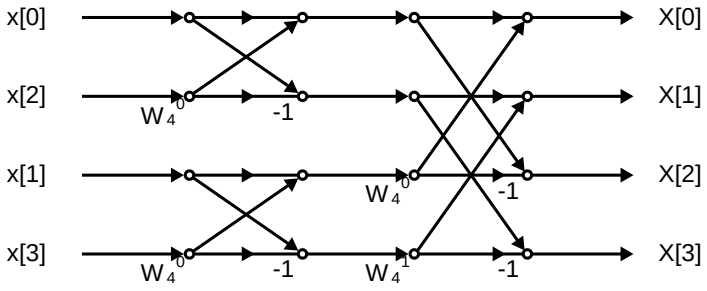

Figure 6.1: 4-point FFT algorithm

```
INPUT A1_RE
INPUT A1_IM
# ... (definition of A2 to A4)

OUTPUT C1_RE
OUTPUT C1_IM
# ... (definition of C2 to C4)

SIGNAL B1_RE
SIGNAL B1_IM
# ... (definition of B2 to B4)
```

Figure 6.2: Part of signal definition in the 4-point FFT implementation

```
B1_RE = A1_RE + A2_RE
B1_IM = A1_IM + A2_IM
B2_RE = A1_RE - A2_RE
B2_IM = A1_IM - A2_IM
B3_RE = A3_RE + A4_RE
B3_IM = A3_IM + A4_IM
B4_RE = A3_RE - A4_RE
B4_IM = A3_IM - A4_IM

C1_RE = B1_RE + B3_RE
C1_IM = B1_IM + B3_IM
C2_RE = B2_RE + B4_IM
C2_IM = B2_IM - B4_RE
C3_RE = B1_RE - B3_RE
C3_IM = B1_IM - B3_IM
C4_RE = B2_RE - B4_IM
C4_IM = B2_IM - B4_RE
```

Figure 6.3: Source code of the 4-point FFT (without signal definition)

The 4-point algorithm uses 8 input signals (4 real and 4 imaginary parts), 8 output signals and 8 internal signals. An example of its definition is shown in Figure 6.2. The signal is defined by its role (input, output, internal signal) and its name. The data type is

not defined in the source code, which allows to generate an assembly code for multiple data types.

The whole algorithm description is shown in Figure 6.3. This code is also abstracted from the instruction set of the target processor despite the fact that syntax variability is more like assembly language than a high-level language.

The algorithm can be visualized through the generated DOT file [81] [82] (see Figure 6.4). The rectangle symbols represent input, output and internal signals and the ovals represent all mathematical operations.

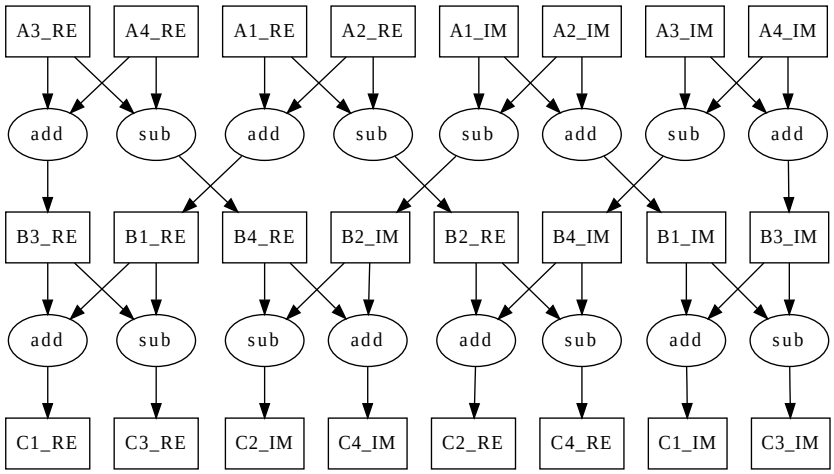

Figure 6.4: Graphical representation of the 4-point FFT

```
    ADD .L1 A1, A3, A9      ; B1_IM = A1_IM + A2_IM
||  SUB .S1 A0, A2, A10     ; B2_RE = A1_RE - A2_RE
||  ADD .D1 A0, A2, A8      ; B1_RE = A1_RE + A2_RE
    ADD .L1 A4, A6, A12     ; B3_RE = A3_RE + A4_RE
||  ADD .S1 A5, A7, A13     ; B3_IM = A3_IM + A4_IM
||  SUB .D1 A1, A3, A11     ; B2_IM = A1_IM - A2_IM
    SUB .L1 A5, A7, A4      ; B4_IM = A3_IM - A4_IM
||  ADD .S1 A8, A12, A0     ; C1_RE = B1_RE + B3_RE
||  SUB .D1 A4, A6, A7      ; B4_RE = A3_RE - A4_RE
    ADD .L1 A10, A4, A2     ; C2_RE = B2_RE + B4_IM
||  SUB .S1 A11, A7, A3     ; C2_IM = B2_IM - B4_RE
||  ADD .D1 A9, A13, A1     ; C1_IM = B1_IM + B3_IM
    SUB .L1 A9, A13, A5     ; C3_IM = B1_IM - B3_IM
||  SUB .S1 A10, A4, A6     ; C4_RE = B2_RE - B4_IM
||  SUB .D1 A8, A12, A4     ; C3_RE = B1_RE - B3_RE
    SUB .D1 A11, A7, A7     ; C4_IM = B2_IM - B4_RE
    NOP
```

Figure 6.5: Generated source code for the 4-point FFT with fixed-point representation

The final code generated for 32-bit fixed-point number representation is shown in Figure 6.5 with the appropriate comments with operations from the original code, where `||` sign marks parallel execution of instructions. The tool mapped the algorithm only into data path A. Due to parallelism, the instructions are executed up to 3 at the same time. The last `NOP` operation is only for filling the last execution cycle when all output data is available in the registers for the next use and can be replaced.

Figures 6.6 and 6.7 show the generated usage maps of the processor resources. The rows of the maps represent the instruction cycle from the beginning of the execution and columns represent the hardware resources (functional units, registers). The map for the functional units (Figure 6.6) contains instructions, which are executed. The light-gray instructions correspond to the assigning operations of the `Cx` signals in the source code (Figure 6.3).

The register map (Figure 6.7) contains the signals assigned to the registers. The gray signals are input for the gray operations in the functional unit map (Figure 6.6). The last instruction cycle is not actually part of the algorithm execution. It is only sign of the prepared signals for next processing.

Cycle	.L1	.S1	.M1	.D1
1	ADD	SUB		ADD
2	ADD	ADD		SUB
3	SUB	ADD		SUB
4	ADD	SUB		ADD
5	SUB	SUB		SUB
6				SUB

Figure 6.6: Functional unit usage in FFT4 (32-bit integer)

Cycle	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
1	A1_RE	A1_IM	A2_RE	A2_IM	A3_RE	A3_IM	A4_RE	A4_IM						
2									B1_RE	B1_IM	B2_RE			
3												B2_IM	B3_RE	B3_IM
4	C1_RE				B4_IM			B4_RE						
5		C1_IM	C2_RE	C2_IM										
6					C3_RE	C3_IM	C4_RE							
Output								C4_IM						

Figure 6.7: Assignment of signals in FFT4 (32-bit integer)

Execution time of the code compiled for 32-bit integer values is 7 instruction cycles (Figure 6.6 and 6.7). Processing of the single precision floating-point data takes 12 instruction cycles (Figure 6.8 and 6.9). In the graphical representation of signal assignment to the registers (Figure 6.7 and 6.9) the last cycle does not belong to the processing, only shows the available data on the output.

The output implementation in the integer data representation uses 3 functional units, because the .M unit has not defined the ADD or SUB operations. For the floating-point data representation, the .D unit is also unused for its incapability of floating-point operations.

The usage of the registers is practically constant during the program execution. It is given by the character of the implemented algorithm, where the input values ale replaced by the same number of the internal variables. The code for the integer data type slightly increases the allocated registers, because the first temporary results are known before the deallocation of the input values. The code for the floating-point data type does not do that, because the floating-point operations take more instruction cycles for its execution. The utilization level of the functional units and registers is showed in Figure 6.10.

Cycle	.L1	.S1	.M1	.D1
1	FADDSP	FADDSP		
2	FSUBSP	FSUBSP		
3	FADDSP	FADDSP		
4	FSUBSP	FSUBSP		
5				
6	FADDSP	FADDSP		
7	FADDSP	FSUBSP		
8	FSUBSP	FSUBSP		
9	FSUBSP	FSUBSP		
10				
11				

Figure 6.8: Functional unit usage in FFT4 (single precision floating-point)

Cycle	A0	A1	A2	A3	A4	A5	A6	A7
1	A1_RE	A1_IM	A2_RE	A2_IM	A3_RE	A3_IM	A4_RE	A4_IM
2								
3								
4	B1_RE	B1_IM						
5			B2_RE	B2_IM				
6					B3_RE	B3_IM		
7							B4_RE	B4_IM
8								
9	C1_RE	C1_IM						
10			C2_RE	C2_IM				
11					C3_RE	C3_IM		
Output							C4_RE	C4_IM

Figure 6.9: Assignment of signals in FFT4 (single precision floating-point)

To compare the compiler output with the hand-written code, the loading and storing data should also be considered. But it will be probably very similar with 4 complex samples on the input and output.

The handling of the higher number of signals was verified with the 8-point FFT. This algorithm also requires the multiplication, not only additions and subtractions like in the 4-point FFT so the use of all functional units of the data path is expected in some instruction cycles. The implementation has 17 input signals, where one of them represents the twiddle factor, 16 represent output signals and 40 internal signals. The graphical representation of the algorithm is shown in Figure 6.11.

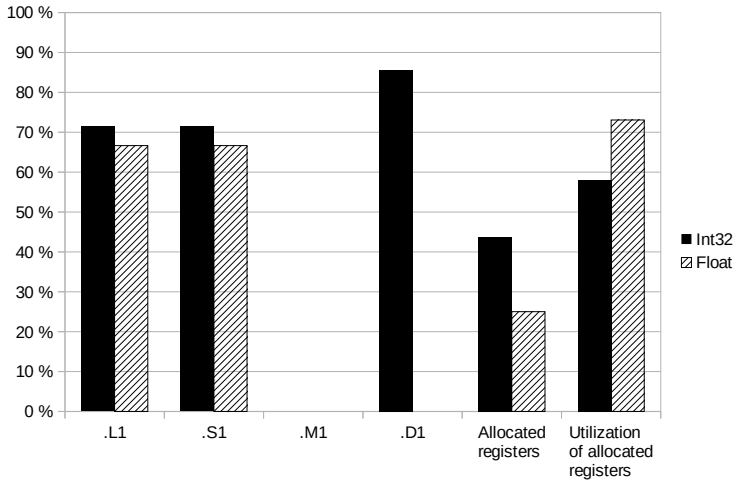

Figure 6.10: Resource utilization of the 4-point FFT

The usage of the functional units for 32-bit integer algorithm is shown in Figure 6.12 and the signal assignment is in Figure 6.13. The structure of the maps is practically the same as in the 4-point FFT case. The registers are allocated effectively despite the fact, that there is no optimization method. The resource utilization can be seen in Figure 6.14.

The tool can handle relatively many signals in previous cases, because there is high-level of parallelism in the algorithm which can be handled by the hardware and signals are quickly deallocated. The number of the internal signals is also approximately constant during the algorithm execution.

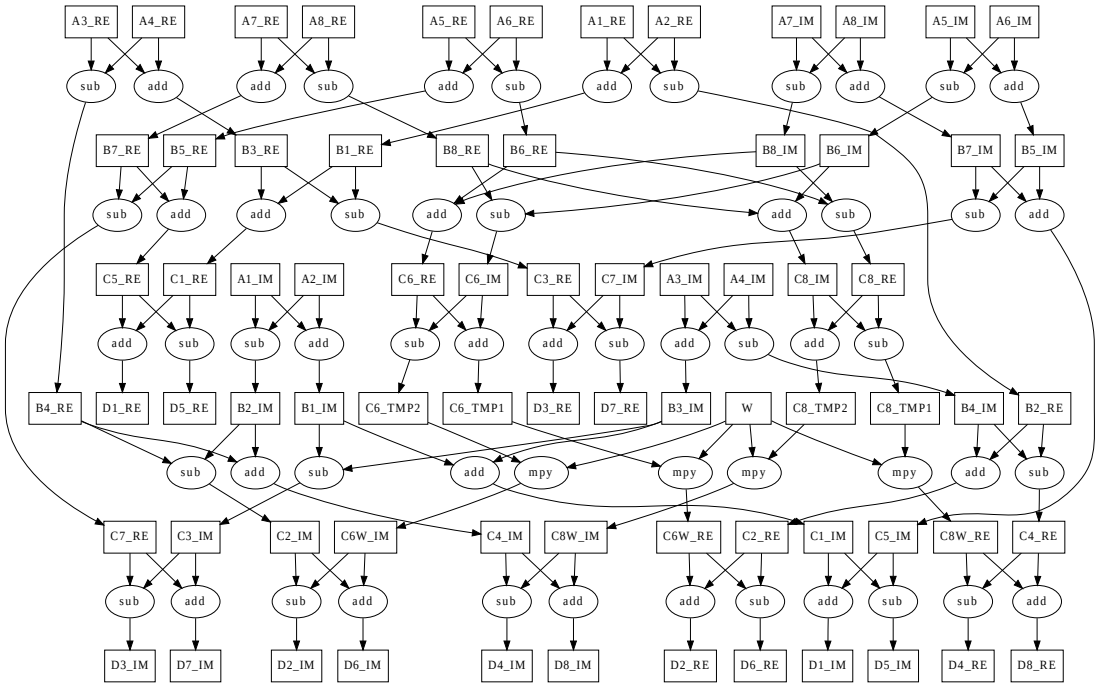

Figure 6.11: Graphical representation of 8-point FFT

Cycle	.L1	.S1	.M1	.D1
1	ADD	SUB		ADD
2	ADD	ADD		SUB
3	SUB	ADD		SUB
4	SUB	SUB		ADD
5	ADD	SUB		ADD
6	ADD	ADD		SUB
7	SUB	SUB		ADD
8	SUB	ADD		SUB
9	ADD	ADD		ADD
10	SUB	SUB		SUB
11	ADD	ADD		SUB
12	SUB	ADD	MPY32	SUB
13	ADD	ADD	MPY32	ADD
14	SUB	SUB	MPY32	SUB
15	ADD		MPY32	SUB
16	SUB			ADD
17	ADD			SUB
18	ADD			SUB
19	ADD			SUB

Figure 6.12: Usage of functional units in FFT8 (32-bit integer)

Cycle	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19
1	A1_RE	A1_IM	A2_RE	A2_IM	A3_RE	A3_IM	A4_RE	A4_IM	A5_RE	A5_IM	A6_RE	A6_IM	A7_RE	A7_IM	A8_RE	A8_IM				
2	B1_RE		B1_IM	B3_RE													B2_RE	B3_IM		
3		B2_IM			B4_RE	B4_IM	B5_RE	B5_IM	B6_RE	B6_IM	B7_RE	B7_IM	B8_RE	B8_IM	C1_RE	C1_IM			C3_RE	W
4																	C5_IM	C6_RE		
5		C3_IM	C4_RE	C2_IM	C4_IM	C5_RE	C6_IM	C7_RE	C8_RE	C8_IM	C7_IM	C6_TM P1								
6	C2_RE						C6_TM P2		C8_TM P1	C8_TM P2			D3_RE	D5_IM	D7_IM	D8_RE				
7		C6W_R E				D3_IM	D1_RE	D6_RE	D5_RE	D6_IM	D7_RE	D1_IM								
8	C6W_I M	D2_RE		D2_IM																
9	C8W_R E		D4_RE		D8_IM															
10	C8W_I M																			
Output	D4_IM																			

Figure 6.13: Assignment of signals in FFT8 (32-bit integer)

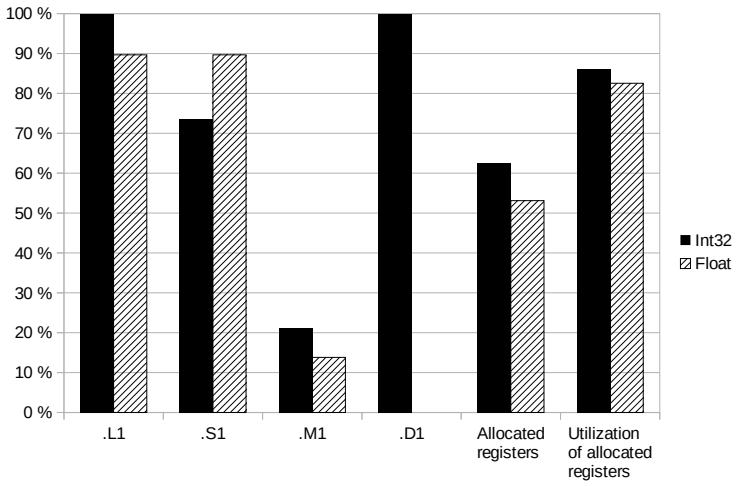

Figure 6.14: Resource utilization of the 8-point FFT

6.1.1.2 Matrix multiplication

The multiplication of two matrices should show the different behavior of the tool, because the target architecture has only one functional unit in data path, which can perform multiplication.

Several matrix multiplication algorithms of different size were implemented for demonstration purpose. The simplest case of the matrix multiplication is the size of 2x2. The source code representing this mathematical operation uses 8 input signals, 4 output

signals and 8 internal signals. Total number of elementary arithmetic operations is 12. The graphical representation is shown in Figure 6.15.

The main operation in the case of the matrix multiplication for TMS320C6678 is the `MPY32` or `MPYSP` depending on the data type. This is the biggest limitation, because this instruction can be executed only by the .M unit. Because the parallel option of the execution is not available, the tool uses the pipelining. The significant difference between integer and floating-point result is the execution time of the algorithm, because of the different number of the instruction cycles needed for the instructions.

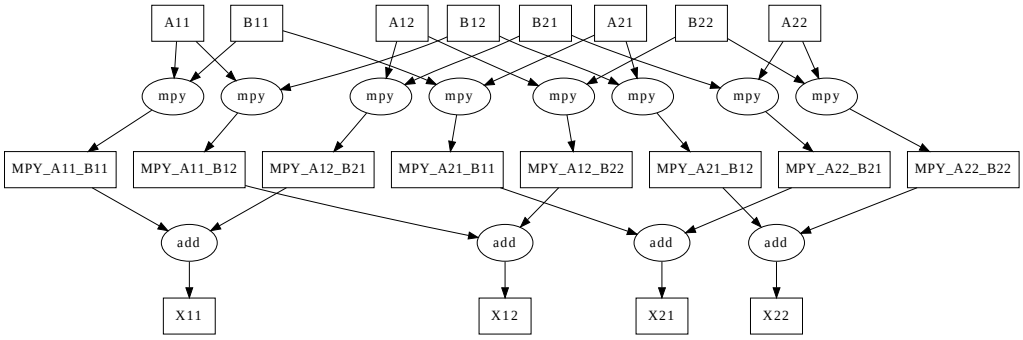

Figure 6.15: Graphical representation of matrix multiplication 2x2

The register utilization is decreasing by the time of the program execution. In both cases, 8 registers are used for the input values at the beginning. At the end, the usage is reduced to 4 registers, which are used for the output values. The other registers could be used for the other purpose. The usage of the hardware resources is shown in Figure 6.16.

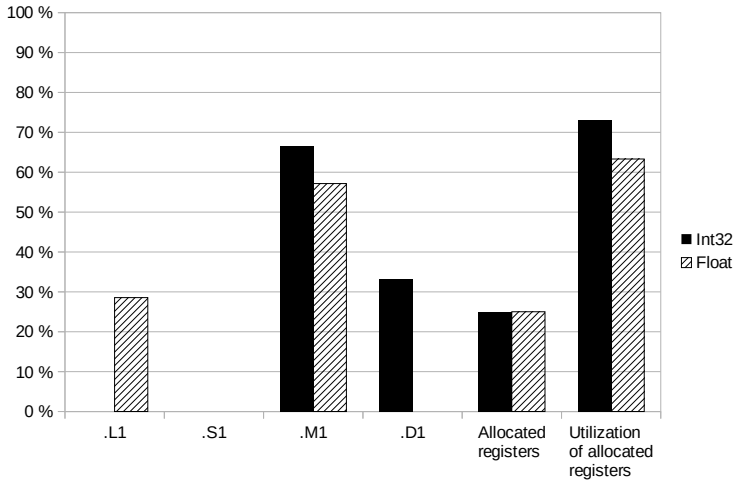

Figure 6.16: Resource utilization for the matrix 2x2 multiplication

The next case is the matrix multiplication of matrix size 3x3. The source code uses 18 input signals, 9 output signals and 36 internal signals. Number of all arithmetic operation is 45. The generated signal flow representation is shown in Figure 6.17.

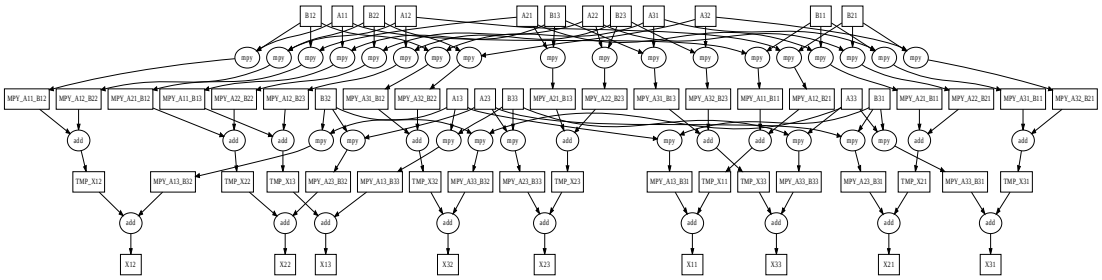

Figure 6.17: Graphical representation of matrix multiplication 3x3

The result for the matrices 3x3 is similar as for 2x2 (Figure 6.18). Because the algorithm consists of more `MPY32` or `MPYSP` operations, the `ADD` or `FADDSP` instructions can be executed in parallel after retrieving results from the first operations. The difference between integer and floating-point output is the execution time of whole algorithm.

The number of used registers is higher due to dimension of the input and output, but the character of the allocated space is the same.

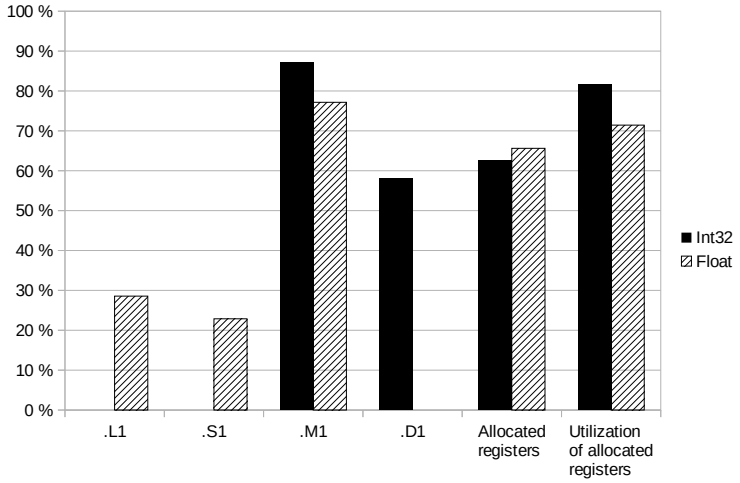

Figure 6.18: Resource utilization for the matrix 3x3 multiplication

The usage of the resources is shown in Table 6.1. There are two types of average usage. The first is for allocated usage, which is computed only for functional units and registers which are used. The second is total usage, which is computed for all resources in data path. The unit usage in the integer cases is higher for two reasons. First is that not all units are able to perform floating-point operations. The second is, that the floating-point takes longer time to execute, so there can be some gaps in the code. The

register usage represents the number of user slots from all registers in the data path, which can be used for data storage during the execution.

Table 6.1: Average hardware resources usage on selected algorithms

Algorithm	Data type	Instruction cycles	Allocated usage [%]		Total usage [%]	
			Functional unit	Registers	Functional unit	Registers
Mat. mpy 2x2	Int32	13	46.15	73.08	23.08	18.27
	Float	15	40.00	63.33	20.00	15.83
	Double	15	40.00	63.33	20.00	31.67
Mat. mpy 3x3	Int32	32	70.31	81.56	35.16	50.98
	Float	36	41.67	71.43	32.25	46.88
FFT4R	Int32	5	80.00	76.00	60.00	23.75
	Float	9	66.67	73.61	33.33	18.40
	Double	10	80.00	78.75	40.00	39.38
FFT4C	Int32	7	76.19	66.33	57.14	29.02
	Float	12	66.67	79.17	33.33	19.79
	Double	12	66.67	79.17	33.33	39.58
FFT8R	Int32	13	73.08	74.23	73.08	46.39
	Float	22	57.58	74.43	43.18	37.22
FFT8C	Int32	20	70.00	86.00	70.00	53.75
	Float	30	62.22	82.55	46.67	43.85

6.1.2 Values stored in memory

The following case counts with the input values stored in the memory. This means that the input of the algorithms is only the pointer to that data. Also, the result will be stored back to the memory, so it will be comparable to the classic high-level language functions.

6.1.2.1 Fast Fourier Transform

The mathematical structure of the algorithm is the same as in the chapter 6.1.1.1. The difference is in the input/output signal definitions. The example of the differences is shown on Figure 6.19.

```
INPUT POINTER X
INPUT POINTER Y

SIGNAL A1_RE
SIGNAL A1_IM
# ... (definition of A2 to A4)

SIGNAL C1_RE
SIGNAL C1_IM
# ... (definition of C2 to C4)

A1_RE = X[0]
A1_IM = X[1]
# ... (load A2 to A4)

Y[0] = C1_RE
Y[1] = C1_IM
# ... (store C2 to C4)
```

Figure 6.19: Difference of the input/output definition

There can be seen that there are only 2 input signals passed to the function, which are pointers to input data and output buffer for result. The input and output signals from Figure 6.2 are now defined as internal signals of the algorithm.

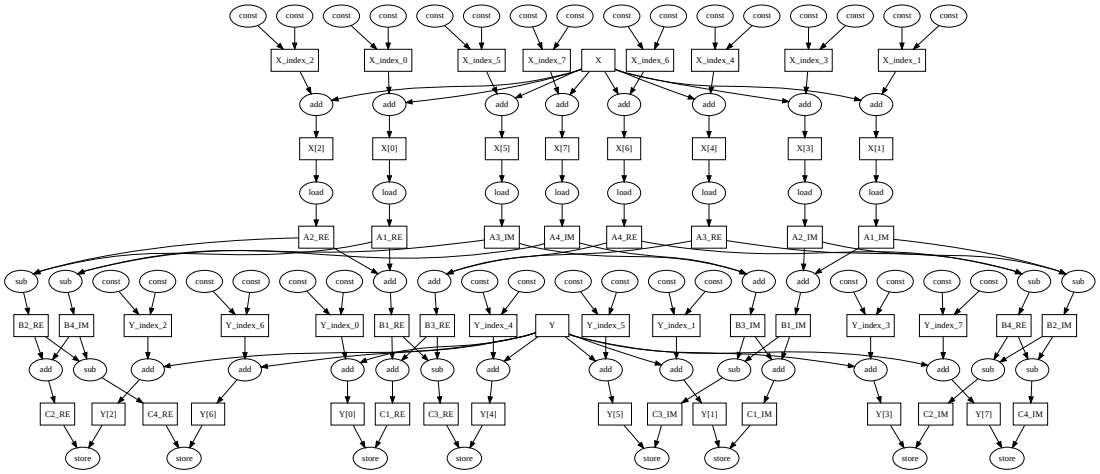

Figure 6.20: Graphical representation of the 4-point FFT with memory operations

Figure 6.20 shows the signal flow diagram of the 4-point FFT with the complex input. Compared to the implementation with the input samples stored in the registers (Figure 6.4) there are much more signals and operations. It is given by the multi-operation nodes (described in chapter 5.3.1.3), which are creating another signals and operations for achieving desired result.

Cycle	.L1	.S1	.M1	.D1
1		MVKL		
2		MVKLH		
3		MVKL		ADD
4		MVKLH		LDW
5		MVKL		ADD
6		MVKLH		LDW
7		MVKL		ADD
8		MVKLH		LDW
9		MVKL		ADD
10		MVKLH		LDW
11		MVKL		ADD
12		MVKLH		LDW
13	ADD	MVKL		ADD
14	SUB	MVKLH		LDW
15	ADD	MVKL		ADD
16	SUB	MVKLH		LDW
17		MVKL		ADD
18		MVKLH		LDW
19		MVKL		ADD
20		MVKLH		
21	ADD	MVKL		ADD
22	ADD	MVKLH		SUB
23	ADD	MVKL		ADD
24	ADD	MVKLH		SUB
25	ADD	MVKL		ADD
26	SUB	MVKLH		SUB
27	SUB	MVKL		ADD
28	SUB	MVKLH		SUB
29		MVKL		ADD
30		MVKLH		STW
31		MVKL		ADD
32		MVKLH		STW
33				ADD
34				STW
35				STW
36				STW
37				STW
38				STW
39				STW

Figure 6.21: Functional unit usage in FFT4 (32-bit integer, data in memory)

The actual operations performed in time are shown on Figure 6.21. The FFT4 algorithm which uses memory as input/output storage space takes 39 instruction cycles to perform. The grayed-out operations in the functional unit usage map are related to the FFT processing, which is from instruction cycle 13 to 28. The rest of the operations are related to the data loading and storing. The true profit in compare to the high-level

language or linear assembly results (Figure 3.9, 3.10) is that the memory operations are performed in parallel with the data processing without unnecessary waiting.

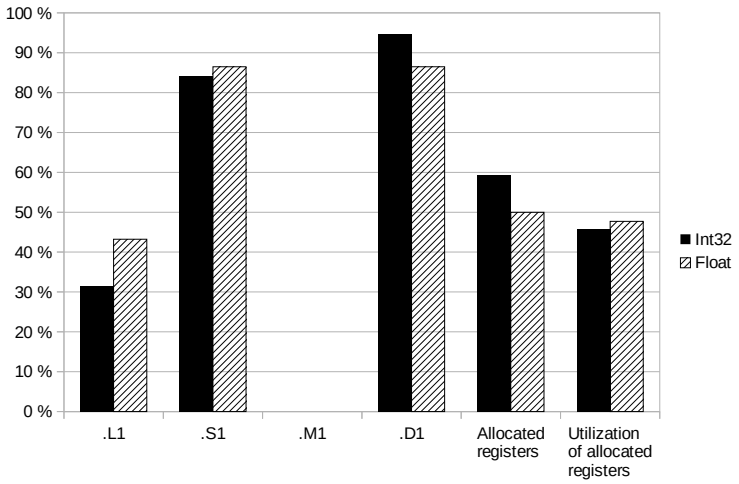

Figure 6.22: Resource utilization of the 4-point FFT with memory operations

6.1.2.2 Matrix multiplication

The algorithm for matrix multiplication was also rewritten for data processing in memory. The modification of the original code is similar as in the previous case. The final generated signal flow is shown on Figure 6.23.

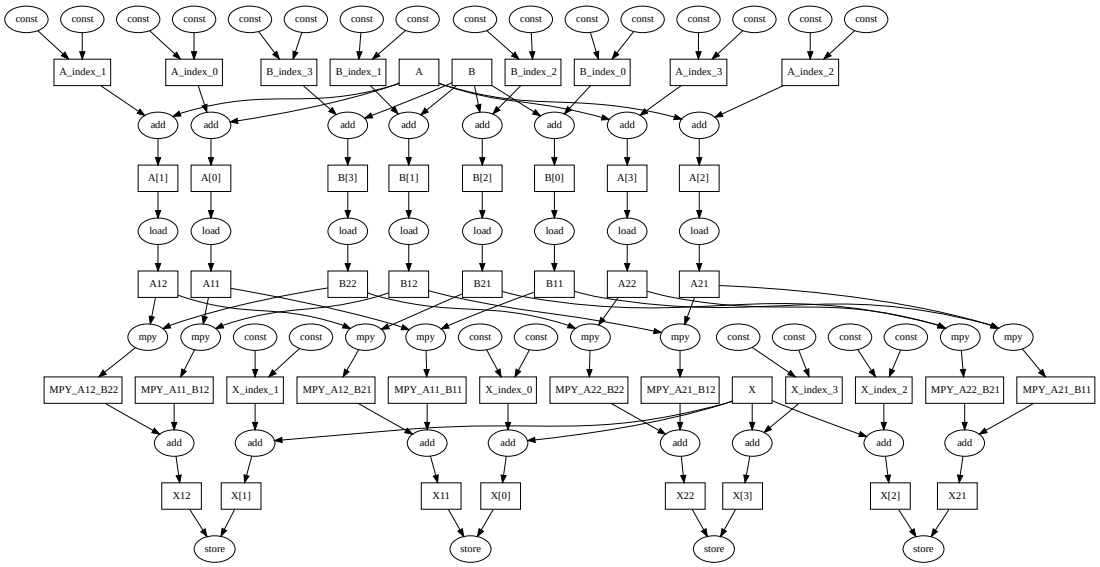

Figure 6.23: Graphical representation of the 2x2 matrix multiplication (data in memory)

The Figure 6.24 shows the functional unit utilization relatively to the total number of instruction cycles of the generated code. In contrast with the previous case of matrix multiplication (Figure 6.16) where the usage of the .M unit was significant because of

the algorithm character, this case has the peaks on .S and .D units. These units are used mainly for the memory access (.S for constant loading, .D for memory access). There can be also seen the ratio between the memory operations and the data processing on the small algorithms.

Table 6.2: Average hardware resources usage on selected algorithms (data in memory)

Algorithm	Data type	Instruction cycles	Allocated usage [%]		Total usage [%]	
			Functional unit	Registers	Functional unit	Registers
Mat. mpy 2x2	Int32	33	45.45	52.27	45.45	19.60
	Float	32	46.88	54.26	46.88	18.65
	Double	52	51.92	44.49	51.92	30.59
Mat. mpy 3x3	Int32	64	59.77	50.48	59.77	41.02
	Float	63	60.71	46.43	60.71	34.82
FFT4R	Int32	33	60.61	51.52	45.45	17.71
	Float	34	58.82	46.32	44.12	17.37
	Double	61	61.20	48.52	45.90	30.33
FFT4C	Int32	39	68.38	28.90	51.28	19.87
	Float	41	65.04	59.27	48.78	18.52
	Double	73	65.75	50.27	49.32	31.42
FFT8R	Int32	63	53.17	54.14	35.17	32.14
	Float	66	50.76	56.86	50.76	30.21
FFT8C	Int32	78	58.97	53.96	58.97	37.10
	Float	85	54.12	53.00	54.12	34.78

Table 6.2 shows the usage of the resource usage of the algorithms which are using the memory access for input and output data. The first main difference from previous test case (Table 6.1) is the number of instruction cycles to perform a data processing. It is given by adding the memory reading and writing, which extends time up to 3-times in compare to the algorithms which are using the registers to passing data to application. Also, there are more free time slots for functional units and registers. It is also caused by data reading, because the processor cannot perform parallel data reading and subsequently data processing at the beginning. The same can be applied at the end of the algorithm with data writing back to the memory.

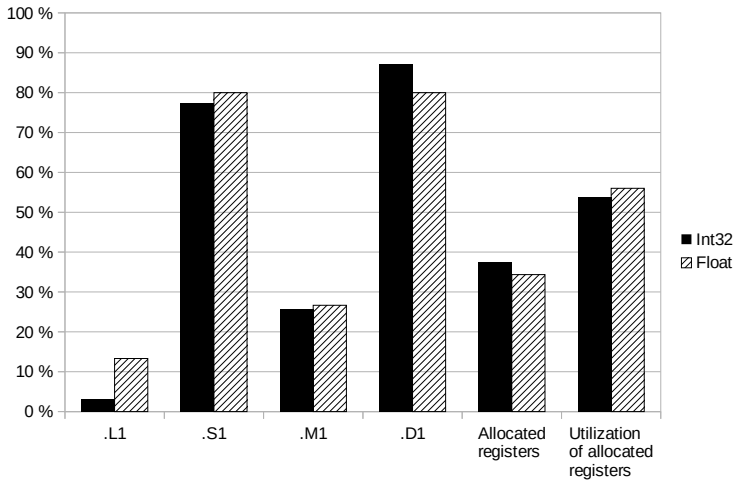

Figure 6.24: Resource utilization for the matrix 2x2 multiplication (data in memory)

6.2 Optimization impact

The previous cases showed results of the instruction mapping without any modification of operation allocation. The tool supports several kinds of priorities during mapping process, which should help to improve generated code. The next part will show the results these methods.

6.2.1 Node priority

The priority of the node mapping can be set to decisions based on the number of functional units or the number of instruction cycles needed to execute assigned instruction. Table 6.3 shows the selected algorithms with memory operations where the methods of node priority mapping were applied. The performance is compared with the result from the previous part with no optimization. The matrix multiplications do not take any benefit of this methods, but FFT algorithms can be executed up to 12 % faster. These top improvements apply on FFT algorithms with floating-point representation and real signal input. The results for algorithm which have input values prepared in registers are not showed, because the execution time of generated codes were the same.

There are two types of algorithms with no, or not so significant improvement. The first type is where the operations have the same features. This is the case of the algorithms with values prepared in registers. The second type are the algorithms where the instructions cannot be easily moved to another functional unit. This is the case of the matrix multiplication. The big part of the instructions performs multiplication which can be done only with .M units. Also the memory operations can be performed only on .D units.

Table 6.3: Node priority mapping improvements (data in memory)

Algorithm	Data type	Instruction cycles			Improvement [%]	
		No priority	Units priority	Cycles priority	Units priority	Cycles priority
Mat. mpy 2x2	Int32	33	33	33	0.00	0.00
	Float	32	32	32	0.00	0.00
	Double	52	54	54	-3.85	-3.85
Mat. mpy 3x3	Int32	64	66	66	-3.13	-3.13
	Float	63	63	63	0.00	0.00
FFT4R	Int32	33	31	30	6.06	9.09
	Float	34	30	30	11.76	11.76
	Double	61	59	59	3.28	3.28
FFT4C	Int32	39	40	40	-2.56	-2.56
	Float	41	39	39	4.88	4.88
	Double	73	70	70	4.11	4.11
FFT8R	Int32	63	59	58	6.35	7.94
	Float	66	58	58	12.12	12.12
FFT8C	Int32	78	76	76	2.56	2.56
	Float	89	77	77	9.41	9.41

On the other side, the algorithms with the highest improvement contains wide variety of instructions. This creates the space for manipulation with the instruction mapping process, but on proposed cases, the results of these two methods are practically same.

6.2.2 Functional unit priority

The next method how to improve the final performance of the code is mapping priority of the functional units. This is based on statistics how many potential operations can be performed on each functional unit. There are two options how the priority is set. The first is the global priority which is given by the number and it is fixed for the architecture. The second is dynamically changing according to remaining unmapped nodes.

The results after applying the priority on mapping were the same as the result without any priority. This was caused by the definition of the architecture, because the original mapping was performed according to this definition. But architecture definition can be different according to the content of the definition file. For that reason, the worst

case was synthetically created by process as the mapping with global priority, but with the opposite conditions.

Table 6.4: Functional unit priority mapping improvements (data in memory)

Algorithm	Data type	Instruction cycles			Improvement [%]	
		Worst case	Global priority	Dynamic priority	Global priority	Dynamic priority
Mat. mpy 2x2	Int32	42	33	30	21.43	28.57
	Float	42	32	32	23.81	23.81
	Double	78	52	52	33.33	33.33
Mat. mpy 3x3	Int32	95	64	59	32.63	37.89
	Float	95	63	63	33.68	33.68
FFT4R	Int32	44	33	33	25.00	25.00
	Float	44	34	34	22.73	22.73
	Double	84	61	61	27.38	27.38
FFT4C	Int32	55	39	39	29.09	29.09
	Float	54	41	41	24.07	24.07
	Double	102	73	73	28.43	28.43
FFT8R	Int32	86	63	63	26.74	26.74
	Float	90	66	66	26.67	26.67
FFT8C	Int32	110	78	78	29.09	29.09
	Float	111	85	85	23.42	23.42

Table 6.4 shows the comparison of the worst case and these two methods of priority mapping. As with the previous methods the algorithms where the input values were prepared in registers, there was no or not significant improvement. For that reason, table shows only implementations with memory operations.

The difference from the previous cases is that the improvement is significant even for the matrix multiplication. The speed-up of the code execution can be relatively high, which is about 25 %. The maximal improvement was for integer matrix multiplication 3x3, with 37 %. This could be unexpected result, because in previous cases this algorithm had slightly worse performance after mapping with priority than the original one.

6.3 Comparison to other methods

The results from the proposed tool were compared with the standard methods of programming. Table 6.5 shows the selected execution times of methods mentioned in the thesis, including data loading and storing into the memory. The hand-written assembly code depends only how it is written. The C code is equivalent of the code passing into the tool's generator. This code was optimized with `-o2` settings. The unoptimized code was about 3 to 4-times slower. The Texas Instrument DSP library for C66x is distributed as static library archives and the change of optimization does not have an effect on the results.

Table 6.5: Comparison of tool results with the standard methods

Algorithm	**Mapping tool**	**ASM code**	**C code**	**TI-DspLib**
FFT4R	34	19	46	-
FFT4C	41	24	80	-
FFT8R	66	34	123	-
FFT8C	85	42	205	145

It can be seen that the hand-written assembly code is achieving the best performance. But the code generated by the tool is executed 2.4-times faster than the compiled C code and 1.7-times faster than the DSP library. The DSP library cannot be compared with the smaller input data, because it has limitation of minimum 8 complex or 32 real values on the input.

6.4 Chapter summary

The proposed instruction mapping tool was tested with several DSP algorithm with possible high-level of parallelism and can be used as cores in the bigger part of the DSP system. These algorithms were mapped into the TMS320C6678, which is VLIW processor with floating-point support.

The first case was FFT and matrix multiplication with different sizes where input values were prepared in registers and output was already written into the registers. These algorithms were mapped with the high density of parallel operations according to possibility of instruction support on the functional units. The generated code was similar to the hand-written code for integer data representation and floating-point data representation as well.

The second case was performed with the same algorithm, but the input and output were placed into the memory, so the difference was in memory access. Because the memory operations cannot be executed on all functional units, these parts of code

presented the bottleneck of the algorithm. But it was only until the algorithms loads the registers for next processing and after that the functional units were utilized as much as possible.

Because the memory access is the bottleneck on the data processing, the tool works only with registers and the memory access is only for data input and output. Thanks to this, the processed data are not stored on stack and the performance can be increased. This approach has also disadvantage. Because the processor core has limited number of the registers, the algorithm cannot be too much complex. The TMS320C6678 can handle relatively lots of data, because it has 4 functional units and 32 registers in single data path. Tested algorithms were also tried to map into the ARM Cortex M4, which has only 16 registers. The tool was not able to map them all, because there is half of the place for data as in the C66x core and the data was processed 4-times slower, so the registers were not freed with the rate as on VLIW architecture.

The tool eliminates the memory access for storing and loading temporary results of the operations and keeps these values in registers. This leads to the speedup of the data processing. But at the same time, this feature is also of its limitation. The tool cannot be used to create complex algorithms. On the other hand, when generated code was compared with the provided optimized libraries, these libraries were limited with the minimal number of input samples. So the tool is destined to be used to optimize parts of algorithms core functions where optimized libraries are not able to handle small input data. Preliminary results were published in [83], [84], [85] and [86].

7 Conclusion

The doctoral thesis was focused on the digital signal processing systems, especially on the software part. The first part of the text introduced the various architectures that can be used for signal processing. It also showed the possibilities of the software realization from the low-level assembly language to the high-level languages with the extensions for parallel processing of the data. For the high-level languages, the basic optimization method which are nowadays used were also introduced.

The second part of the thesis was aimed for software component of the digital signal processing and the new trend which is moving into the parallel data processing. This part practically showed the methods of creating software for parallel architectures from instruction level parallelism to the thread and data parallelism. For this purpose, the DSP TMS320C6678 was chosen because it can handle all of these types of software creating methods. This demonstration showed how the software can affect the final performance of the DSP system. It does not influence only the final execution time, but also the consumed energy.

Data and thread parallelism are good for processing of big amount of data which can be separated into the smaller parts and executed on separated processor cores. But this method is absolutely unsuitable for creating the cores of algorithms itself. This is because the data processing is executed in separated threads which are running on the different cores. This requires the host operating system to create these threads and if necessary, the inter-process communication and synchronization as well. If the core functions of the algorithms will be implemented this way, the overhead of the operating system for threads could be comparable to the processing itself.

The implementation of the DSP core functions is more effective as simple functions. The high-level languages such as ANSI C or low-level assembly language can be used on that purpose. But VLIW architectures, which is also TMS320C6678, are on the market shorter time than the scalar architectures, so the compilers are not so effective. he assembly languages can achieve considerably higher performance. The disadvantage is that the creating software for VLIW architecture requires more concentration.

For that reason, the aim of the thesis is to create a tool which can help to create optimized parts of the code in the assembly language for VLIW processors. This tool is presented in the third part of the thesis. The tool is intended to generate assembly code for desired architecture from abstracted code. The target architecture is not fixed and can be defined by user without tool modification. The tool uses signal-flow graph approach to find the relations between the operations, which are subsequently mapped into the functional units. The mapping of the operations is not linked by the order of the operations in the algorithm definitions as it can be in standard high-level language

compilers. This helps better to find the possible parallel instructions which can be executed on different functional units at the same time. The results can be optionally affected by enabling the automatic consideration of mapping priority which could increase the performance if the generated code. The tool itself is written as console application in C++, which can be compiled on Windows and Unix based systems.

The approach of the tool was verified by several DSP algorithms which can be used as core function of bigger complex algorithm. The tool utilizes the functional units to possible maximum. The performance of generated code was compared to the hand-written assembly code, equivalent C code and DSP library provided by processor vendor. The assembly code has still the best performance, but the generated code exceeded the C code and provided DSP library by the execution time. On the other hand, because the tool uses the memory operations only for getting input data and storing the results to avoid the bottleneck which can be caused by stack access, the tool cannot be used for generating complex functions, but it can be still used for optimizing parts of code with assembly language. These parts can be also reused only by regenerating the code on another architecture, which could not be possible if these optimized parts were written directly.

References

[1] Michael J. Flynn. Very high-speed computing systems. Proceedings of the IEEE. 1966. ISSN: 0018-9219.

[2] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers. 1972. ISSN: 0018-9340.

[3] Manoj Franklin. Computer architecture and organization: From software to hardware. Upper Saddle River: Pearson Education. 2012. ISBN: 0136156703.

[4] Albert Zomaya. Parallel and distributed computing handbook. New York: McGraw-Hill. 1996. ISBN: 0-07-073020-2.

[5] Alex Peleg, Uri Weiser. MMX technology extension to the Intel architecture. Micro, IEEE. 1996. ISSN: 0272-1732.

[6] E. S. Harrison, E. J. Schmitt. The structure of System/88, a fault-tolerant computer. IBM Systems Journal. 1987. ISSN: 0018-8670.

[7] Alan J. George. An overview of RISC vs. CISC. Twenty-Second Southeastern Symposium on System Theory. 1990. ISBN: 0-8186-2038-2.

[8] Jurij Silc, Borut Robic, Theo Ungerer. Processor Architecture: From Dataflow to Superscalar and Beyond. Heidelberg: Springer. 1999. ISBN: 978-3-642-58589-0.

[9] Steven W. Smith. The Scientist & Engineer's Guide to Digital Signal Processing. San Diego: California Technical Pub. 1997. ISBN: 0966017633.

[10] Dake Liu. Embedded DSP Processor Design: Application Specific Instruction Set Processors. Amsterdam: Elsevier. 2008. ISBN: 978-0-12-374123-3.

[11] Haris Javaid, Sri Parameswaran. Pipelined multiprocessor system-on-chip for multimedia. New York: Springer. 2014. ISBN: 978-3-319-01112-7.

[12] Frantz Gene. Digital signal processor trends. IEEE Micro. 2000. ISSN: 0272-1732.

[13] Intel Corporation. Intel® 64 and IA-32 Architectures Developer's Manual. [Online]. 2016. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.

[14] Advanced Micro Devices. AMD64 Architecture Programmer’s, Manual Volume 1: Application Programming. [Online]. 2017. <https://www.amd.com/system/files/TechDocs/24592.pdf>.

[15] Advanced Micro Devices. AMD64 Architecture Programmer’s, Manual Volume 2: System Programming. [Online]. 2018. <https://www.amd.com/system/files/TechDocs/24593.pdf>.

[16] Atmel Corporation. Atmel 8051 Microcontrollers Hardware Manual. [Online]. 2007. <http://ww1.microchip.com/downloads/en/DeviceDoc/doc4316.pdf>.

[17] Microchip Techology Incorporated. PICmicro™ Mid-Range MCU Family Reference Manual. [Online]. 1997. <http://ww1.microchip.com/downloads/en/devicedoc/33023a.pdf>.

[18] Atmel Corporation. AVR Instruction Set Manual. [Online]. 2016. <http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf>.

[19] Lutz Bierl. MSP430 Family Mixed-Signal Microcontroller Application Reports. [Online]. 2000. <http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=slaa024&fileType=pdf>.

[20] ARM Limited. ARM Architecture Reference Manual. [Online]. 2014. <https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf>.

[21] Microchip Technology Incorporated. dsPIC30F Family Reference Manual. [Online]. 2006. <http://ww1.microchip.com/downloads/en/DeviceDoc/70046E.pdf>.

[22] Texas Instruments Incorporated. TMS320C28x Extended Instruction Sets. [Online]. 2015. <http://www.ti.com/lit/ug/spruhs1a/spruhs1a.pdf>.

[23] Texas Instruments Incorporated. TMS320C66x DSP CPU and instruction set reference guide. [Online]. 2010. <http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf>.

[24] Freescale Semiconductor Incorporated. Beyond DSPs: StarCore MSC8xxx and DSP56K Families. [Online]. 2010. <https://www.nxp.com/docs/en/brochure/BYNDDSPBRO.pdf>.

[25] David Blythe. Rise of the Graphics Processor. Proceedings of the IEEE. 2008. DOI 10.1109/JPROC.2008.917718.

[26] Marko J. Misic, Dorde M. Durdevic, Milo V. Tomasevic. Evolution and trends in GPU computing. Proceedings of the 35th International Convention MIPRO. 2012. ISBN: 978-1-4673-2577-6.

[27] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. [Online]. 2012. <https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>.

[28] Timothy G. Rogers, Tor M. Aamodt, Wilson Wai Lun Fung. General-Purpose Graphics Processor Architectures. San Rafael: Morgan & Claypool. 2018. ISBN: 978-1627059237.

[29] NVIDIA Corporation. Graphics Processing Unit (GPU). [Online]. 2019. <https://www.nvidia.com/object/gpu.html>.

[30] Jon Peddie. . New York: Springer. 2013. ISBN: 978-1447149316.

[31] NVIDIA Corporation. NVIDIA Tesla V100 GPU Accelerator. [Online]. 2018. <https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf>.

[32] Philips Incorporated. An Introduction to Very Long Instruction Word Computer Architecture. 1997. Pub# 9397-750-01759.

[33] Cliff Young, Joseph A. Fisher, Paolo Faraboschi. Embedded Computing. Amsterdam: Elsevier. 2005. ISBN: 978-1-4933-0365-6.

[34] John Paul Shen. Modern processor design: fundamentals of superscalar processors. Long Grove: Waveland Press. 2013. ISBN: 9781478607830.

[35] Barney Blaise. Introduction to Parallel Computing. [Online]. 2016. <https://computing.llnl.gov/tutorials/parallel_comp/>.

[36] Randall Hyde. The Art of Assembly Language. San Francisco: No Starch Press. 2003. ISBN: 978-1886411975.

[37] Agner Fog. Optimizing subroutines in assembly language: An optimization guide for x86 platforms. [Online]. 2018. <https://www.agner.org/optimize/optimizing_assembly.pdf>.

[38] Rulph Chassaing, Donald S. Reay. Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK. Hoboken: Wiley-Interscience. . ISBN: 978-0-470-13866-3.

[39] ISO/IEC 1539-1:2010, Information technology - Programming languages - Fortran - Part 1: Base language

[40] Ecma International. ECMA-55 Minimal BASIC, 1st edition. [Online]. 1978. <http://www.ecma-international.org/publications/files/ECMA-STWITHDRAWN/ECMA-55,%201st%20Edition,%20January%201978.pdf>.

[41] Ecma International. ECMA-116 BASIC, 1st edition. [Online]. 1986. <http://www.ecma-international.org/publications/files/ECMA-STWITHDRAWN/ECMA-116,%201st%20edition,%20June%201986.pdf>.

[42] ISO/IEC 9899:2011, Information technology - Programming languages - C

[43] ISO/IEC 14882:2014, Information technology - Programming languages - C++

[44] ISO/IEC 23270:2006, Information technology - Programming languages - C#

[45] James Gosling et al. The Java Language Specification, Java SE 8 Edition. [Online]. 2015. <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>.

[46] Keith Cooper, Linda Torczon. Engineering a Compiler. San Francisco: Morgan Kaufmann. 2012. ISBN: 978-0120884780.

[47] Texas Instruments Incorporated. TMS320C67x DSP library programmer's reference guide. [Online]. 2010. <http://www.ti.com/lit/ug/spru657c/spru657c.pdf>.

[48] ARM Limited. CMSIS - Cortex microcontroller software interface standard. [Online]. 2016. <http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php>.

[49] Microchip Technology Incorporated. DSP library for PIC32. [Online]. 2016. <http://www.microchip.com/SWLibraryWeb/product.aspx?product=DSP%20Library%20for%20PIC32>.

[50] M. Frigo, S. G. Johnson. The design and implementation of FFTW3. Proceedings of the IEEE. 2005. doi: 10.1109/JPROC.2004.840301.

[51] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. [Online]. 2014. <http://www.openmp.org/>.

[52] Open MPI Project. Open MPI: Open Source High Performance Computing. [Online]. 2014. <http://www.open-mpi.org/>.

[53] Edward Kandrot, Jason Sanders. Cuda by Example: an Introduction to General-Purpose GPU. Upper Saddle River, NJ: Addison-Wesley. 2014. ISBN: 978-0131387683.

[54] S. Rajagopalan, S. P. Rajan, S. Malik, S. Rigo, G. Araujo, K. Takayama. A retargetable VLIW compiler framework for DSPs with instruction-level parallelism. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2001. doi: 10.1109/43.959861.

[55] Des Watson. A Practical Approach to Compiler Construction. New York: Springer. 2017. ISBN: 978-3-319-52789-5.

[56] Robert W. Sebesta. Concepts of programming languages. Boston: Pearson. 2012. ISBN: 978-0-13-139531-2.

[57] William von Hagen. The Definitive Guide to GCC. Berkeley: Apress. 2006. ISBN: 978-1-59059-585-5.

[58] Texas Instruments Incorporated. TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide. [Online]. 2010. <http://www.ti.com/lit/ug/spru732j/spru732j.pdf>.

[59] Texas Instruments Incorporated. TMS320C66x CorePac user guide. [Online]. 2013. <http://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf>.

[60] Texas Instruments Incorporated. TMS320C6678 Multicore Fixed and FloatingPoint Digital Signal Processor. [Online]. 2014. <http://www.ti.com/lit/gpn/tms320c6678>.

[61] Advantech Limited. TMDSEVM6678L EVM Technical Reference Manual. [Online]. 2012. <http://wfcache.advantech.com/support/DSPM8301E_EVM%20(6678)3.0/TMDSEVM6678L_Technical_Reference_Manual_2V01_0320.pdf>.

[62] Barney Blaise. OpenMP. [Online]. 2013. <https://computing.llnl.gov/tutorials/openMP/>.

[63] Sen M. Kuo, Bob H. Lee. Real-time digital signal processing. New York: Wiley & Sons. 2001. ISBN: 0-470-84137-0.

[64] James W. Cooley, John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation. 1965. doi: 10.2307/2003354.

[65] Texas Instruments Incorporated. SYS/BIOS (TI-RTOS Kernel) v6.45 User's Guide. [Online]. 2015. <http://www.ti.com/lit/ug/spruex3p/spruex3p.pdf>.

[66] Texas Instruments Incorporated. TMS320 DSP/BIOS v5.42 User's Guide. [Online]. 2015. <http://www.ti.com/lit/ug/spru423i/spru423i.pdf>.

[67] Roman Mego, Tomas Fryza. Performance of parallel algorithms using OpenMP. 23rd International Conference Radioelektronika. 2013. ISBN: 978-14673-5516-2.

[68] Texas Instruments Incorporated. TMS320C6000 programmer's guide. [Online]. 2011. <http://www.ti.com/lit/ug/spru198k/spru198k.pdf>.

[69] Steven A. Tretter. Communication system design using DSP algorithms with laboratory experiments for the TMS320C6713 DSK. New York: Springer. 2008. ISBN: 978-0-387-74886-3.

[70] Tomas Fryza, Roman Mego. Low level source code optimizing for single/multi/core digital signal processors. 23rd International Conference Radioelektronika. 2013. ISBN: 978-1-4673-5516-2.

[71] Tomas Fryza, Roman Mego. Frequency Domain FIR Filter Optimization for Multi-core C6678 DSP. 26th International Conference Radioelektronika. 2016. ISBN: 978-1-5090-1674-7.

[72] Tomas Fryza, Roman Mego. Power Consumption of Multicore Digital Signal Processor: Theoretical Analysis and Real Applications. Proceedings of the 2014 IEEE 23rd International Symposium on Industrial Electronics. 2014. ISBN: 978-1-4799-2399-1.

[73] Ecma International. ECMA-404 The JSON Data Interchange Format, 1st Edition. [Online]. 2013. <http://www.ecmainternational.org/publications/files/ECMA-ST/ECMA-404.pdf>.

[74] World wide web consortium (W3C). Extensible markup language (XML) 1.0. [Online]. 2008. <https://www.w3.org/TR/REC-xml/>.

[75] Dave Gamble. cJSON. [Online]. 2013. <http://cjson.sourceforge.net/>.

[76] The GTK Team. The GTK Project. [Online]. 2019. <https://www.gtk.org/>.

[77] Qt Group. Qt: Cross-platform software development for embedded and desktop. [Online]. 2019. <https://www.qt.io/>.

[78] Roman Mego, Tomas Fryza. Tool for Algorithms Mapping with Help of Signal-Flow Graph Approach. 24th International Conference Radioelektronika. 2014. ISBN: 978-1-4799-3713-4.

[79] Roman Mego. Processor Model for the Instruction Mapping Tool. Proceedings of the First PhD Symposium on Sustainable Ultrascale Computing Systems (NESUS PhD 2016). 2016. ISBN: 978-84-608-6309-0.

[80] Roman Mego. Instruction mapping process on the VLIW architectures. Proceedings of the 22nd conference Student EEICT. 2016. ISBN: 978-80-214-5350-0.

[81] Emden Gansner, Eleftherios Koutsofios, Stephen North, Kiemphong Vo. A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering. 1993. doi: 10.1109/32.221135.

[82] Emden Gansner, Eleftherios Koutsofios, Stephen North. Drawing graphs with dot. [Online]. 2006. <http://www.graphviz.org/Documentation/dotguide.pdf>.

[83] Roman Mego, Tomas Fryza. Efficiency of the Signal Processing Algorithms Using Signal-Flow Based Mapping Tool. Efficiency of the Signal Processing Algorithms Using Signal-Flow Based Mapping Tool. 2015. ISBN: 978-1-4799-8117-5.

[84] Tomas Fryza, Roman Mego. Advanced Mapping Techniques for Digital Signal Processors. 16th IEEE International Symposium on Signal Processing and Information Technology. 2016. ISBN: 978-1-5090-2902-0.

[85] Tomas Fryza, Roman Mego. Instruction-level Programming Approach for Very Long Instruction Word Digital Signal Processors. Proceedings of the 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2017). 2018. ISBN: 978-1-5386-1911-7.

[86] Roman Mego, Tomas Fryza. A Tool for VLIW Processors Code Optimizing. Proceedings of the 13th International Conference on Computer Engineering and Systems (ICCES 2018). 2019. ISBN: 978-1-5386-5111-7.

Curriculum vitae

Roman Mego

Technicka 3082/12
616 00 Brno
Czech Republic

E-mail: roman.mego@vutbr.cz

Research interests

Control, communication and signal processing applications in embedded systems and its optimization.

Education

since 2012 Brno University of Technology, Brno, Czech Republic

- Doctor of Philosophy (Ph.D.), Electronics and Communication
- Thesis: Parallelism in digital signal processing

2010 - 2012 Brno University of technology, Brno, Czech Republic

- Master’s degree (Ing.), Electronics and Communication
- Thesis: RFID based access system in rooms

2007 - 2010 Brno University of Technology, Brno, Czech Republic

- Bachelor’s degree (Bc.), Electronics and Communication
- Thesis: PC oscilloscope - hardware part

Academic appointments

- 2012 - 2017 Department of Radio Electronics, Brno, University of Technology
- 2014 - 2016 Research assistant in communication systems (PEKOS) projects
- 2015 - 2017 Research assistant in Systems for effective hardware modeling and software mapping

Computer skills

Programming languages

- C/C++, C# - Advance
- VHDL – Intermediate
- MatLab, GNU Octave – Intermediate

CAD systems

- KiCad, Eagle – Advance
- FreeCAD, AutoCad – Intermediate

Documents and graphics editors

- MS Office, Libre Office – Advanced
- Gimp, Inkscape, RawTherapee – Intermediate

Others

- Linux server administration – Intermediate

Experience	• since 2011 ModemTec – Research and development, embedded system design, signal processing and communication • 6/2012 - 8/2012 Freescale Semiconductor – student internship • 2005 - 2006 DcaLaser – CNC programming and technical documentation conversion
Language skills	• Slovak – Native speaker • English – Intermediate

List of publications

2020

Ladislav Stastny, Roman Mego, Josef Pihera, Jaroslav Hornak. Selectivity of inductive coupling for partial discharge measurement in MV cables. International Conference on Diagnostics in Electrical Engineering (Diagnostika 2020). 2020. ISBN: 978-1-7281-5879-2 (submitted for publication).

2019

Roman Mego, Tomas Fryza. A Tool for VLIW Processors Code Optimizing. In Proceedings of the 13th International Conference on Computer Engineering and Systems (ICCES 2018). 2019. doi: 10.1109/ICCES.2018.8639186.

2018

Tomas Fryza, Roman Mego. Instruction-level Programming Approach for Very Long Instruction Word Digital Signal Processors. In Proceedings of the 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2017). 2018. doi: 10.1109/ICECS.2017.8292060.

2017

Bedrich Benes, Roman Mego. Narrowband Power Line Communication Over Medium Voltage: An Excellent Tool for Line Diagnostics. Metering & Smart Energy International. Issue 4. 2017. ISSN: 1025-8248.

2016

Tomas Fryza, Roman Mego. Advanced Mapping Techniques for Digital Signal Processors. In 16th IEEE International Symposium on Signal Processing and Information Technology. 2016. doi: 10.1109/ISSPIT.2016.7886037.

Ladislav Stastny, Roman Mego, Lesek Franek, Zdenek Bradac. Zero Cross Detection Using Phase Locked Loop. In 14th IFAC Conference on Programmable Devices and Embedded Systems - PDeS 2016. IFAC-PapersOnLine (ELSEVIER). 2016. doi: 10.1016/j.ifacol.2016.12.050.

Roman Mego. Implementation of retargetable configurable CORDIC algorithm for FPGA devices. Sbornik prispevku studentske konference Blansko 2016. 2016. ISBN: 978-80-214-5389-0.

Tomas Fryza, Roman Mego. Frequency Domain FIR Filter Optimization for Multi-core C6678 DSP. In 26th International Conference Radioelektronika. 2016. doi: 10.1109/RADIOELEK.2016.7477430.

Roman Mego. Instruction mapping process on the VLIW architectures. In Proceedings of the 22nd conference Student EEICT. 2016. ISBN: 978-80-214-5350-0.

Roman Mego. Processor Model for the Instruction Mapping Tool. In Proceedings of the First PhD Symposium on Sustainable Ultrascale Computing Systems (NESUS PhD 2016). 2016. ISBN: 978-84-608-6309-0.

2015

Roman Mego, Tomas Fryza. Efficiency of the Signal Processing Algorithms Using Signal-Flow Based Mapping Tool. In Proceedings of 25th International Conference Radioelektronika 2015. 2015. doi: 10.1109/RADIOELEK.2015.7129035.

Roman Mego. Hi-Speed USB Communication with FPGA. In Sbornik prispevku studentske konference Kohutka 2015. 2015. ISBN: 978-80-214-5239-8.

2014

Tomas Fryza, Roman Mego. Power Consumption of Multicore Digital Signal Processor: Theoretical Analysis and Real Applications. In Proceedings of the 2014 IEEE 23rd International Symposium on Industrial Electronics. 2014. doi: 10.1109/ISIE.2014.6864904.

Roman Mego, Tomas Fryza. Tool for Algorithms Mapping with Help of Signal-Flow Graph Approach. In Proceedings of 24th International Conference Radioelektronika 2014. 2014. doi: 10.1109/Radioelek.2014.6828429.

Roman Mego. Behavior of hardware acceleration on real-time operating systems. In Sbornik prispevku studentske konference Zvule 2014. 2014. ISBN: 978-80-214-5005-9.

2013

Roman Mego. Tomas Fryza. Performance of parallel algorithms using OpenMP. In Microwave and Radio Electronics Week MAREW 2013. 2013. doi: 10.1109/RadioElek.2013.6530923.

Tomas Fryza, Roman Mego. Low Level Source Code Optimizing for Single/Multi-core Digital Signal Processors. In MAREW 2013. 2013. doi: 10.1109/RadioElek.2013.6530933.

2012

Roman Mego. RFID Access Terminal. In Proceedings of the 18th conference Student EEICT. 2012. ISBN: 978-80-214-4461-4.

Roman Mego, Tomas Fryza. RFID pristupovy terminal. Elektrorevue - Internetovy casopis (http://www.elektrorevue.cz). 2012. ISSN: 1213-1539.