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 – SHORT VERSION
DIZERTAČNÍ PRÁCE – ZKRÁCENÁ VERZE

AUTHOR AUTOR PRÁCE	Ing. Roman Mego
SUPERVISOR ŠKOLITEL	doc. Ing. Tomáš Frýza, Ph.D.

BRNO 2020

Keywords

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

Klíčová slova

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

Rukopis dizertační práce je uložen na:

Ústav radioelektroniky
Fakulta elektrotechniky a komunikačních technologií
Vysoké učení technické v Brně
Technická 3082/12
616 00 Brno

Contents

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. This group might include classic scalar architectures such as complex instruction set computers (CISC) [2] or reduced instruction set computers (RISC) [3]. The advantage is the simplicity of implementation, which requires only one functional unit, and low requirements in software design [4].

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. Vector and matrix operations are typical for this group, so the processors are sometimes called the vector processors [4]. 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.

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, what reduces the chance of the errors. Except this feature, it provides no benefit like the increase of the computing power [4].

1.1.4 Multiple instructions, multiple data (MIMD)

MIMD systems use several mutually independent functional units, which can handle different data. 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 [4].

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. 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. This gave the opportunity to create the RISC, which makes the ALU smaller, the execution of instructions faster, and the compilers could be better optimized [5].

Classic processors CISC and RISC are adjusted mainly for control applications. Average application of this type performs branch operation on every 7th instruction [6]. 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 multiply and accumulate (MAC), fused multiply-add (FMA), vector operations or saturated arithmetic [7][8][9]. For this reason, digital signal processors (DSP) were created with similar architecture to the RISC processors. The DSPs can use the idea of separate buses for data and instructions from Harvard architecture. what increases throughput [10].

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 [11]. 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 [12].

1.2.3 Very long instruction word (VLIW)

Core of the processor based on VLIW [13][14] 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 [15]. 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.

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. 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 [16].

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. 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 [16].

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. These systems can be scaled to the desired application respecting the advantages and disadvantages of the combined systems.

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. This subsection will introduce some methods of creating software.

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 [17] [18].

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 [19], BASIC [20][21], C [22], C++ [23], C# [24] or Java [25]. 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 [26], 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 [27][28][29] or by the third party [30], compiler extensions, such OpenMP [31] for program execution on shared memory system or MPI [32] for distributed memory system or with special programming languages like CUDA [33] for general-purpose processing on GPU. There are also some projects such as [34] that are able to handle the instruction level parallelism more effective.

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 [35]:

- 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 [26] and [35]. The well known methods include:

- redundancy elimination
- constant propagation optimization
- useless code elimination
- inline expansion

2 The objectives of the dissertation thesis

There are many possibilities how to realize digital signal processing systems. 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 [36] cores, which is older series supporting only fixed-point arithmetic, and with the C66x [37] 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 [38] 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 [37]. Each core consists of two data paths, two sets of thirty-two 32-bit registers, and two sets of four functional units. Each functional unit is primary used for a different type of operations. In addition to standard operations, 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.) [39]. 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 [38].

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

OpenMP [31] 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 [40].

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.

FIR filter is implemented according to (3.1) from [41]. 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.

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

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 [41], 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.

For the demonstration of FFT, the Cooley-Tukey algorithm [42] 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 3.1 schematically shows progress of used loops in algorithm.

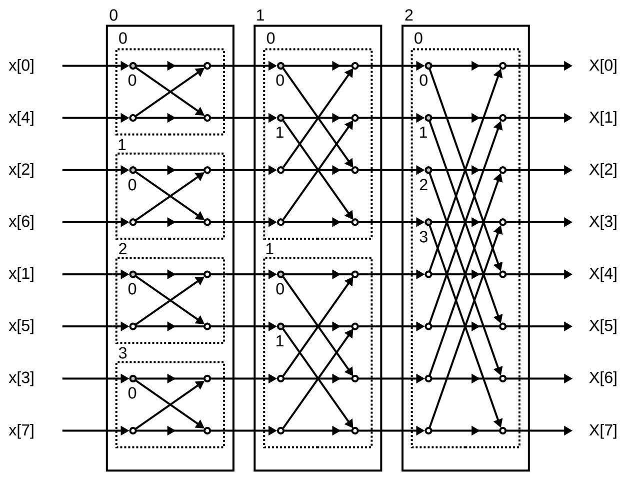


Figure 3.1: FFT radix-2 with highlighted loop iterations

Final parallel code cannot run without operating system, which controls threads. TI provides real-time kernel called SYS/BIOS [43] or DSP/BIOS [44]. It is designed to use in embedded applications which requires real-time scheduling.

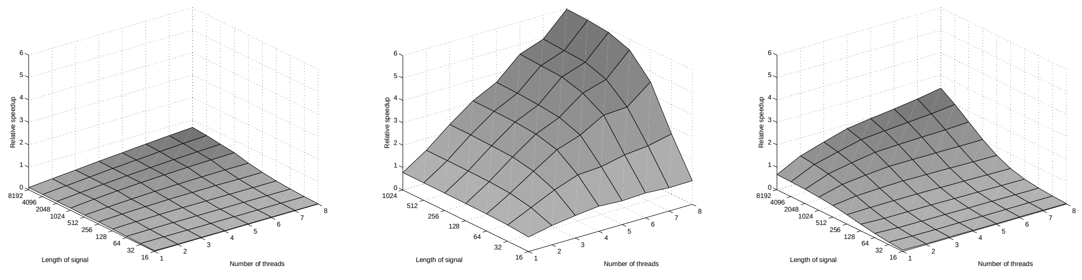
3.2.3 Measured performance of OpenMP

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 3.1).

Table 3.1: 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

Figure 3.2 shows 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 3.1. 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.



a) *b)* *c)*

Figure 3.2: Relative speedup of a) FIR filter, b) DFT, c) FFT

Table 3.2 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 3.2: 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 [45]. 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 3.3 summarizes the computing demands of functions written in C language and low-level assembly as well. The function name FFT4R represents a function for real FFT with $N = 4$, FFT16C is the function for complex FFT with $N = 16$, etc.

Table 3.3: FFT implementation performance comparison

Function	Input	C implementation		Low-level		Relative speedup
		Data path	**CPU cycles**	**Data path**	**CPU cycles**	
FFT4R	4-point real	A+B	46	A	19	2.42
FFT4C	4-point complex	A+B	80	A	24	3.33
FFT8R	8-point real	A+B	123	A	34	3.62
FFT8C	8-point complex	A+B	205	A	42	4.88
FFT16R	16-point real	A+B	425	A	88	4.83
FFT16C	16-point complex	A+B	642	A	100	6.42

The C 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.

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 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.

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. The part of low-level implementation of the FFT is shown in Figure 3.3. The first ADDSP (single precision floating-point addition, see [39]) operation is the equivalent of the first addition operation of the C code from Figure 3.6.

```
    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 3.3: Hand-written assembly code

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 [46]. This feature should help to reduce developing time [47].

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 3.4.

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

Figure 3.4: Example of linear assembly code

In the disassembly form of the example code (Figure 3.5) 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 [39]. 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 3.5: 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 is shown in Figure 3.6.

```
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 3.6: Disassembly of the FFT algorithm written in C

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 that the code is executed mostly sequentially, one instruction after the other. The other issue is the frequent access to the memory. Other information about usage of the functional units can be found in [48].

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 3.4: 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 3.4 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 [30], 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 [27]. 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.

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. The proposed test cases are as follows:

- Empty loop
- Load/Store operations
- Fixed-point operations
- Floating-point operations
- FFT routines

4.3 Experimental Results

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 4.1.

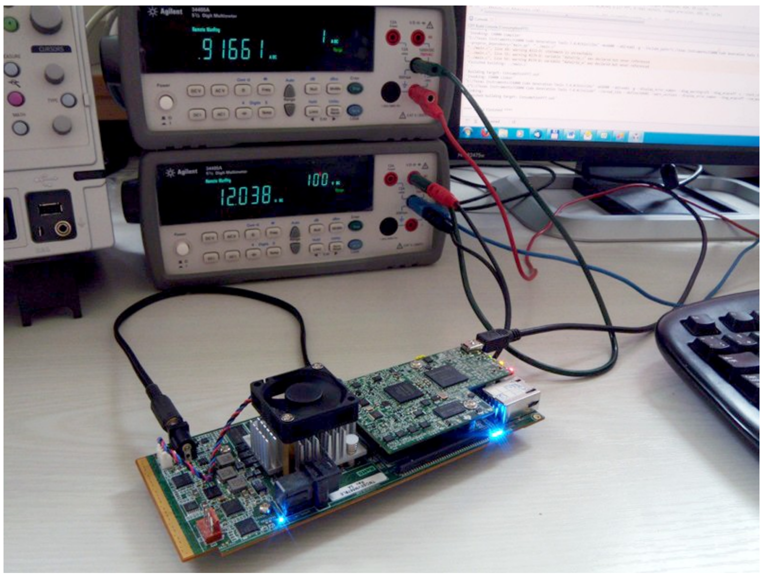


Figure 4.1: Workplace for the measuring the power consumption

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 4.2, 4.3 and 4.4 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.

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".

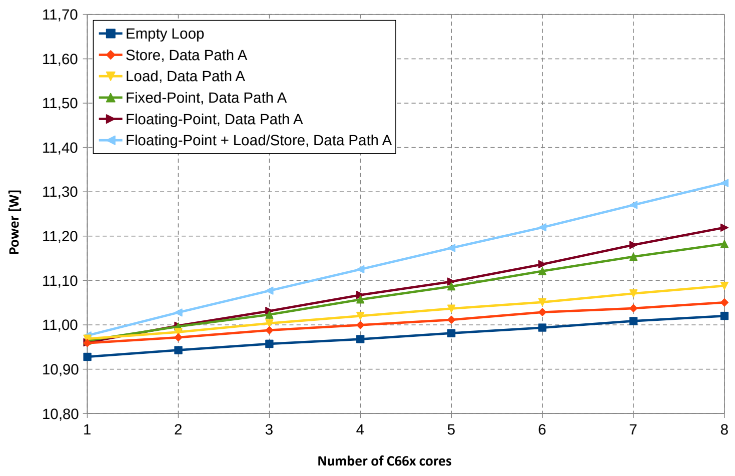


Figure 4.2: Power consumption of theoretical test cases at data path A

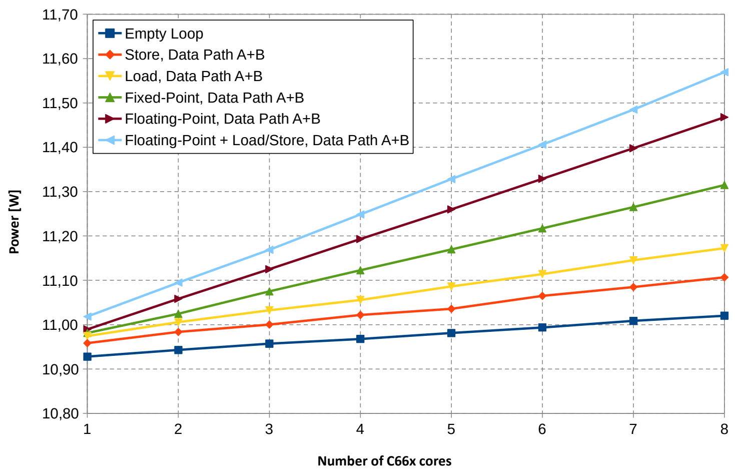


Figure 4.3: Power consumption of theoretical test cases at data paths A and B

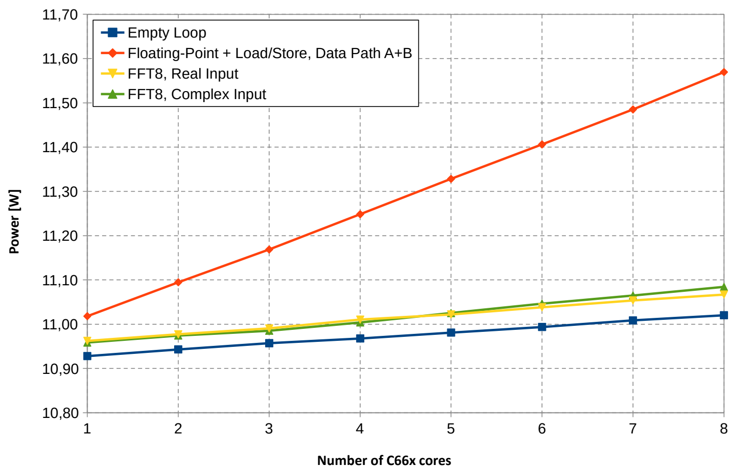


Figure 4.4: Power consumption of FFT routines at data paths A and B

5 Instruction mapping tool for DSPs

The following section presets the functionality of the proposed approach. The purpose of the technique is to ease the optimization process of the signal processing algorithm by generating the low-level assembly code. The user defined code is independent on the target architecture, so it could be reused in different projects.

The first part of the following text shows the definitions of the input data, which contains target architecture and algorithm description. The second part describes the mapping process itself.

5.1 Architecture definition

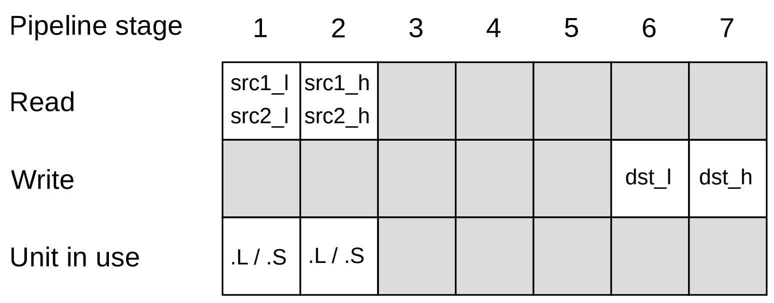
The target processor architecture is one of two input information needed to generate low-level assembly code. The architecture is stored in JavaScript Object Notation (JSON) format [49]. The stored object is divided into two parts. The first part consists of the structure with available resources, the second is the list of supported instructions.

The hardware resources model is based on TMS320C6678. It is a multicore fixed and floating-point digital signal processor (DSP) by Texas Instruments (TI), integrating eight C66x cores [37]. Each core consists of two identical data paths A and B, where each data path contains four functional units and thirty-two 32-bit general purpose registers. The functional units .L, .S, .M and .D are not equal, and every unit is designed for a different purpose and supports different instructions.

Each data path is defined by functional units and registers. The registers can be joined into the groups representing the data types supported by instructions.

The instruction set is given for the whole architecture. There are three types of instructions. The first is arithmetic instruction for basic mathematic operations. The second is memory instruction for data loading into the registers or data movement into the memory. The last is a general function. It is intended for operations which do not fit into the previous categories.

Each instruction has defined its format, function, timing, supported functional units. Arithmetic operations have also defined supported data types. Most of the parameters need not be explained in detail, except of the timing. The timing is described by the number of instruction cycles needed in 3 stages of pipelining. The example is shown in Figure 5.1.



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 5.1: Execution progress of ADDDP instruction

During the execution of `ADDDP` [39] (double precision floating-point addition) instruction, the functional unit .L (or .S) is fully utilized in the first two cycles and cannot be used to start an execution of the next instruction. In addition, it reads input arguments from 2 register pairs, so they cannot be overwritten with other values. From the 3rd to the 5th instruction cycle, there are no extra requirements for the registers or the functional units. In that time, they can be used for other purposes. In the last two instruction cycles, the result is stored back into the registers, so the values previously stored there should not be needed anymore. The results can be used in the 8th instruction cycle here.

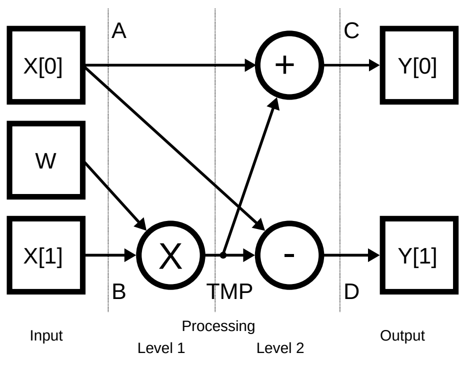
5.2 Algorithm description

As mentioned, the proposed method uses the signal-flow graph-based approach, which is similar to the HDL. The algorithm description contains two basic elements, signals and nodes.

The signal is equivalent to the variable in C language. The difference is that the variable in standard high-level languages can be reassigned multiple times whereas in the tools description it can be assigned only once.

The second element in the algorithm description is the node. It is practically the operation on the signals. The example of the operation can be algebraic operation, memory loading/storing or constant definition.

Figure 5.2 shows the graphical representation of the example code. It has 3 signals `X`, `Y` and `W` which can be compared to the input arguments of the function in C language. Signal `X` is pointer to array with input values, signal `Y` is pointer to output array and `W` is other input parameter. There are also signals `A`, `B`, `C`, `D` and `TMP` which are used for temporary results. Note, the operations are not needed to be written in the same order as they should be processed.



```
INPUT POINTER X
INPUT POINTER Y
INPUT W
SIGNAL A
SIGNAL B
SIGNAL C
SIGNAL D
SIGNAL TMP

A = X[0]
B = X[1]
Y[0] = C
Y[1] = D
C = A + TMP
D = A - TMP
TMP = B * W
```

Figure 5.2: Signal-flow diagram from example algorithm

5.3 Mapping process

The goal of the mapping process is to assign operations from the algorithm to the hardware resources of the target processor. The process begins by the parsing of input files describing both the algorithm and the target architecture. The parsed algorithm is stored as list of nodes and signals. Some of the operations can be composed of the multiple nodes, typically the memory operations where the pointer is firstly modified to point the desired place in memory and then this pointer is used to store or load value.

The nodes and signals structure contains additional information, such as assigned instruction, functional unit or registers which will be needed later for algorithm mapping on the processor resources. At this point, only instructions can be assigned to the node according to its operation.

5.3.1 Node sorting

When the algorithm is parsed, the relations between nodes can be found. It is realized by pairing the input and output signals of nodes. This creates the possible execution order of the nodes. Multiple nodes can have assigned the same execution level independently on the architecture. This parameter is only informative to the next steps to ensure correct functionality.

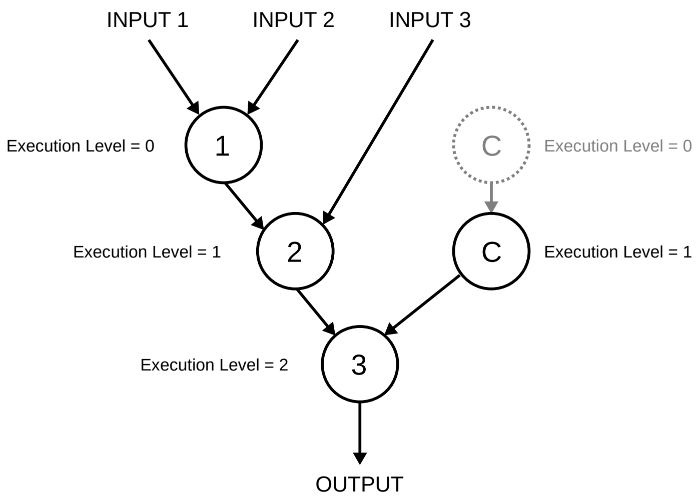


Figure 5.3: Determining execution level

Figure 5.3 shows how the execution level is determined. Nodes which process input signals have the execution level equal to zero (node 1). This means that they can be executed immediately after launch. If node processes at least one signal which is result of another node, its execution level will be higher than the highest value of the nodes that create its input signals (nodes 2, 3). Constants (node C) have execution level equal to zero at the beginning of this process to ease assignment on the other nodes. After all nodes have its level assigned, the constants are moved right before the all nodes which use its value.

When the execution levels are determined, the list with nodes can be sorted according this parameter. At this point, the algorithm can be mapped to the functional units, but the result will be highly depending on the algorithm definition in the input file. For this reason, additional parameters are added for possible increase of performance.

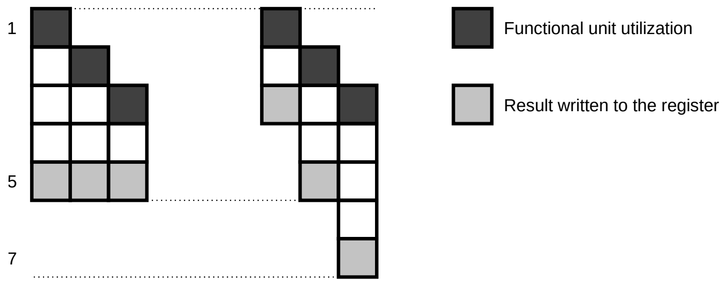


Figure 5.4: Instruction execution order based on CPU cycles

The first parameter is the number of instruction cycles. Figure 5.4 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 side is the worst case, when the instructions are executed in the reverse order. The execution of all instruction takes 7 CPU cycles instead of 5.

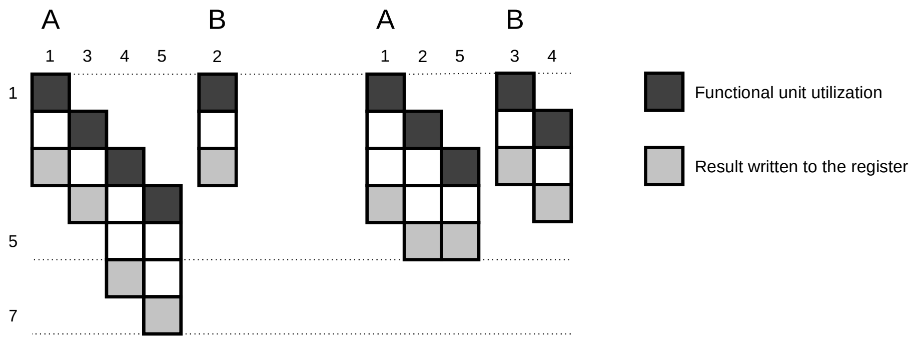


Figure 5.5: Instruction execution order based on number of supported functional units

The second parameter is the number of supported functional units where the instruction can be executed. Figure 5.5 shows the situation on two functional units A and B and five instructions. The sorter instruction (3 CPU cycles) can be executed on both functional units. The longer instruction (4 CPU cycles) can be executed only on functional unit A. The number on top indicates the order of instruction mapping. The case on the left side is the worst case, when the short instructions are allocated first and after that allocation continues with longer instructions. The result is that the functional unit is executing only instruction and the rest is executed on the functional unit A. The execution of all instructions takes 7 CPU cycles. The situation on right side is ideal, because the longer instructions were allocated first, so they are not blocked by the shorter instructions. The execution now takes 5 CPU cycles.

5.3.2 Functional unit allocation

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

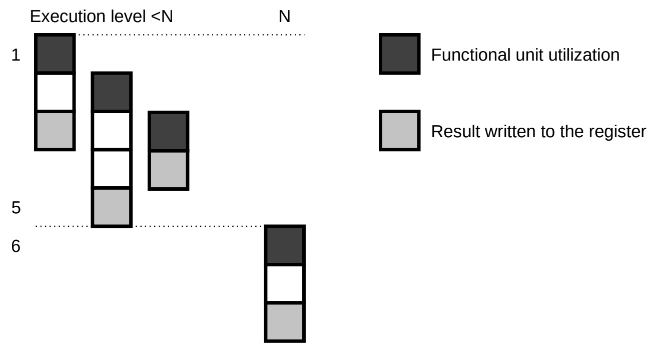


Figure 5.6: Determining first possible CPU cycle for execution

Figure 5.6 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.

The instruction mapping into the functional unit is similar to the first-fit method in 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 also set to take the priority on the functional unit examination.

Allocation without priority is actually equivalent of the first-fit method. The tool starts examining the functional units in the usage map from the first possible instruction cycle, which can be used to execute selected instruction. 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.

When the instructions are mapped without any functional unit priority, the result will be dependent on the functional unit order in the architecture definition. The first simple method prefers the functional units that supports the least number of instructions present in the algorithm, 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.

The next 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 a 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.

5.3.3 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.

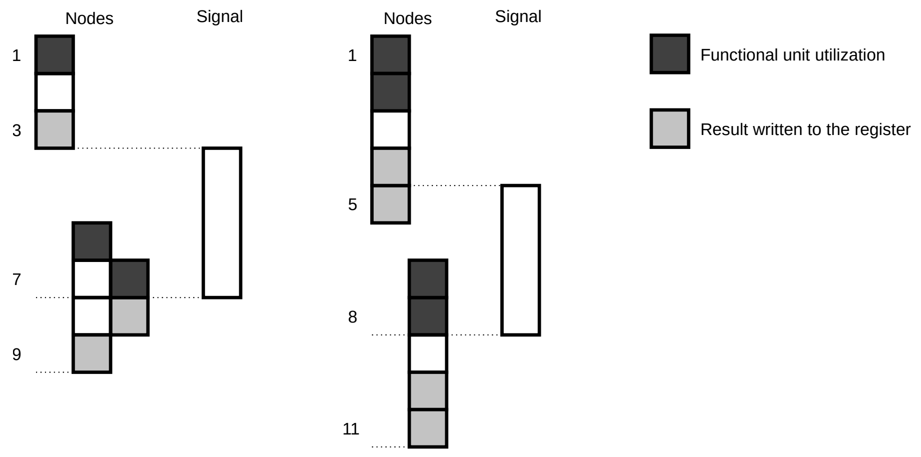


Figure 5.7: Determining signal lifetime

Figure 5.7 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 such as overview of usage maps as well.

6 Experimental results

The goal of the proposed mapping technique is to use a potential of VLIW architectures, which is to process data on multiple functional units in parallel operations. Because the target architecture uses pipelining, not only the number of used functional units will be evaluated, but also the usage ratio. For evaluation, several algorithms were implemented with the aim to the possibility of the parallel execution, especially FFT and matrix multiplication.

6.1 Basic behavior of algorithm mapping

First tests were performed on algorithm versions without memory access. The input values are stored in registers and results are stored back to registers as well. The available results are explained on illustrative 4-point FFT radix-2 with time decimated complex input. The algorithm has 8 (4 real and 4 complex) input and 8 output signals. The operations are only additions and subtractions. The simplification is achieved by twiddle factor

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

are only additions and subtractions. The simplification is achieved by twiddle factor

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

The algorithm description without signal definitions is shown in Figure 6.1. 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 [50] [51] (see Figure 6.2). The rectangle symbols represent input, output and internal signals and the ovals represent all mathematical operations.

```
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 6.1: Source code of the 4-point FFT (without signal definition)

The whole algorithm description is shown in Figure 6.1. 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 [50] [51] (see Figure 6.2). The rectangle symbols represent input, output and internal signals and the ovals represent all mathematical operations.

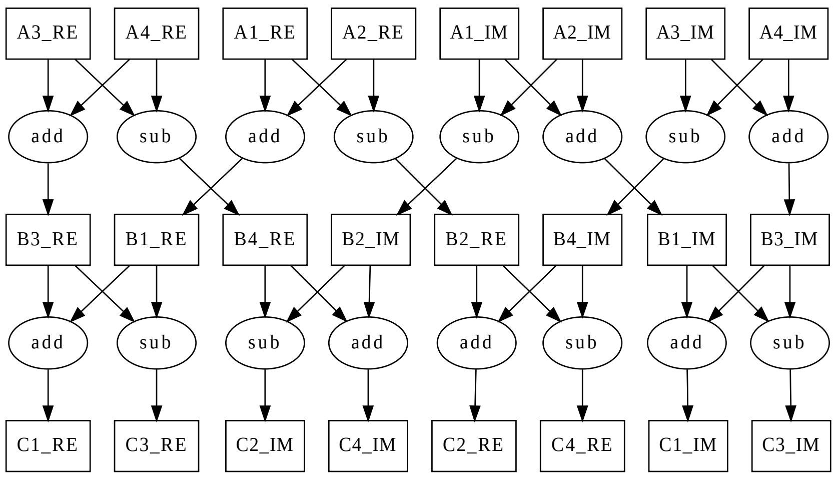


Figure 6.2: Graphical representation of the 4-point FFT

The final code generated for 32-bit fixed-point number representation is shown in Figure 6.3 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 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.

```
    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 6.3: Generated source code for the 4-point FFT with fixed-point representation

Execution time of the code compiled for 32-bit integer values is 7 instruction cycles. For comparison, processing of the single precision floating-point data takes 12 instruction cycles.

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 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. The 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

6.1.1 Values stored in memory

The following case counts with the input values stored in the data 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.

The mathematical structure of the algorithm is the same as in the previous case. The difference is in the input/output signal definitions. Instead of 8 input and 8 output signals with values, the algorithm has 2 input pointers, one for access to values from memory and one for storing the results.

Figure 6.4 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, the final implementation contains more signals and operations. It is given by the multioperation nodes, which are creating another signals and operations for achieving desired result. These nodes are memory loading and storing which in first step creates the constant with the offset, then modifies the pointer and finally loads or stores the value. Table 6.2 shows the performance of the implemented algorithms. The resource utilization is obviously smaller than in previous case. The code which performs the algorithm is usually about 1/3 of the total execution time. The rest is the code for memory operations. The loading and storing can be performed only on .D unit.

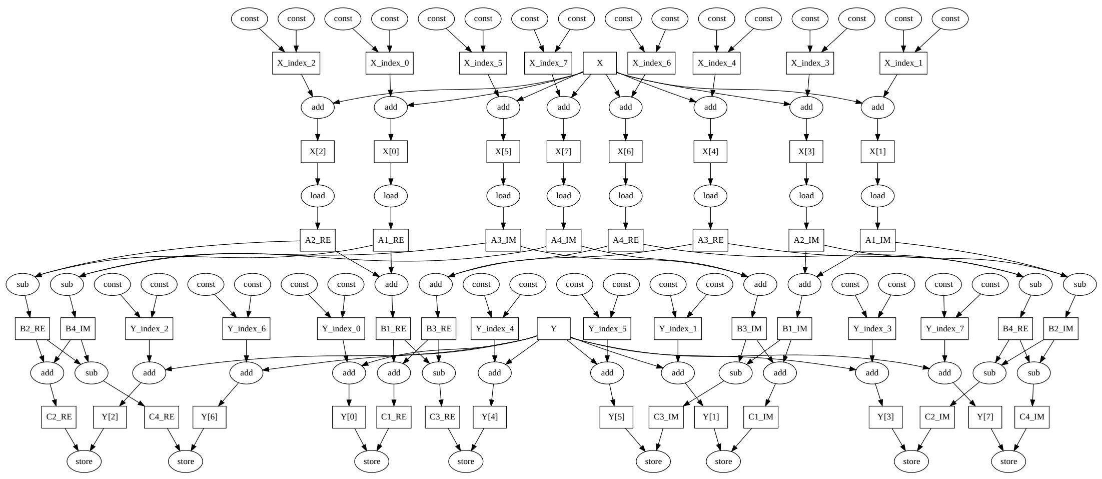


Figure 6.4: Graphical representation of the 4-point FFT with memory operations

Figure 6.5: Functional unit usage in FFT4 (32-bit integer, data in memo

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

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. The first part of 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 these 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 times of generated codes were the same.

There are two types of algorithms with none, or in significant improvements. 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.

On the other side, the algorithms with the highest improvement contain 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 the 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 are showed in the second part of Table IV. The performance is compared with the case when the architectures functional units were defined in the worst-case order for each examined algorithm. Now it can be seen that the difference of the execution time can be up to 37 %.

Table 6.3: Priority mapping improvements (data in memory)

Algorithm	**Data type**	**Node priority improvement [%]**		**Functional unit priority improvement [%]**	
		Units priority	**Cycles priority**	**Global priority**	**Dynamic priority**
Mat. mpy 2x2	Int32	0.00	0.00	21.43	28.57
	Float	0.00	0.00	23.81	23.81
	Double	-3.85	-3.85	33.33	33.33
Mat. mpy 3x3	Int32	-3.13	-3.13	32.63	37.89
	Float	0.00	0.00	33.68	33.68
FFT4R	Int32	6.06	9.09	25.00	25.00
	Float	11.76	11.76	22.73	22.73
	Double	3.28	3.28	27.38	27.38
FFT4C	Int32	-2.56	-2.56	29.09	29.09
	Float	4.88	4.88	24.07	24.07
	Double	4.11	4.11	28.43	28.43
FFT8R	Int32	6.35	7.94	26.74	26.74
	Float	12.12	12.12	26.67	26.67
FFT8C	Int32	2.56	2.56	29.09	29.09
	Float	9.41	9.41	23.42	23.42

Table 6.3 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 6.4 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 6.4: 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.

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. 1966. Proceedings of the IEEE. ISSN: 0018-9219.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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>.

[21] 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>.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] 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.

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

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

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

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

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

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.

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.

Mego, Roman. Parallelism in digital signal processing: short version of 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.