A Tool for VLIW Processors Code Optimizing

Roman Mego, *Member, IEEE*
Department of Radio Electronics
Brno University of Technology
Brno, Czech Republic

Tomas Fryza, *Member, IEEE*
Department of Radio Electronics
Brno University of Technology
Brno, Czech Republic

***Abstract*—The paper demonstrates the behavior of low- and high-level programming languages on the multicore digital signal processors based on Very Long Instruction Word architecture. The aim of the paper is to present a tool that can be used to implement any digital signal processing algorithm on such processors with efficiency of the low-level languages, but with the advantages of the high-level programming languages. Preliminary result is the software that uses a signal-flow graph approach to describe an algorithm, generates low-level assembly code and provides (graphical) information about the algorithm.**

I. Introduction

The performance of an application is not only dependent on device's hardware, but also on the software. It is really important part of the application, because well optimized code could make better performance on the low-cost hardware than bad written code running on the high-priced devices.

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 any 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 optimizing of the critical parts. 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.

The high-level programming 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. 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, they are not able to handle some special cases, such as: inability to express special DSP operations (addition, subtraction and multiplication with saturation), inability to express vector operations, inability to mark independent parts of program which can be run in parallel due to sequential character of notation, or 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 [1]–[3] or by the third party [4], compiler extensions, such OpenMP for program execution on shared memory system or MPI for distributed memory system or with special programming languages like CUDA for general purpose processing on GPU. There are also some projects such as [5] that are able to handle the instruction level parallelism more effective.

Standard optimizing methods are set of analyze and transform operations performed on source code to run it faster or consume less hardware resources. These operations find and replace parts of code with more efficient alternatives. The compilers use two main techniques to determine the code parts to optimize: control flow analysis and data flow analysis [6].

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 and can be found in [6] and [7].

II. Signal-flow Graph Tool for Code Optimizing

The software plays the key role in the whole signal processing systems based on DSPs (Digital Signal Processors). For evaluating methods of digital signal processing algorithms, multicore VLIW (Very Long Instruction Word) processor is commonly chosen, because it allows to implement parallelism by the high-level language through the operating system support and the instruction-level parallelism by the hand optimized code. The TMS320C6678 [8] from Texas Instruments fits perfectly, because it is the multicore VLIW based DSP controlled by 1.25 GHz. This allows to demonstrate the instruction-level parallelism and threading the level parallelism as well.

The main problem of the standard compilers for the VLIW architectures (or similar) is that the sequential notation of the program makes difficult to find the possible parallel operations. The tool presented in this section uses the idea of the signal-flow graph, which could be also found in the hardware description languages (HDL). It means that the source code does not represent the execution sequence of the algorithm, but only the relations between signals.

The tool is divided into two basic parts. First part is designed for the architecture definition from the user interface. The second part is used to generate the final code in low-level assembly from the algorithm description and the architecture definition.

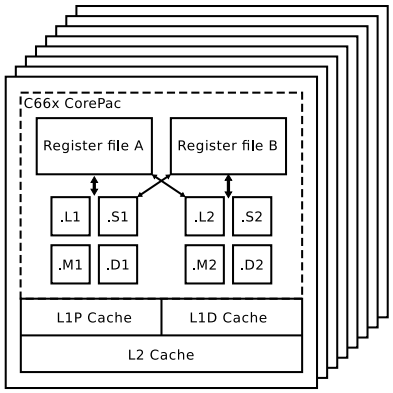

Fig. 1. Structure of the TMS320C6678 digital signal processor core(s)

The architecture is defined in text file in the JavaScript Object Notation (JSON) format, which is an alternative to Extensible Markup Language (XML). The 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 files are hand-edited.

The modern DSP cores have quite large amount of resources for parallel operations, but there are also limitations. The first is that the common functional units are not equal. They are not capable to execute the same instructions. For example, functional units of TMS320C6678 DSP (see Figure 1) are marked .L1, .L2, .S1, .S2, .D1, .D2 and .M1, .M2. The .D units are primary used for loading and storing data from/to the memory. The .L and .S units are designed for the general arithmetic, logic and branch operations. The last, .M units are able to perform multiply operations with fixed-point and 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 previously mentioned hardware resources into two identical data paths. These 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 core is capable to transfer a 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 cores.

A. Algorithm Description

The algorithm notation uses the signal-flow graph description based on HDL and the tool's syntax uses two base elements: signals and nodes. The signals are equivalent for variables, but there is a limitation for their use. In classic sequential programming languages, like C, the variable can change its value during runtime many times. Here, the signal value can be assigned only once. There are three types of signals: input signals which are allocated at the beginning of runtime, output signal must be valid until the end of execution, and temporary signals could be created and expired when required. Each signal is represented by its name, data type and role in the algorithm. Example of the definition format is shown on Figure 2.

TABLE I. SIGNAL DEFINITION ROLES

Signal role	Description
INPUT	The signal carries the input data. It it 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 II. 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)

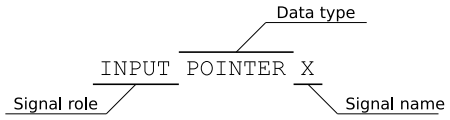

Fig. 2. Signal definition format

The order of 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 a pointer to input data. The supported values are listed in Table I and II.

The other elements in the tool's 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. The mapping process leads to the semi-ideal low-level assembly code of the algorithm for target architecture.

The nodes are practically every operation with the signal, but the tool recognizes five types of operations, which have different definition format. These operations are: arithmetic operation, function, constant assignment, signal assignment, and memory operation. The format is shown on Figure 3.

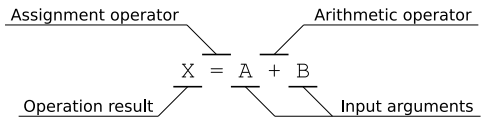

Fig. 3. Operation format

An arithmetic format contains basic mathematical operations with two input arguments, which produce one result. The definition is similar to other programming languages, but there can be one operation per line only. The algorithm description is not limited to the standard arithmetic operations. If the operation is different from the standard arithmetic operation, it is written in the function format like in C language. The

difference is that the syntax is extended by the multiple function outputs (Figure 4).

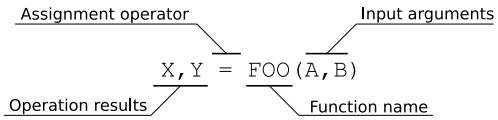

Fig. 4. Function format

Some operations defined in the algorithm can be divided into the multiple nodes. These operations are typically memory reading/writing and constant loading. The constant loading is only divided into the loading of the upper and lower part of the register, or loading more registers in case of wider data type. The memory operations are working with the pointers. Typically, an algorithm wants to load or store array element, which is defined by the arrays pointer and the element index. To get the value from memory, the tool needs to get the element address and after that it can load the value. This process consists at least from three nodes which are shown in Figure 5.

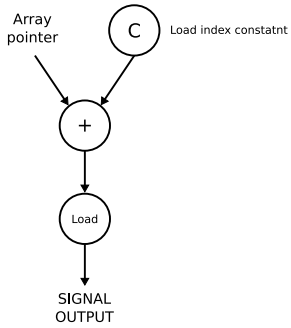

Fig. 5. Loading process based on updated data pointer

The mapping process it self is based on the first-fit method. For that reason, the processing order of the nodes must be considered before the functional unit allocating. 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, and the execution level of the node must be higher than the highest execution level of the nodes which creates its input signals.

The additional parameters for choosing the first node to fit are the number of cycles needed to execute the instruction and number of functional units that are able to execute the instruction. By these parameters it is achieved the higher level of parallelism, because the operations that can be executed only on single functional unit are not blocked by the operations which can be executed on another units. Also when the execution time of the instruction is considered on processor which supports pipelining, the execution time it self can be reduced (Figure 6).

III. Experimental Results

A core of the processor based on VLIW architecture contains multiple functional units with ability to execute multiple instructions at once. Software for VLIW is made of instruction packets, which are created statically during the software

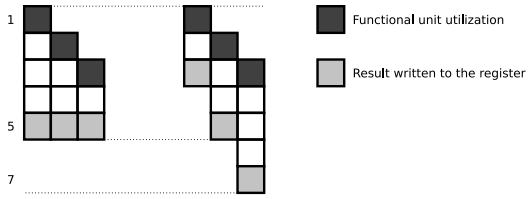

Fig. 6. Reduction of execution time by instruction reordering

compilation. Thanks to this, the hardware of VLIW core structure can be simplified. This makes space for the additional functional units, its functionality, or the increase of CPU 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.

The TMS320C6678 is a multicore fixed/floating point digital signal processor and it is containing of eight C66x DSP cores [9]. 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 combine more fixed point product and sum operations in a CPU cycle (SIMD) which 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 [8].

The proposed mapping tool was verified on several simple algorithms with the aim of determining the efficiency of the processors functional units and general purpose registers. Three basic algorithms were chosen for their high potential of parallelization and for their indispensability in signal processing and communication domain: FFT, matrix multiplication, and DCT.

Two basic concepts were analyzed: a situation when the input values are already prepared in the general purpose registers and the results of the algorithm are stored in registers to pass them away from the function as well. The second concept counts with the input values stored in the memory. This means that the input of the algorithms is only a 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.

In the proposed tool, the algorithm structure can be visualized through the generated DOT file. The rectangle symbols represent input, output, and internal signals and the ovals represent all mathematical operations. Generated visualization of 4-point DCT can be seen in Figure 7.

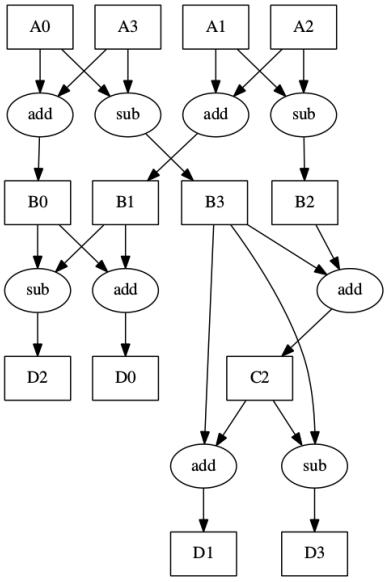

Fig. 7. Graphical representation of the 4-point DCT

TABLE III. AVERAGE HARDWARE RESOURCES USAGE ON REGISTER-BASED ALGORITHMS

Algorithm	Data type	Cycles	Usage of units [%]	Usage of regs [%]
Mat. mpy 2×2	Int32	13	25.00	18.27
	Float	15	21.43	15.83
Mat. mpy 3×3	Int32	32	36.29	50.98
	Float	36	32.14	46.88
FFT4 complex	Int32	8	57.14	25.39
	Float	13	33.33	18.27
FFT8 complex	Int32	20	73.68	53.75
	Float	30	48.28	43.85

The tool maps the algorithm only into data path A, but an identical function with different data could be executed in data path B concurrently.

Another useful output of the tool is a generation of usage maps of the processor resources. The rows of the maps represent the instruction cycle from the beginning of the execution and cols represent the hardware resources. The map for the functional units contains instructions, which are executed. The register map contains the signals assigned to the registers and helps developers to work with the register file.

The summary of experimental results based on both register and memory approaches are shown in Table III and Table IV, respectively. These register-based approach assumes that the input values are already stored in the general purpose DSP registers, so the memory operations are not included. On the other hand, in memory-based approach, the algorithm's input contains only pointer to the data source and destination arrays, so the memory operations are included and generated by the tool.

Here, the 32-bit fixed-point and 32-bit floating-point representations were used. For simple matrix multiplications and FFT transforms, numbers of CPU cycles needed to accomplish the algorithm execution are indicated. The average usage of functional units and general purpose registers are shown as well.

TABLE IV. AVERAGE HARDWARE RESOURCES USAGE ON MEMORY-BASED ALGORITHMS

Algorithm	Data type	Cycles	Usage of units [%]	Usage of regs [%]
Mat. mpy 2×2	Int32	32	48.39	20.21
	Float	31	50.00	19.25
Mat. mpy 3×3	Int32	65	59.77	42.79
	Float	62	62.70	37.35
FFT4 complex	Int32	39	52.63	27.24
	Float	38	54.05	23.85
FFT8 complex	Int32	75	62.16	51.88
	Float	77	60.53	48.58

According to the results, the performance or effectiveness of the specific algorithms could be considered. The higher value of functional unit usage, the lower number of CPU cycles the units are idle. The implementation of 8-point FFT in fixed-point representation is more effective then the float-point representation.

IV. CONCLUSION

The paper presents a tool to describe an algorithm from digital signal processing domain. The tool is able to find out any dependencies in the algorithm description and to generate a low-level code for the target VLIW processor. The average usage of main hardware resources, such as functional units and core registers is provided by the tool as well. In the text, the basic tool functionality was outlined with help of FFT, matrix multiplications, and DCT algorithms. In the future, more results from different DSP algorithms will be provided.

ACKNOWLEDGMENT

Research described in this paper was financed by the Czech Ministry of Education within the frame of the National Sustainability Program under grant LO1401. For research, infrastructure of the SIX Center was used.

REFERENCES

[1] Texas Instruments Inc.. TMS320C67x DSP library programmer's reference guide. [Online]. 2010. http://www.ti.com.cn/cn/lit/ug/spru657c/spru657c.pdf.

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

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

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

[5] 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.

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

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

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

[9] Texas Instruments Inc. TMS320C66x CorePac user guide. [Online]. 2013. http://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf.