Journal of ELECTRICAL ENGINEERING, VOL 73(2022), NO6, 387–395

sciendo

PAPERS

Instruction mapping techniques for processors with very long instruction word architectures

Roman Mego[1], **Tomas Fryza**[2]

This paper presents an instruction mapping technique for generating a low-level assembly code for digital signal processing algorithms. This technique helps developers to implement retargetable kernel functions with the performance benefits of the low-level assembly languages. The approach is aimed at exceptionally long instruction word (VLIW) architectures, which benefits the most from the proposed method. Mapped algorithms are described by the signal-flow graphs, which are used to find possible parallel operations. The algorithm is converted into low-level code and mapped to the target architecture. This process also introduces the optimization of instruction mapping priority, which leads to the more effective code. The technique was verified on selected kernels, compared to the common programming methods, and proved that it is suitable for VLIW architectures and for portability to other systems.

K e y w o r d s: digital signal processors, parallel architectures, low-level code, instruction mapping, signal-flow graph

1 Introduction

The computing power of processors has been steadily increasing since its invention. This goal was achieved by higher operating frequencies [1] or by the addition of more features, such as a special instruction set for signal processing [2]. This instruction set may include multiply and accumulate operations, fused multiply-add, vector operations or saturated arithmetic [3], [4]. The extra instructions are realized with higher transistor density, which means higher power consumption despite the fact, that each transistor has lower operating voltage to achieve fewer thermal losses. It could result in overheating problems, especially with higher operating frequencies and smaller dimensions, which should be considered during system design [5].

Processor development has taken a different direction in the last two decades. Higher computing performance has been achieved by using parallel data processing [6]. It includes instruction sets for vector processing, multiple functional units in one processor core, or multiple cores or processors in one system.

The software point of view is also important for final performance. The time-critical parts of programs are usually written in low-level assembly language to achieve optimization in terms of code size and execution speed. On the other hand, it is not easy to write a complex software and low-level code is usually fixed only on specific architecture. These problems can be solved by using high-level languages, such as ANSI/ISO C, C++, and others. Modern compilers can highly optimize final code. This feature works perfectly on scalar processors but on certain architectures containing instruction parallelism, such as VLIW (very long instruction word) [7], the standard optimization is still not effective.

In [8], Rajagopalan *et al* extended the VLIW compilation environment to develop the retargetable optimizations for DSPs (digital signal processors). There are other tools and frameworks, but they are intended for generating code on different platforms and accelerators, such as FPGAs, GPUs, CPUs, or heterogeneous systems.

In [9], an approach for automatic generation of architecture-level models for processors and accelerators from their RTL (register transfer level) designs was proposed. Fang *et al* [10], introduced an automatic hardware design tool and generator of RTL codes in Verilog HDL, which can be implemented to FPGA devices. Xing *et al* in [11], presented a formal model of the GPU using instruction-level abstraction techniques. The model enables the verification of multithreaded programs. Steuwer *et al* in [12], introduced a new internal representation for OpenCL, which encodes constructs as functional patterns. In [13], authors proposed a new thread low-level architecture independent mapping scheme for GPGPU (general-purpose graphics processing unit) programming. A set of lower-level abstractions of large-scale heterogeneous systems, including multidimensional arrays and SIMD (single instruction multiple data) vectorization was described in [14]. A template-based optimization framework to automatically generate fully optimized assembly code for several dense linear algebra kernels on varying multi-core CPUs was presented by Zhang *et al* in [15].

This paper describes a tool suitable for generating a low-level assembly code for signal processing kernels on VLIW architectures and includes memory access and instruction mapping optimization techniques.

[1] Member, IEEE, Department of Radio electronics, Brno University of Technology, Brno, 61600 Czechia, mego@vut.cz, [2] Senior Member, IEEE, Department of Radio electronics, Brno University of Technology, Brno, 61600 Czechia

https://doi.org/10.2478/jee-2022-0053, Print (till 2015) ISSN 1335-3632, On-line ISSN 1339-309X

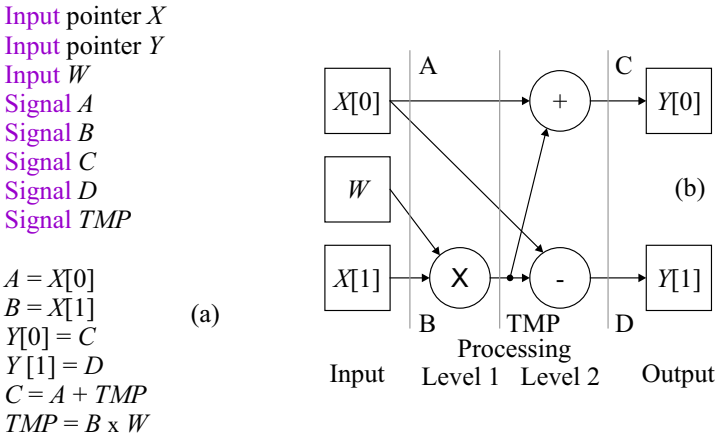

Fig. 1. Example of algorithm description:(a) – syntax,(b) – signal-flow diagram

We introduce the current programming and optimization methods and identified the main problems in creating the efficient codes. The main idea of the proposed method for instruction mapping on DSPs with instruction parallelism is described together with a tool for creating optimized low-level code for digital signal processing functions and illustrates the new optimization processes to achieve the better performance of the generated code. Experimental results which include output efficiency on selected digital signal processing algorithms are presented.

2 Related works

The programming languages as ANSI C and C++ are commonly used in software development for embedded devices. Compilers for these languages generate an efficient code on scalar systems for control application but they achieve lower results in some exceptional cases. The following examples refer to the standard C/C++ expressions:

- inability to express special operations with saturation;
- inability to express vector operations;
- inability to mark the independent part of programs which can be run in parallel.

Some of them were removed using the optimized DSP and math libraries provided by processor manufacturers, such as [16] or [17]. These libraries usually consist of preprocessor macros for using special instructions or whole functions, which are written in low-level assembly. Using this approach leads to highly optimized, but non-portable code.

Another solution of writing parallel data processing is to use compiler extensions, such as OpenMP. This approach is aimed on multi-core systems so it cannot be used on to define instruction-level parallelism.

The code optimizations are set of analyze and transform operations which find and replace parts of code with more efficient alternatives. The compilers use two main techniques to determine the code parts to optimize [18-20]:

- control flow analysis;
- data flow analysis.

Control flow analysis is based on the examination of control statements which can branch the program. In this case, the optimizations are applied on the possible paths of program execution. Data flow approach analyzes the usage of data in program and can be used for reducing number of variables, optimize loading of constants and data transfers.

These methods separate source code into basic blocks and execute operations to increase performance, such as dead and redundant code elimination, loop unrolling, constant folding, copy propagation transformation, and others [18]-[20].

3 Signal-flow graph approach

The standard optimization methods used by compilers are aimed to reduce unnecessary code, but they are designed more likely for the control applications. The goal of proposed approach is to eliminate causes of the performance degradation. The main points are:

- reduce memory access and increase usage of the general- purpose registers for storing temporal data;
- reduce data transfer between internal data paths;
- increase instruction parallelism by breaking the rules of the execution order given by notation of the source code. With signal-flow graphs, an algorithm is described only with the relations between signal values and without the processing order. The relations are like the data-flow analysis in commonly used compilers but for different purpose. Normally, it is used to identify identical or similar parts of the code.

The proposed tool uses this description for finding independent parts of data processing, which lead to identification of possible parallel execution order.

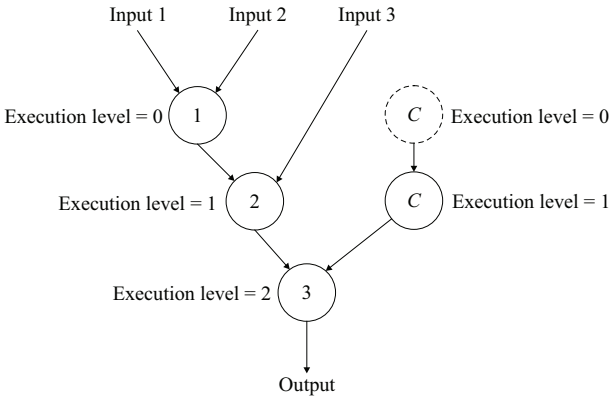

Fig. 2. Determining of execution level

The signal-flow graph-based approach is like the HDL (hardware description language). The algorithm description contains two basic elements, signals and nodes. The signal is equivalent to a variable in C language but in the tool's description it can be assigned only once.

For illustration, Fig. 1(a) shows an example of the algorithm description code and Fig. 1(b) its graphical representation. The code here has three signals X, Y and W which can be compared to the input arguments of the function in C language. Signal X is the pointer to array with input values, signal Y is the pointer to output array and W is the other input parameter. Internal signals A, B, C, D and TMP are used for temporal results. Note that, the algorithm here consists of two processing levels and operations are not needed to be strictly written in the processing order. The more detailed description of algorithm syntax was published in [21].

4 Instruction mapping technique

The goal of the mapping process is to assign operations from the algorithm description to the target processor hardware resources. The parsed algorithm is stored as the list of nodes and signals. Some of the operations can be composed of the multiple nodes, typically the memory operations via pointers. The nodes and signal structure contains additional information, such as assigned instructions, functional units or registers.

At this point, only instructions can be assigned to the node according to its operation.

4.1 Node sorting

When an algorithm is parsed, the relations between nodes is found and the possible execution order is created. Figure 2 shows how the execution level is determined. Nodes which process input signals have the execution level zero (node 1) and they can be executed immediately. If a node processes at least one signal which is result of another node, then its execution level is higher (nodes 2, 3). Let constants (node C) have execution level zero to ease assignment on the other nodes. When all nodes have assigned its level, the constants are moved right before the nodes which use its value; here the node C was moved to the execution level 1. At this point, the algorithm can be mapped to the functional units, but the result will be highly depending on the algorithm definition. For this reason, additional parameters are added for performance increase.

The first parameter is the number of instruction cycles. Figure 3 shows two examples with three pipelined independent instructions i_1, i_2, i_3 with different duration but executed by the same functional unit A. The green squares represent moments when the instruction execution starts and blue ones when results are written, and instruction ends. The right-hand case has the ideal order, when the first executed instruction takes five CPU cycles, and the last instruction takes three cycles before the result is written to the destination register(s). The case on the left-hand side is the worst case when the instructions are executed in the reverse order. The execution of all instructions here takes seven CPU cycles instead of five.

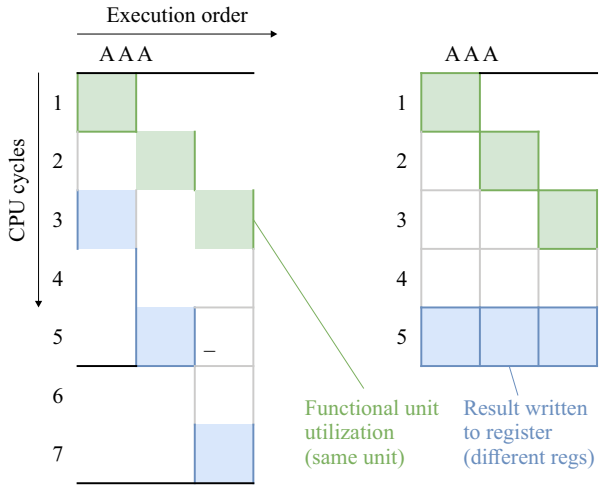

Fig. 3. Instruction execution order based on CPU cycles. Left: worst order, right: ideal orderDetermining of execution level

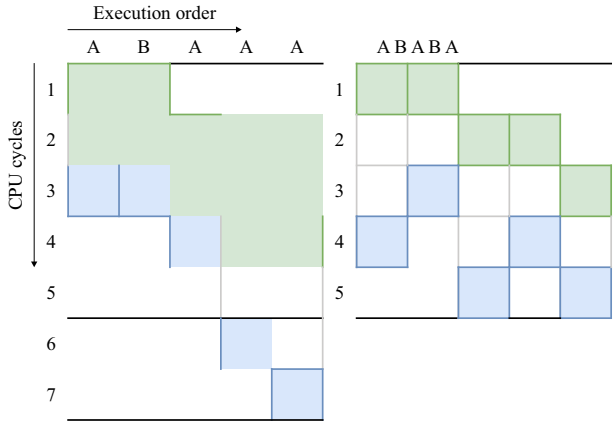

Fig. 4. Instruction execution order based on number of supported functional units: (left) – worst order, (right) – ideal order

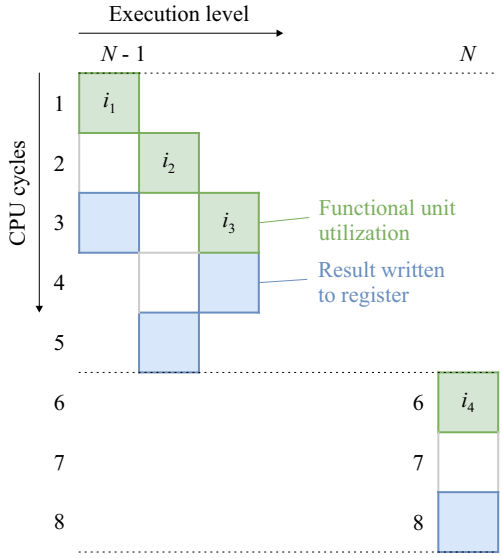

Fig. 5. Determining first possible moment for execution of i_4

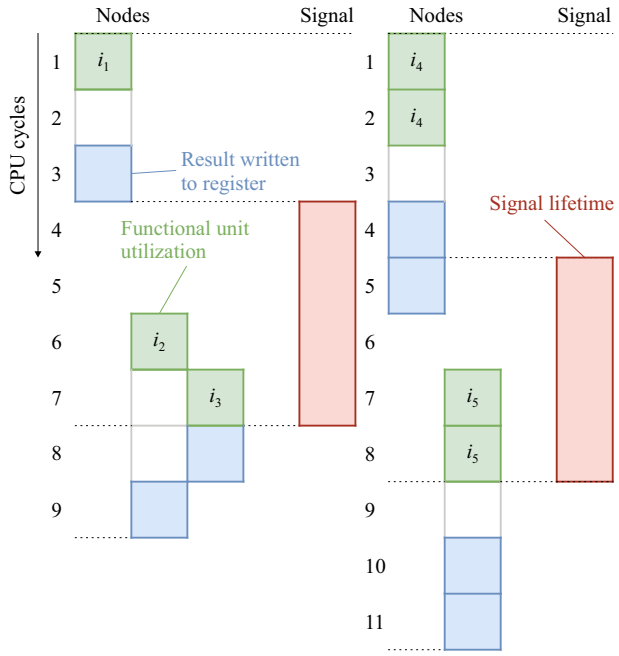

Fig. 6. Determining signal lifetime: (left) – one CPU cycle for reading/writing operations, (right) – two CPU cycles

The second parameter is the number of supported functional units by which an instruction can be executed. Figure 4 shows the situation with two functional units A and B and five independent instructions i_1 to i_5 with different duration of three or four CPU cycles. Let the shorter instructions i_1, i_2, i_3 (three CPU cycles) be executed on both functional units. The longer instructions i_4, i_5 (four CPU cycles) can be executed only on functional unit A. The case on the left-hand side is the worst case, when short instructions are allocated first and longer instructions are allocated later. The result is that the functional unit B executes only one instruction i_2 and the rest is executed by functional unit A. It takes seven CPU cycles to finish execution of all five instructions. The situation on the right-hand side is ideal because the longer instructions were allocated first, so they are not blocked by the shorter instructions. The total execution now takes five CPU cycles.

4.2 Functional unit allocation

Each node needs to have a minimal start cycle when the instruction can be allocated. This cycle is determined only if all instructions from the previous execution level(s) are completely mapped. Exceptions are nodes with execution level zero which can be executed immediately at the beginning.

Let Fig. 5 show the four instructions and let i_4 on execution level N depend on results from i_1, i_2, and i_3 on lower execution level $N-1$. Since the last result of these instructions is written on 5-th CPU cycle, the examined instruction i_4 could be executed in the 6-th CPU cycle at the earliest.

The instruction mapping into a functional unit is based on the first-fit method when instruction is mapped into the first suitable position. Unlike the memory management, this allocation process considers two dimensions: functional unit and time. The priority on functional unit examination can be applied as well.

A functional unit that can execute the selected instruction is searched for from the first possible cycle. When an unused functional unit is found, the node is placed into the map. If there is not any available functional unit, the instruction cycle is incremented, and the process is repeated. When the functional unit priority is not applied, the mapping depends on the order of functional unit in architecture definition.

The first allocation method prefers the functional units that supports the least number of instructions presented 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.

In the second allocation method, the functional unit examination is dynamic according to the instructions in the remaining unallocated nodes. In each node allocation step, it finds several upcoming nodes which can be executed on each functional unit. The highest priority has the functional unit with the smaller number.

4.3 Signal allocation

Signals are allocated to general purpose registers after all nodes are mapped. The reason is relationship between the node's execution time and the signal's lifetime. Here, the lifetime of a signal means time when register hold a value from given node and cannot be rewritten. Because the number of registers is limited, the signals are not allocated during the whole algorithm process, but only for the necessary time. The lifetime of the signal starts by writing a value and it ends with the last reading. Only

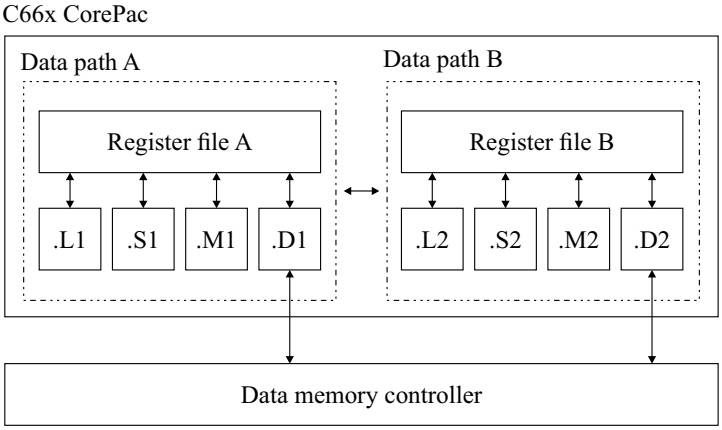

Fig. 7. Simplified structure of TMS320C6678 DSP core

the input signals are allocated from the first instruction cycle, and the output signals keeps their values until the end of the algorithm.

Figure 6 shows examples of the signal lifetime used by two nodes. On the left-hand side, the signal lifetime starts one CPU cycle after the writing of i_1 value and it ends after the last instruction reading of the second target node, *ie* by instruction i_3.

The second case on the right-hand side shows the situation when instructions need more than one CPU cycle for reading and writing operations, typically when double values are processed. The lifetime ends after the read (instruction i_5), like in the previous case but it starts after the first CPU cycle of the write operation (instruction i_4).

When all signals have its lifetime, they are allocated to the registers. The 2D map of the register usage in time is created, and the registers are placed into this map using the first-fit method. After this procedure, the final low-level assembly code is generated according to the target device description.

5 Experimental results

The potential of the proposed mapping technique was verified by VLIW-architecture processors, which process data on multiple functional units in parallel. The used hardware resources model was based on TMS320C6678 device. It is a multicore DSP by TI, with fixed and floating-point arithmetic support. The DSP integrates eight C66x cores, each with two identical data paths A and B. Each data path contains four functional units and thirty-two 32-bit general purpose registers, see Fig. 7, [22]. The functional units .L, .S, .M and .D are designed for a different purpose and support different instructions. The .L and .S functional units are designed for general mathematical operations, the .M units have multiplication capabilities and .D are primarily used for memory access.

5.1 Basic behavior

The functional verification was first performed without memory access when input values and results are stored in register files. For illustration, the achievable results are presented by comprehensible 4-point FFT radix-2 with time decimated complex input [23]. Thanks to the optimizations from [24], the 4-point version contains only addition and subtraction operations. The part of algorithm description without signal definitions is shown in Listing 1 and for better understanding, it can be visualized by the generated DOT file [25] (see Fig. 8), where rectangle symbols represent input, output and internal signals and oval symbols represent all mathematical operations.

Listing 1: Part of 4-point FFT algorithm description

```
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
```

The final code generated for 32-bit fixed-point representation is shown in Listing 2 with the appropriate comments. The || signs represent the parallel executions of instructions.

Listing 2: Part of 4-point FFT generated code with complex values and fixed-point representation

```
||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
```

Table 1. Average hardware resources usage on selected algorithms without and with memory access

Algorithm	Data type	Without memory access					With memory access				
		CPU cycl.	Used (%)		Total (%)		CPU cycl.	Used (%)		Total (%)	
			FUs[a]	Regs[b]	FUs	Regs		FUs	Regs	FUs	Regs
Mpy 2×2	int32	13	46.2	73.1	23.1	18.3	33	45.5	52.3	45.5	19.6
	float	15	40	63.3	20	15.8	32	46.9	54.3	46.9	18.7
	double	15	40	63.3	20	31.7	52	51.9	44.5	51.9	30.6
Mpy 3×3	int32	32	70.3	81.6	35.2	50.9	64	59.8	50.5	59.8	41
	float	36	41.7	71.4	32.3	46.9	63	60.7	46.4	60.7	34.8
FFT4R	int32	5	80	76	60	23.8	33	60.6	51.5	45.5	17.7
	float	9	66.7	73.6	33.3	18.4	34	58.8	46.3	44.1	17.4
	double	10	80	78.6	40	39.4	61	61.2	48.5	45.9	30.3
FFT4C	int32	7	76.2	66.3	57.1	29	39	68.4	28.9	51.3	19.9
	float	12	66.7	79.2	33.3	19.8	41	65	59.3	48.8	18.5
	double	12	66.7	79.2	33.3	39.6	73	65.8	50.3	49.3	31.4
FFT8R	int32	13	73.1	74.2	73.1	46.4	63	53.2	54.1	35.2	32.1
	float	22	57.6	74.4	43.2	37.2	66	50.8	56.9	50.8	30.2
FFT8C	int32	20	70	86	70	53.8	78	58.9	53.9	58.9	37.1
	float	30	62.2	82.6	46.7	43.9	85	54.1	53	54.1	34.8
BQ1	int32	10	30	48.3	15	9.1	24	36.1	65	27.1	10.2
	float	16	18.8	38.5	9.4	7.2	30	21.7	56	21.7	8.8
	double	16	18.8	38.5	9.4	14.5	40	28.8	46.3	28.8	17.3
BQ2	int32	16	25	48.3	18.8	16.6	39	38.5	62.5	38.5	15.6
	float	26	23.1	40.9	11.5	14.1	44	34.1	56.8	34.1	14.2
	double	26	23.1	10.9	11.5	28.1	68	39.7	55.8	39.7	24.4
BQ3	int32	21	28.6	46.7	21.4	23.4	51	44.1	65.6	44.1	18.4
	float	33	18.2	40.3	13.6	20.2	53	42.5	55.5	42.5	17.3
	double	33	18.2	40.3	13.6	40.3	89	45.5	64.8	45.5	28.3
BQ4	int32	23	34.8	49.8	26.1	32.3	63	47.6	57.1	47.6	19.6
	float	37	21.6	42.5	16.2	27.9	69	43.5	57	43.5	17.8

[a] Functional units, [b] General purpose registers

From the generated low-level code can be seen that only data path A was used and two instruction words supply .L, .S.,and .D functional units. Note that it is not complete listing of the 4-point FFT algorithm.

The following algorithms were selected for basic testing: matrix multiplication, FFT with real and complex inputs, and biquad filters. The duration and average occupancy of functional units and core registers are shown

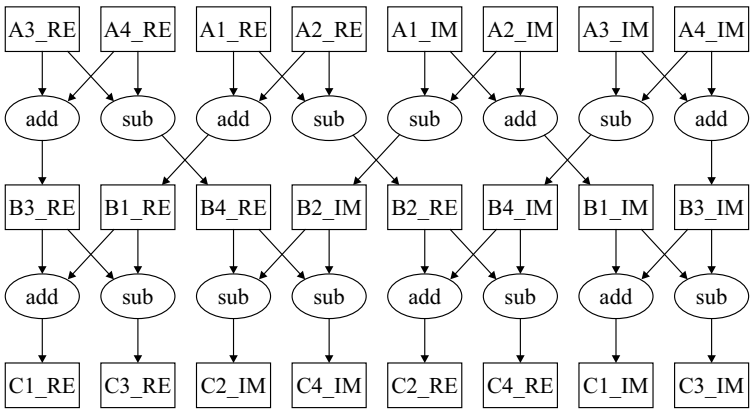

Fig. 8. Graphical representation of 4-point FFT with complex values and fixed-point representation

Table 2. Improvement of the execution time on algorithms with memory access according to used optimization

Algorithm	Data type	Number of CPU cycles						Improvement (%)			
		No optim.	Worst case	Node priority		Unit priority		Node priority		Unit priority	
				Cycle	Unit	Global	Map	Cycle	Unit	Global	Map
Mpy 2×2	int32	33	42	33	33	33	30	21.4	21.4	21.4	28.6
	float	32	42	32	32	32	32	23.8	23.8	23.8	23.8
	double	52	78	54	54	52	52	30.8	30.8	33.3	33.3
Mpy 3×3	int32	64	95	66	66	64	59	30.5	30.5	32.6	37.9
	float	63	95	63	63	63	63	33.7	33.7	33.7	33.7
FFT4R	int32	33	44	31	30	33	33	29.5	31.8	25.0	25.0
	float	34	44	30	30	34	34	31.8	31.8	22.7	22.7
	double	61	84	59	59	61	61	29.8	29.8	27.4	27.4
FFT4C	int32	39	55	40	40	39	39	27.3	27.3	29.1	29.1
	float	41	54	39	39	41	41	27.8	27.8	24.1	24.1
	double	73	102	70	70	73	73	31.4	31.4	28.4	28.4
FFT8R	int32	63	86	59	58	63	63	31.4	32.6	26.7	26.7
	float	66	90	58	58	66	66	35.6	35.6	26.7	26.7
FFT8C	int32	78	110	76	76	78	78	30.9	30.9	29.1	29.1
	float	85	111	77	77	85	85	30.6	30.6	23.4	23.4
BQ1	int32	24	27	24	24	24	24	11.1	11.1	11.1	11.1
	float	30	34	30	30	30	30	11.8	11.8	11.8	11.8
	double	40	49	40	40	40	40	18.4	18.4	18.4	18.4
BQ2	int32	39	51	38	38	39	38	25.5	25.5	23.5	25.5
	float	44	56	44	44	44	44	21.4	21.4	21.4	21.4
	double	68	92	68	68	68	68	26.1	26.1	26.1	26.1
BQ3	int32	51	71	50	50	51	49	29.6	29.6	28.2	31.0
	float	53	72	54	54	53	53	25.0	25.0	26.4	26.4
	double	89	126	91	91	89	89	27.8	27.8	29.4	29.4
BQ4	int32	63	92	62	62	63	63	32.6	32.6	31.5	31.5
	float	69	94	68	68	69	69	27.7	27.7	26.6	26.6

on the left-hand side of Tab. 1. The allocated usage was computed only from functional units and registers which were used. The total usage was computed for all hardware resources in one data path.

5.2 Values stored in data memory

The real-world verification assumes that input and output data are stored in data memory and accessed through pointers and can be comparable with high-level language functions.

Compared to the previous case, the final implementation contains more signals and operations, given by the multioperation nodes. These nodes are memory loading and storing via pointers which can be performed only by .D functional units. The right-hand part of Table I shows the performance of selected algorithms. The computing algorithm itself usually takes only about 1/3 of the total duration, the rest takes memory access process.

5.3 Node priority

Previous cases have shown the results of instruction mapping without any adjustment to their allocation. The pro posed techniques support several priorities during the mapping process, which should help to improve the generated code. The priority decision is based on the number of functional units, or the number of instruction cycles needed to execute the assigned instructions. Results from both approaches are displayed on the right-hand side of Tab. 2. Note that, the improvements are related to the algorithms with memory access from Tab. 1.

It can be seen that the node priority mapping does not propose any benefit for the matrix multiplications, because the needed instructions cannot be easily moved to another functional unit(s). It is because the multiplication operations can only be done by .M units and the memory access operations can be performed only by .D units.

On the other hand, the algorithms with the highest improve- ment contain a wide variety of instructions and it can be seen that FFT algorithms were executed up to 12 % faster if the floating-point representation of real sig-

Table 3. Execution time comparison with standard VLIW programming approaches

Algorithm	Mapping tool	Other approaches		
		Handwritten assembly	C code equivalent	DSPLib Library
Mpy 2×2	32	22	40	64
Mpy 3×3	63	45	110	-
FFT4R	30	19	46	-
FFT4C	39	24	80	-
FFT8R	58	34	123	-
FFT8C	77	42	205	145
BQ1	30	16	31	68
BQ2	44	30	58	-
BQ3	53	41	67	-
BQ4	62	49	86	-

Table 4. Execution time comparison with high-level language on scalar processor

Algorithm	Mapping tool	Other approaches	
		C code equivalent	CMSIS DSP library
Mpy 2×2	61	37	172
FFT4R	61	44	-
BQ1	29	22	70
BQ2	63	42	122
BQ3	92	55	175
BQ4	123	72	227

nal values was used. Therefore, the priority node improvements are critical issue. Here, number of used functional units and needed CPU cycles were observed.

5.4 Functional unit priority

Another method for improving of final code properties is the functional units mapping priority. It is based on statistics; how many potential operations can be performed on each functional unit. There are two options how the priority was set: global priority which is fixed for the architecture and dynamically changing unit according to the remaining unmapped nodes. The results are shown in Tab. 2. The performance was com- pared with the worst-case order for each examined algorithm. The difference of the execution time can be up to 37 %.

5.5 Comparison with other solutions

The results of proposed techniques were compared with other programming approaches and available library functions. The comparison is shown in Tab. 3 and criteria were the execution time given by the number of CPU cycles. Hand-written assembly code achieves the best results here, but can only be used for the simplest algorithms, as it requires significant programming experience with the target platform. The C code equivalent was compiled with TI C6000 compiler v7.3.1 and -o2 setting. It can be seen; the C code results were about 3 to 4-times slower. Only the 8-point complex FFT is available within the official TI DSP library for C66x architectures [16] and our result was about 1.7-times faster. For verification of proposed tool behavior, the ARM Cortex M4F scalar processor was defined and evaluated. The execution time comparison for biquad filters implemented by proposed techniques and by C language is shown in Tab. 4. The C code was compiled by ARM GCC v10.2.1, optimized with -o2 settings. As it can be seen, the final execution time for lower- order filters are close for both methods. The difference in higher-order filters was caused by the memory access when GCC is using a single PUSH/POP instruction for loading and storing multiple core registers.

6 Conclusion

This paper presented the instruction mapping techniques for generating low-level assembly codes. The techniques are primarily aimed to VLIW architectures to ease the optimization of the critical parts of DSP algorithms, from communication and signal processing domains. The proposed method comes with the program notation, based on the signal-flow graph. Here, any algorithm is described only with relations between variables and not the execution order. This approach can be a significant advantage especially on VLIW architectures in compare with the high-level compilers.

The performance of proposed method was verified by several DSP algorithms and compared to the handwritten assembly code, C code equivalent and DSP library provided by processor vendors. According to achieved results, the low-level assembly code has still the best performance for the simplest algorithms, but the proposed code exceeded the C code and provided DSP library functions.

The results were achieved by the new optimization process, using only a single data path. The second data path of each processor core can be used for multiple data processing and thus double the performance. On the other hand, because of memory access operations, the tool cannot be used for generating complex functions. Still, it is suitable for optimizing parts of low-level codes. Such parts can be reused on other architectures only by regenerating the target codes. This can not be possible if optimized parts were written directly.

Acknowledgements

Research described in this paper was supported by the Internal Grant Agency of the Brno University of Technology under project no. FEKT-S-20-6325.

References

[1] H. Sutter, "The free lunch is over a fundamental turn toward concurrency in software", *Dr. Dobbs Journal*, vol. 30, pp. 16–22, 2005.

[2] A. Wang, E. Killian, D. Maydan, and C. Rowen, "Hardware/software instruction set configurability for system-on-chip processors", *Proceedings of the 38th Annual Design Automation Conference, ser*, New York, NY, USA: Association for Computing Machinery, 2001, p. 184188, https://doi.org/10.1145/378239.378460.

[3] D. Liu, *Embedded DSP Processor Design: Application Specific Instruc- tion Set Processors*, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[4] H. Javaid and S. Parameswaran, *Pipelined Multiprocessor System-on- Chip for Multimedia*, Switzerland: Springer, 2012.

[5] M. Frankiewicz and A. Kos, "Microprocessor frequency control method under thermal and energy savings constraints", *IEEE Transactions on Components, Packaging and Manufacturing Technology*, no. 12, pp. 1755–1762, 2015.

[6] W. Wolf, "Multiprocessor system-on-chip technology", *IEEE Signal Processing Magazine*, no. 6, pp. 50–54, 2009.

[7] J. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*, 2005.

[8] S. Rajagopalan, S. Rajan, S. Malik, S. Rigo, G. Araujo, and K. Takayama, "A retargetable vliw compiler framework for DSPS with instruction-level parallelism", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, no. 11, pp. 1319– 1328, 2001.

[9] Y. Zeng, A. Gupta, and S. Malik, "Automatic generation of architecture- level models from RTL designs for processors and accelerators", *2022 Design, Automation & Test Europe Conference & Exhibition (DATE)*, pp. 460–465.

[10] C. Fang, Z. Zhang, X. You, and C. Zhang, "Automatic hardware design tool based on reusing transformation", *2019 IEEE 13th International Conference on ASIC (ASICON)*, pp. 1–4.

[11] Y. Xing, B.-Y. Huang, A. Gupta, and S. Malik, "A formal instruction- level gpu model for scalable verification", *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 18, 2014, pp. 198–204.

[12] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance GPU code generation", *2017 IEEE/ACM International Symposium on Code Generation and Optimization* (CGO), pp. 74–85.

[13] K. Ohno, T. Kamiya, T. Maruyama, and M. Matsumoto, "Automatic optimization of thread mapping for a GPGPU programming framework, *Second International Symposium on Computing and Networking*, 2014.

[14] E. Schnetter, "Performance and optimization abstractions for large scale heterogeneous systems in the cactus/chemora framework", *2013 Extreme Scaling Workshop (xsw 2013), 2013*, 33–42.

[15] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus", *SC 13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013*, 1–12.

[16] T. Instruments, "TMS320C67x DSP Library Programmers Reference Guide", *Reference Guide*, [Online]. Available: https://www.ti.com/lit/ug/spru657c/spru657c.pdf.

[17] A. Limited, "Common Microcontroller Software Inter- face Standard (CMSIS)", *ARM*, https://developer.arm.com/tools-and-software/embedded/cmsis.

[18] W. von Hagen, *The Definitive Guide to GCC*, 2006.

[19] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2011.

[20] T. A. Mogensen, *Introduction to Compiler Design*, Switzerland: Springer, 2017.

[21] R. Mego and T. Fryza, "A tool for VLIW processors code optimizing", *13th International Conference on Computer Engineering and Systems* (ICCES), pp. 601–604, 2018.

[22] T. Instruments, "TMS320C66x DSP CorePac User Guide", *User Guide, 2013*, https://www.ti.com/lit/ug/sprugw0c/sprugw0c.pdf.

[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Mathematics of Computation*, pp. 297–301, 1965.

[24] T. Fryza and R. Mego, "Low level source code optimizing for sin- gle/multi/core digital signal processors", *23rd International Conference Radioelektronika* (RADIOELEKTRONIKA, 2013), 288291, 2013.

[25] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo, "A technique for drawing directed graphs", *IEEE Transactions on Software Engineering*, no. 3, pp. 214–230, 1993.

Received 14 September 2022