OUKA
The University of Osaka
Institutional Knowledge Archive

Title	Studies on Performance Evaluation and Design Productivity Improvement for Digital Signal Processing Systems
Author(s)	Kumura, Takahiro
Citation	大阪大学, 2012, 博士論文
Version Type	VoR
URL	https://hdl.handle.net/11094/26850
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

博甲 15549

Studies on Performance Evaluation and Design Productivity Improvement for Digital Signal Processing Systems

January 2012

Takahiro KUMURA

Studies on Performance Evaluation and Design Productivity Improvement for Digital Signal Processing Systems

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2012

Takahiro KUMURA

Publications

Journal Article (Refereed)

[J1] Takahiro Kumura, Norio Kayama, Shinichiro Shionoya, Kazuo Kumagiri, Takao Kusano, Makoto Yoshida, Masao Ikekawa, Ichiro Kuroda, and Takao Nishitani, "*Performance evaluation of the AV CODEC on a low-power SPXK5SC DSP core*," IEICE Transactions on Information and Systems, Vol. E88-D, No. 6, pp. 1224–1230, June 2005.

[J2] Takahiro Kumura, Soichiro Taga, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Software development tool generation method suitable for instruction set extension of embedded processors*," IPSJ Transactions on System LSI Design Methodology, Vol. 3, pp. 207–221, August 2010.

Invited Magazine Article (Refereed)

[M1] Takahiro Kumura, Masao Ikekawa, Makoto Yoshida, and Ichiro Kuroda, "*VLIW DSP for mobile applications*," IEEE Signal Processing Magazine, Vol. 19, No. 4, pp. 10–21, July 2002.

Conference Papers (Refereed)

[C1] Takahiro Kumura, Daiji Ishii, Masao Ikekawa, Ichiro Kuroda, and Makoto Yoshida, "*A low-power programmable DSP core architecture for 3G mobile terminals*," Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 2, No. ITT-4-3, pp. 1017–1020, May 2001.

[C2] Masao Ikekawa, Masaki Hori, Kohei Nadehara, Takahiro Kumura, Makoto Yoshida, Ichiro Kuroda, and Takao Nishitani, "*Multimedia signal processor for mobile applications*," Proceedings of IEEE International Conference on Multimedia and Expo (ICME), pp. 55–59, August 2001.

[C3] Takahiro Kumura, Norio Kayama, Shinichiro Shionoya, Kazuo Kumagiri, Takao Kusano, Makoto Yoshida, and Masao Ikekawa, "*AV CODEC prototype system using the low-power SPXK5SC DSP core*," Proceedings of Workshop on Signal Processing Systems (SiPS), pp. 69–74, August 2003.

[C4] Takahiro Kumura, "*Customizing CDT's registers/memory/disassembly views for assembly programming on an in-house DSP*," EclipseCon, March 2008.

[C5] Takahiro Kumura, Yuichi Nakamura, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Model based parallelization from the Simulink models and their sequential C codes*," Proceedings of Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), March 2012, to appear.

[C6] Takahiro Kumura, Soichiro Taga, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Automatic generation of GNU Binutils and GDB for ASIP cores based on plug-in method*," Proceedings of Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), March 2012, to appear.

Domestic Conference Papers

[D1] Takahiro Kumura, Masao Ikekawa, and Ichiro Kuroda, "*Instructions of multiply-accumulate with rounding for exploitation of data bus extension on 16-bit fixed point DSP*," Proceedings of IEICE General Conference, Vol. A-3-3, March 1999 (in Japanese).

[D2] Takahiro Kumura, Masao Ikekawa, and Ichiro Kuroda, "*Using rounding multiply-accumulate instruction with separated accumulators in FFT*," Proceedings of IEICE Society Conference, Vol. A-4-7, September 1999 (in Japanese).

[D3] Takahiro Kumura, Masao Ikekawa, Ichiro Kuroda, and Toshihiro Hayata, "*Performance of path search method with AFC for W-CDMA*," Proceedings of IEICE General Conference, Vol. B-5-54, March 2000 (in Japanese).

[D4] Takahiro Kumura, Toshihiro Hayata, Masao Ikekawa, and Ichiro Kuroda, "*Frequency offset estimation using binary search in very-low CNR environments*," Proceedings of IEICE Society Conference, Vol. B-5-26, October 2000 (in Japanese).

[D5] Takahiro Kumura, Masao Ikekawa, and Ichiro Kuroda, "*Fast viterbi decoding on a general purpose DSP with viterbi instructions*," Proceedings of IEICE General Conference, Vol. A-4-40, March 2001 (in Japanese).

[D6] Masao Ikekawa, Takahiro Kumura, Daiji Ishii, Makoto Yoshida, and Ichiro Kuroda, "*A low-power 4-way VLIW DSP core architecture*," Proceedings of IEICE Society Conference, Vol. A-4-43, August 2001 (in Japanese).

[D7] Takahiro Kumura, Masao Ikekawa, Makoto Yoshida, and Ichiro Kuroda, "*Performance analysis of a low-power 4-way VLIW DSP core*," Proceedings of IEICE Society Conference, Vol. A-4-44, August 2001 (in Japanese).

[D8] Takahiro Kumura, Masao Ikekawa, and Ichiro Kuroda, "*Source code translation from a conventional DSP to a VLIW DSP*," Proceedings of IEICE Society Conference, Vol. A-4-7, August 2002 (in Japanese).

[D9] Takahiro Kumura, Masao Ikekawa, and Ichiro Kuroda, "*AV CODEC evaluation system on the low-power SPXK5 DSP core*," Proceedings of IEICE General Conference, Vol. A-4-14, March 2003 (in Japanese).

[D10] Takahiro Kumura and Masao Ikekawa, "*DSP architecture for variable length decoding*," Proceedings of IEICE Symposium on Signal Processing, Vol. C6-2, November 2005 (in Japanese).

[D11] Takanori Morimoto, Takahiro Kumura, Nagisa Ishiura, Masao Ikekawa, and Masaharu Imai, "*VLIW extension of software development environment construction tool ArchC*," IEICE Technical Report VLD2007-134, Vol. 107, No. 415, pp. 95–100, January 2008 (in Japanese).

[D12] Syohei Yoshida, Takahiro Kumura, Nagisa Ishiura, Masao Ikekawa, and Masaharu Imai, "*Automatic generation for GCC for instruction set extension on configurable processor*," IPSJ Technical Report ARC, Vol. 2008-ARC-176, No. 1, pp. 29–34, January 2008 (in Japanese).

[D13] Tetsuya Yamamoto, Takahiro Kumura, Masao Ikekawa, Nagisa Ishiura, and Masaharu Imai, "*Optimum code scheduling for VLIW DSP SPXK5 considering conditional execution*," IEICE Technical Report VLD2007-134, Vol. 108, No. 478, pp. 95–100, January 2008 (in Japanese).

[D14] Yuji Kunitake, Takahiro Kumura, and Hiroto Yasuura, "*A case study on instruction set extension for variable length decoding on a custom processor*," IPSJ Technical Report ARC, Vol. 2010-ARC-187, No. 21, pp. 1–6, January 2010 (in Japanese).

[D15] Takahiro Kumura, Soichiro Taga, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Software development tool generation method suitable for instruction set extension of embedded processors*," IEICE Technical Report VLD2009-120, Vol. 109, No. 462, pp. 127–132, March 2010 (in Japanese).

[D16] Soichiro Taga, Takahiro Kumura, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Automatic retargeting of binutils and GDB based on plugin method*," IEICE Technical Report VLD2010-95, Vol. 110, No. 360, pp. 69–74, January 2011 (in Japanese).

[D17] Takahiro Kumura, Masato Edahiro, Yuichi Nakamura, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Parallel C code generation from Simulink models*," Proceedings of IPSJ Workshop on Embedded Technologies and Network (ETNET), CPSY2010-80, Vol. 110, No. 473, pp. 303–308, March 2011 (in Japanese).

Summary

This thesis studies performance evaluation and design productivity improvement for digital signal processing systems. To realize digital signal processing systems using programmable processors, one of most important things is to explore an optimized combination among digital signal processing algorithms, processor architectures, and software development tools. From the processor's point of view, evaluating processor performance against target applications and providing software development tools at early design stages for the evaluation play important roles in this exploration of the optimized combination. As LSI designs have grown larger, overall performance evaluation on real workloads becomes difficult on system LSIs integrated with a large number of hardware components and processor cores before fabrication. Software simulators are too slow to deal with real-time workloads, and full hardware prototyping is unable to respond well to design improvements. As the calculation amount required in digital signal processing applications has been increasing such that a single processor cannot afford to handle, performance evaluation becomes difficult since workload must be distributed among multiple processors through parallelization of a target application. With these as background, important things to improve the design productivity of processors for digital signal processing applications are software development tools available at an early design stage for performance evaluation, overall performance evaluation on real workloads before LSI fabrication, and software parallelization of calculation-requiring digital signal processing applications to ease performance evaluation. First, this thesis describes a design experience of a digital signal processor core as an example of hand-optimized performance evaluation at a processor design stage. Next, to improve the efficiency of overall performance evaluation, this thesis describes a method for rapidly verifying and evaluating overall performance on real-time workloads of LSIs before fabrication. The developed method makes it possible to emulate the target LSI composed of the processor core shown in the design experience and many peripherals running at a scaled-down operating frequency 1/3 and to evaluate audio and video processing on the LSI for actual situations. Then, to provide software development tools at an early design stage, this thesis describes a method to generate software development tools, and demonstrates that a generated compiler gives as good performance such that can be used for performance evaluation. Finally, to increase the productivity in terms of parallelization of digital signal processing algorithm, this thesis describes a method to generate parallel C code from models that represent behavior of digital signal processing applications as block diagrams. The total work of this thesis has made improvement on major three aspects in terms of design productivity of processors and software for digital signal processing systems.

Acknowledgements

I would sincerely like to thank Professor Masaharu IMAI, Professor Takao ONOYE, Professor Nagisa ISHIURA, and Associate Professor Yoshinori TAKEUCHI for having supervised me during this work in Osaka University.

The development of the new DSP core described in chapters 2 and 3 has been made possible by the hard work of a very large group of people at NEC Corporation and former NEC Electronics Corporation, particularly, Mr. Masao IKEKAWA, Dr. Ichiro KURODA, Mr. Makoto YOSHIDA, and Professor Takao NISHITANI, to whom I express my sincere appreciation.

The work described in chapter 4 has been made possible by collaboration with, appropriate assistance of, and suggestions of Mr. Soichiro TAGA, Mr. Syohei YOSHIDA, Ms. Aiko WATANABE, Dr. Takuji HIEDA, and Dr. Yuki KOBAYASHI, to whom I express my sincere appreciation.

The research work described in chapter 5 would have not been possible without appropriate assistance and suggestions of Professor Masaharu IMAI, Associate Professor Yoshinori TAKEUCHI, Professor Nagisa ISHIURA, Professor Masato EDAHIRO, and Dr. Yuichi NAKAMURA.

I would like also to thank all my current and former colleagues in NEC Corporation, current and former members of Integrated System Design Laboratory in Osaka University, and co-authors of my publications.

Finally, but not least, I would like to thank my wife Asuka for her understanding and support during this work. Her support and encouragement were in the end what made this thesis possible.

Contents

Publications i
Journal Article (Refereed) i
Invited Magazine Article (Refereed) i
Conference Papers (Refereed) i
Domestic Conference Papers ii

Summary iv

Acknowledgements v

1 Introduction 1
1.1 Digital signal processing devices and implementation 1
1.2 Processor design for digital signal processing 2
1.3 Overall performance evaluation before fabrication 3
1.4 Software development tools for early performance evaluation 3
1.5 Productivity of software parallelization for multiple processors 4
1.6 Thesis organization 4

2 Design of A Digital Signal Processor Architecture 5
2.1 Demand for high-performance and low-power programmable digital signal processors 5
2.2 Architecture 6
2.3 Orthogonal instruction set 8
2.3.1 Variable length instruction packet 11
2.3.2 Saturation mode 11
2.3.3 Special instructions for particular applications 13
2.4 Application benchmarks 14
2.4.1 Delayed least mean square adaptive filtering 14
2.4.2 Viterbi decoding 19
2.4.3 MPEG-4 video encoding and decoding 23
2.5 Summary 25

3 Performance Evaluation Method Before SoC Integration 26
3.1 System performance evaluation for LSIs using processors 26
3.2 SPXK5 and SPXK5SC 27

3.3 AV CODEC prototype system . . . 28
3.4 Implementation of AV CODEC . . . 30
3.5 Performance evaluation . . . 32
3.6 Architecture design based on the emulation system . . . 36
3.7 AV CODEC performance of μPD77050 . . . 37
3.8 Summary . . . 38

4 Software Development Tool Generation for Architecture Design 40
4.1 ASIPs and their software development tools . . . 40
4.2 Software development tool generation using plugins . . . 41
4.2.1 Tool generation flow . . . 42
4.2.2 Structure of the generated plugins . . . 42
4.2.3 Internal tool flow working with the plugins . . . 45
4.2.4 Assembling and encoding new instructions . . . 45
4.2.5 Machine description of new instructions . . . 47
4.2.6 Adding intrinsic functions on the GCC . . . 49
4.3 Instruction set description using XML . . . 50
4.3.1 Register definition . . . 51
4.3.2 Instruction definition . . . 52
4.3.3 Instruction behavior definition . . . 54
4.4 Experiment . . . 55
4.4.1 Generated toolchain for SIMD extension . . . 56
4.4.2 Code generation using intrinsic functions . . . 56
4.5 Related work and discussion . . . 59
4.6 Summary . . . 64

5 Productivity improvement on parallelization of digital signal processing algorithm for multiple processors 67
5.1 Multicore processors and software parallelization . . . 67
5.2 Related work . . . 69
5.3 Parallel C code generation from Simulink models . . . 70
5.4 Analyzing C code . . . 72
5.5 Analyzing model . . . 72
5.6 Flattening block hierarchy . . . 73
5.7 Breaking loop structures . . . 73
5.8 Generating parallel C code . . . 75
5.9 Experiment . . . 76
5.9.1 Audio equalizing . . . 76
5.9.2 Lane detection . . . 77
5.10 Discussion . . . 79
5.11 Summary . . . 79

6 Conclusion 82

List of Figures

1.1 Exploration of optimized combination of algorithms, processor architectures, and software development tools to realize digital signal processing systems. 2

2.1 The SPXK5 block diagram showing seven functional units and four buses. 8
2.2 Pipeline diagram. 9
2.3 Chip micrograph of the SPXK5 and its memory banks. 10
2.4 Instruction packets containing one to four instructions. 13
2.5 C program for the Delayed LMS algorithm. 16
2.6 Two possible patterns of MADDR 16-bit MAC instruction. 17
2.7 Innermost loop code segment of an assembly program corresponding to Figure 2.5. 18
2.8 Unrolled innermost loop of Figure 2.5. 19
2.9 Accelerated assembly code segment for the DLMS algorithm. 20
2.10 Add-compare-select operations. 21
2.11 ACS operations on the SPXK5. 22
2.12 Assembly code segment for four ACS operations. 22
2.13 MAE calculation flow diagram for the SPXK5. 24
2.14 MPEG-4 encoder/decoder cycle counts (SP@L2, 352x288 pixels, 15 frames/sec). . 25

3.1 Block diagram of the SPXK5. 29
3.2 Block diagram of the SPXK5SC. 30
3.3 Block diagram of the AV CODEC prototype system. 31
3.4 AV CODEC prototype system. 32
3.5 Average cycles per second of each function during AV CODEC execution. 34
3.6 Average bus traffic per second of each bus master during AV CODEC execution. . 35
3.7 Block diagram of the μPD77050. 37
3.8 AV CODEC performance of μPD77050. 38

4.1 Concept flow underlying tool generation. 43
4.2 Generating plugins for GNU Binutils, GDB, and GCC. 44
4.3 Tool internal flow for enhanced assemblers, disassemblers, linkers, and simulators working with plugins. 46
4.4 Assembling instructions on the plugins. 48
4.5 A generated instruction pattern. 49
4.6 Instruction field structure of MYADD. 52

4.7 Example of an XML document with additional ISA specification. 53
4.8 Behavior description of instruction MYADD written in C language. 55
4.9 Block diagram of the V850 processor with SIMD extension. 56
4.10 Increase in the number of executed instructions of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes. 60
4.11 Increase in the code size of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes. 61
4.12 FIR filtering functions: C language. 65
4.13 FIR filtering functions: assembly language. 66

5.1 The process to generate parallel C code from a Simulink model. 70
5.2 The flow chart of task execution. 71
5.3 Flattening block hierarchy. 74
5.4 How to break a loop structure. 75
5.5 Task dependency graph extracted from the audio equalizing model. 78
5.6 Task dependency graph extracted from the lane detection model. 80
5.7 Histogram of task execution time ratio for the audio equalizing. 81
5.8 Histogram of task execution time ratio for the lane detection. 81

List of Tables

2.1 Chip characteristics of the SPXK5. 9
2.2 Instruction set summary. 12
2.3 General DSP function benchmark comparison. 15

3.1 Functional specification of the AV CODEC prototype system. 33
3.2 Memory allocation of the prototype system. 33
3.3 AV CODEC parameters for performance evaluation. 35
3.4 Estimated cycle counts and bus traffic of the AV CODEC on 250-MHz SPXK5SC and 83-MHz AHB. 35
3.5 The μPD77050 specification. 39

4.1 Architecture summary of the V850 microcontroller with SIMD extension. 57
4.2 The code amount of the generated plugins for the SIMD extension. 57
4.3 Comparison among tool generation methods using the GNU toolchain. 63
4.4 Lines of source codes: (a) original C code, (b) modified C code using intrinsic functions, (c) assembly code generated from (b), and (d) hand-optimized assembly code. 64

5.1 Experiment results of generating parallel C code from Simulink models. 77

Chapter 1

Introduction

Digital signal processing technology has been evolving in recent decades in accordance with the advance of semiconductor technology. The process of sampling and quantizing analog signals in analog-to-digital conversion makes it possible to handle the signals without any degradation and to process them in mathematical ways such as compressing, decompressing, eliminating noises, enhancing, predicting, and so on. Initially, the main applications of digital signal processing were voice coding, data modems, and echo cancellation. Then, the advances of the two technologies have made it possible to realize baseband processing in faster wired/wireless communication systems, audio encoding/decoding systems, and video encoding/decoding systems.

1.1 Digital signal processing devices and implementation

There are many methods and devices to realize digital signal processing algorithms written in the languages suited to implementation [1, 2]. It is very important to use suitable methods and devices for different purposes in terms of power consumption, required performance, and programmability. Dedicated circuits such as application specific integrated circuits (ASICs) have strength in terms of speed of operating frequency and power consumption while they do not have programmability. Programmable processors make it possible to change algorithms of digital signal processing by using software while they generally consume more power than ASICs and provide less performance than ASICs. Among several types of programmable devices ranging from central processing units (CPUs) [3] to field programmable gate arrays (FPGAs) [4], this thesis focuses on the design productivity of processors and software for digital signal processing applications.

Programmable processors, as mentioned above, can handle different digital signal processing applications if software for them is given. After processors are fabricated, software can change how a processor works. If several different applications are implemented on a single device and they do not work concurrently, processors are generally a better choice in terms of semiconductor area efficiency compared with ASICs. To realize digital signal processing applications using programmable processors, one of most important things is to explore an optimized combination among digital signal processing algorithms, processor architectures, and software development tools as shown in Figure 1.1.

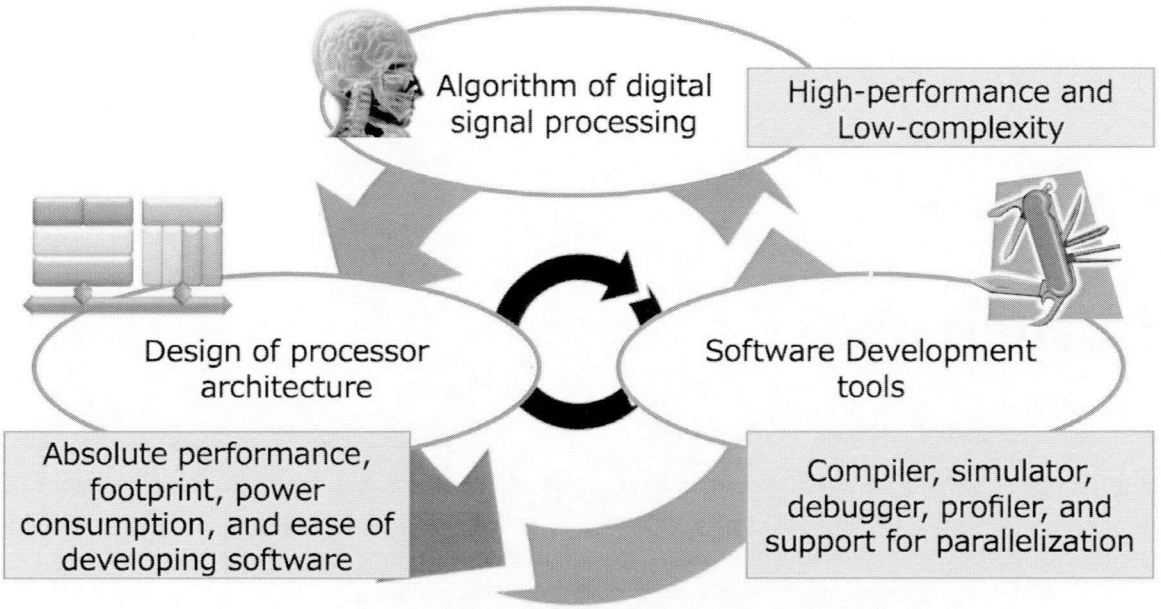


Figure 1.1: Exploration of optimized combination of algorithms, processor architectures, and software development tools to realize digital signal processing systems.

From the processor's point of view, efficient evaluation of processors' performance against target applications and software development tools at an early design stage for this evaluation play important roles in this exploration of the optimized combination. As the calculation amount required in digital signal processing applications has been increasing such that a single processor cannot afford to handle, performance evaluation becomes difficult since workload must be distributed among multiple processors through parallelization of a target application. With these as background, important things to improve the design productivity of processors for digital signal processing systems are: (1) software development tools available at an early design stage for performance evaluation, (2) overall performance evaluation on real workloads before LSI fabrication, and (3) software parallelization of calculation-requiring digital signal processing applications to ease performance evaluation. Therefore, this thesis has studied these major aspects in terms of design productivity of processors and software for digital signal processing systems.

1.2 Processor design for digital signal processing

To efficiently implement different kinds of digital signal applications on a programmable processor, processors to be designed should be assessed their performance of conceivable applications. In the past, aptitude for digital signal processing was judged based on only how many multiply-accumulate

operations are performed in a second. Now the situation has changed. Since algorithms of digital signal processing get more diverse and complex, programmable processors should be designed so as to efficiently perform many kinds of operations in addition to multiply accumulation. With regard to this point in performance evaluation during processor design, this thesis describes a design experience of a digital signal processor core in chapter 2.

1.3 Overall performance evaluation before fabrication

As LSI designs have grown larger, estimation of overall performance on real workloads becomes difficult on system LSIs integrated with a large number of hardware components and processor cores. It is very important to evaluate the overall performance of target LSIs on real workloads before actual LSI fabrication. However, software simulators, while relatively inexpensive, versatile, and accurate, are too slow to deal with real-time workloads, e.g., audio/video signals, and full hardware prototyping is unable to respond well to design improvements. In this regard, as an intermediate approach between software simulation and full hardware prototyping, this thesis describes a method for rapidly verifying and evaluating overall performance on real-time workloads of LSIs before fabrication in chapter 3.

1.4 Software development tools for early performance evaluation

At the design experience of a digital signal processor architecture described in chapter 2, at an early stage of the architecture design, only simple software development tools not including any C compilers was developed for early performance evaluation. This made the early performance evaluation of the architecture inefficient in particular for larger applications because any C compiler was not available at an early stage of the architecture design. As a remedy for this inefficiency of the early performance evaluation, a hybrid programmable processor architecture composed of a base processor and application specific functional units is promising, which is an application specific instruction set processor (ASIP) emerged in the 1990s [5]. ASIPs are designed so as to have specialized instruction set that provide high perform on specific operations required in a particular application. Since the fundamental instruction set of a base processor is not changed, it is possible to optimize the C compiler for the base processor in advance. However, there is a problem. There has not been an efficient way to exploit the processor's existing GNU toolchain as a base compiler for ASIPs. The GNU toolchain is an open-source and a de-fact toolchain in the field of embedded software development. Since the GNU toolchain supports many kinds of processors, it is very suitable to generate software development tools for the ASIPs based on existing embedded processors. In addition, source availability of the GNU toolchain gives chances for ad-hoc modification that might be required at early design stages of ASIPs. To address this problem, this thesis describes a method of software development tool generation for instruction set extension of existing embedded processors in chapter 4.

1.5 Productivity of software parallelization for multiple processors

The calculation amount required in digital signal processing applications has been increasing such that a single processor cannot afford to handle. The reasons may include enlargement of image resolution from standard definition (SD) to high definition (HD), faster bit-rate of mobile wireless communication, and complication of signal processing algorithms for rich functionalities. To realize such calculation-requiring applications, using multiple processors on a single LSI attracts attention [6]. Platforms with multiple cores are now prevalent everywhere from desktops and graphics processors to laptops and embedded systems. However, writing efficient digital signal processing applications working in parallel that utilize the computing capability of many processing cores still require much effort. Although it is very important to parallelize software working on a multicore processor to exploit its inherent performance, parallelization is one of most difficult optimization processes and should be eased by something useful such as language, compiler, and design tool. To increase the productivity in terms of parallelization of digital signal processing algorithm, this thesis describes a method to generate parallel C code from models that represent behavior of digital signal processing applications in chapter 5.

1.6 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 describes a design experience of a processor architecture suited to signal processing for communication and multimedia applications. Chapter 3 describes a method for rapidly verifying and evaluating overall performance on real-time workloads of system LSIs by using an intermediate approach between software simulation and full hardware prototyping, hardware emulation using FPGAs. Chapter 4 describes a method of software development tool generation for architecture design. Chapter 5 proposes a method to improve productivity on parallelization of digital signal processing algorithm for multiple processors. Finally, chapter 6 concludes this thesis and summarizes future work.

Chapter 2

Design of A Digital Signal Processor Architecture

This chapter addresses the design of a digital signal processor (DSP) core architecture with low power dissipation for use in third-generation (3G) mobile terminals. To obtain higher performance, the DSP core employs a 4-way VLIW (very long instruction word) approach, as well as a dual-multiply-accumulate (dual-MAC) architecture with good orthogonality. It is able to perform both video and speech CODEC for 3G wireless communications at 384 Kbits/sec with a power consumption of approximately 50 mW. This chapter presents an overview of both the DSP core architecture and a DSP instruction set, and it also gives some application benchmarks.

2.1 Demand for high-performance and low-power programmable digital signal processors

The fast bit-rate, complex modem processing required by next generation mobile terminals (e.g., rake combining, channel equalization, and forward error correction, etc.) will necessitate signal processing power that can only be provided by high-performance embedded devices. Further, multimedia applications (e.g., video CODECs [7, 8], speech/audio CODECs, echo cancellers, speech recognition systems, etc.) will also need to be executed concurrently on such terminals.

This means that the development of programmable processors will be particularly important because their flexibility and shorter development periods will be crucial to the success of both enhanced modem processing and enhanced multimedia applications. That is to say, first of all, with respect to modem processing, the flexibility provided by programmable processors will be needed in order to respond to further changes in communication protocols and the need for algorithm improvements. Multimedia applications have an even stronger requirement for flexibility. On the same handheld terminal, for instance, a videophone would require encoding and decoding of video and speech/audio, while the browser would require only decoding, but at a far higher resolution for the video, and at a hi-fi level for the audio. To be able to handle all the possible combinations in which these two applications might be used separately or concurrently, a dedicated processor would inevitably need to provide a far greater amount of processing power than would be needed with a

programmable processor, whose flexibility could eliminate much resource waste.

Among the various existing programmable processor architectures [9], programmable digital signal processors (DSPs) seem particularly well suited to handheld terminals because of their energy efficiency. In this regard, recent processors in such programmable DSP architectures as very long instruction word (VLIW) based architectures are particularly noteworthy because they enable the development of high-level language compilers that generate efficient codes [10, 11], which will be especially important in reducing development time for the applications to be used in DSPs. Although the superscalar architectures that have orthogonality in their instruction set would also be suitable for high-level language compilers, VLIW architectures would require less hardware complexity because the superscalar architectures generally need a hardware logic block to find instruction-level parallelism in instruction codes. Unfortunately, however, almost all early VLIW-based DSPs have been developed for high-end applications, and they consume too much power for use with handheld terminals.

A new DSP core has been developed for handheld terminals, the SPXK5 [12]. It is a 4-way VLIW DSP core with a highly orthogonal instruction set. It achieves high-performance and flexibility, and it is compatible with high-level languages. In addition, its architecture features low-power consumption. This chapter describe the SPXK5 architecture and its performance in DSP applications.

This chapter also consider the question of application-specific enhancements. Such architecture enhancements as add-compare-select instructions or co-processors for the Viterbi decoding algorithm are employed in some recent programmable DSPs, and for video CODECs, other architectures include either single-instruction multiple-data (SIMD) instructions or media co-processors [13–15]. While such application-specific enhancements are valuable when their applications are actually in use, they do nothing to enhance the performance of other applications, and the more they are added, the greater the increase in chip-size and energy requirements. In other words, for handheld terminals, such enhancements need to be chosen in a careful and balanced way. This has been done in developing the SPXK5, in which a wide range of signal processing algorithms are efficiently implemented.

2.2 Architecture

The SPXK5 is a 16-bit general-purpose DSP core based on the incorporation of a VLIW architecture into the NEC μPD7701x architecture [16]. Its low-power consumption and high-performance make it suitable for use in handheld terminals. Figure 2.1 shows a block diagram of the SPXK5, which contains register files, buses, several control blocks, and seven functional units. The functional units consist of two multiply-accumulate (MAC) units for 16-bit by 16-bit multiplication and 40/16-bit accumulation; two arithmetic logical units (ALU) for addition/subtraction, shift, and logical operations; two data address units (DAU) for load and store; and one system control unit (SCU) for branch, zero overhead looping, and conditional execution.

The SPXK5 has been designed so that up to four of the seven units can work in parallel during the same clock cycle. With an instruction length of 16 bits, this means that the SPXK5's instruction fetch length is 64 bits. While the ability to issue instructions to all seven functional units might be desirable in certain respects, this would require the SPXK5 either to have such a large memory

bandwidth or to operate its instruction bus at such high speeds as to increase power consumption beyond desired levels. With a maximum of four units operable in parallel, however, power consumption can be kept low enough for use in handheld terminals.

The SPXK5 is based on a load/store architecture. Data from the memory connected to it is read into its registers, and operations are executed on the basis of that data. The results of these operations are input in the registers and then written back into the memory. The SPXK5 has eight general-purpose 40-bit registers (R0-R7). To avoid increasing power consumption in the SPXK5, the number of general-purpose 40-bit registers (8), which was found in the original μPD7701x architecture, has not been increased, but this insufficient number has been compensated for by allowing each 40-bit register also to be so divided as to provide two 16-bit register portions (R0H, R0L, ...). Since such portions can be handled by almost all instructions, the number of registers is, in effect, roughly doubled, greatly enhancing performance with no significant increase in power consumption.

The SPXK5 also has eight address registers (DP0-DP7) and eight corresponding offset registers (DN0-DN7). The address registers are 32 bits wide, and the offset registers are 16 bits wide. The address registers indicate addresses to access memory and can be used for address calculation in such simple arithmetic operations as addition and subtraction, which eliminates the need to use the general-purpose registers. Each of the offset registers has an offset value for use in modifying its corresponding address register.

Outside the SPXK5, instructions and data architecturally reside in a single unified 32-bit memory space which consists of a number of memory banks. Instructions and data are stored separately, i.e., in different banks from one another in order to make sure that the SPXK5 can concurrently access both of them. That is to say, while they are in the same single 32-bit memory space, they are kept in physically separate memory banks. Memory banks to store instructions are all connected to a single 64-bit instruction bus and each memory bank to store data is connected to both of two 32-bit data buses. With three buses, the SPXK5 can access three different memory banks at the same time. Specifically, it can fetch a 64-bit instruction word and can load/store two 32-bit data items during the same clock cycle. Through a 32-bit data bus, two 16-bit data items located adjacent to one another in a memory bank can be transferred to registers as a single 32-bit data item.

For faster operational frequency, the SPXK5 has a six-stage deep pipeline: instruction fetch (IF), dispatch queue (DQ), decode (DE), DP register update (DP), and execution phases 1 (EX1) and execution phase 2 (EX2) shown in Figure 2.2. Instructions for both the ALU and the SCU are executed during the EX1 stage. MAC operations are executed during both the EX1 and the EX2 stages. Load/store operations are executed during both the DP and the EX1 stages. The DP register update operation for post-modification load/store is executed during the DP stage, and the address calculation for pre-modification load/store is executed during the DE stage. Since each instruction has the different execution stages, there is the possibility that the result of a calculation might not be available at the next clock on some combinations of instructions (i.e., a pipeline hazard). In order to avoid pipeline hazards, such code generators as compilers and assemblers are used to control instruction scheduling so as to maintain the correctness of programs. This software solution eliminates the need for the SPXK5 to contain an interlocking system for register files as a hardware solution; this would need to work every cycle and would dissipate considerable power. Because the SPXK5 relies on software for the optimized instruction scheduling that is required by superscalar and VLIW architectures, it requires fewer circuits and consumes less power.

Table 2.1 summarizes the chip characteristics of the SPXK5. A 0.13-micron logic process was used to fabricate the SPXK5, which measures 2.56 square millimeters and can operate at 250 MHz, equivalent to 1000 MIPS. Its energy efficiency reaches 0.05 mW/MIPS at 0.9 volts. Figure 2.3 shows a chip micrograph of the SPXK5 and its memory banks, which consist of 64 Kbytes instruction memory, 64 Kbytes data memory, and 16 Kbytes instruction cache.

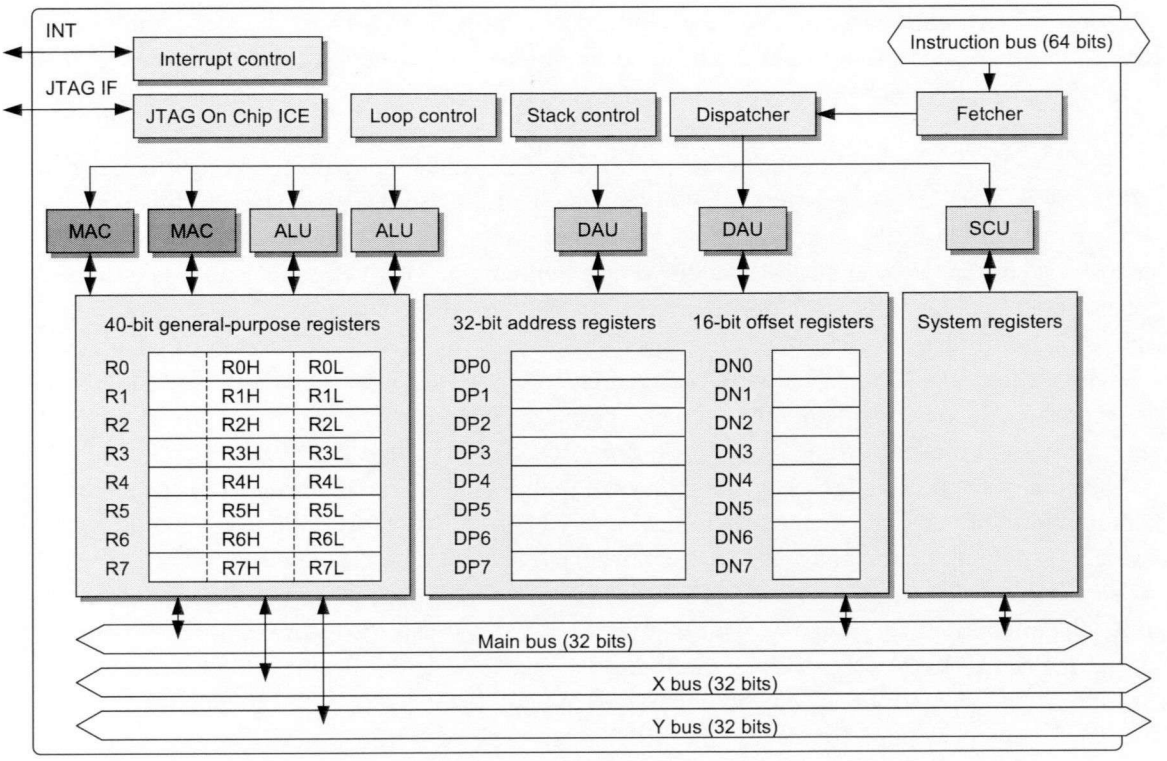
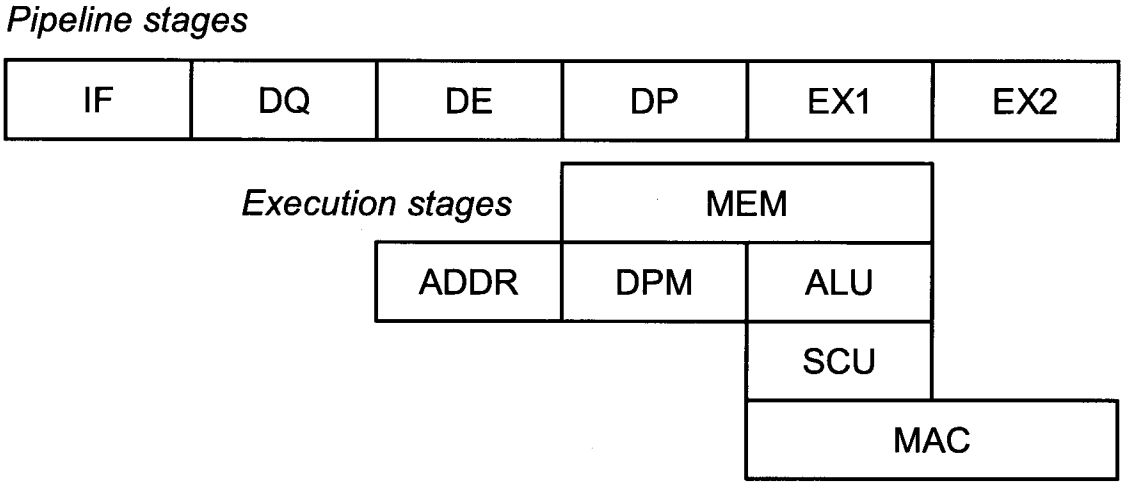


Figure 2.1: The SPXK5 block diagram showing seven functional units and four buses.

2.3 Orthogonal instruction set

The SPXK5 can issue up to four instructions in parallel during the same clock cycle, and the total length of the instructions can be as great as 64 bits. Users are free to choose instructions for parallel issue so long as the number and total length of those instructions fall within these limits. In addition, all arithmetic, logical, and store operations other than MAC operations are single-cycle operations. These fewer restrictions on parallel instructions (high orthogonality) and single-cycle operations are very important features in the SPXK5 instruction set architecture because they allow users and



Pipeline stages		*Execution stages*	
IF	Instruction fetch	ADDR	Pre modified address calculation
DQ	Dispatch queue	DPM	DP register modification
DE	Decode	MEM	Memory access
DP	DP register update	ALU	ALU operations
EX1	Execution phase 1	MAC	MAC unit operations
EX2	Execution phase 2	SCU	SCU operations

Figure 2.2: Pipeline diagram.

Table 2.1: Chip characteristics of the SPXK5.

Technology	0.13 µm, 5-level metal CMOS
Die size	1.6 mm x 1.6 mm
Transistor count	730 K transistors
Maximum operational frequency	250 MHz at 1.5 V, 180-200 MHz at 0.9 V
Average power consumption	0.15 mW/MIPS at 1.5 V, 0.05 mW/MIPS at 0.9 V

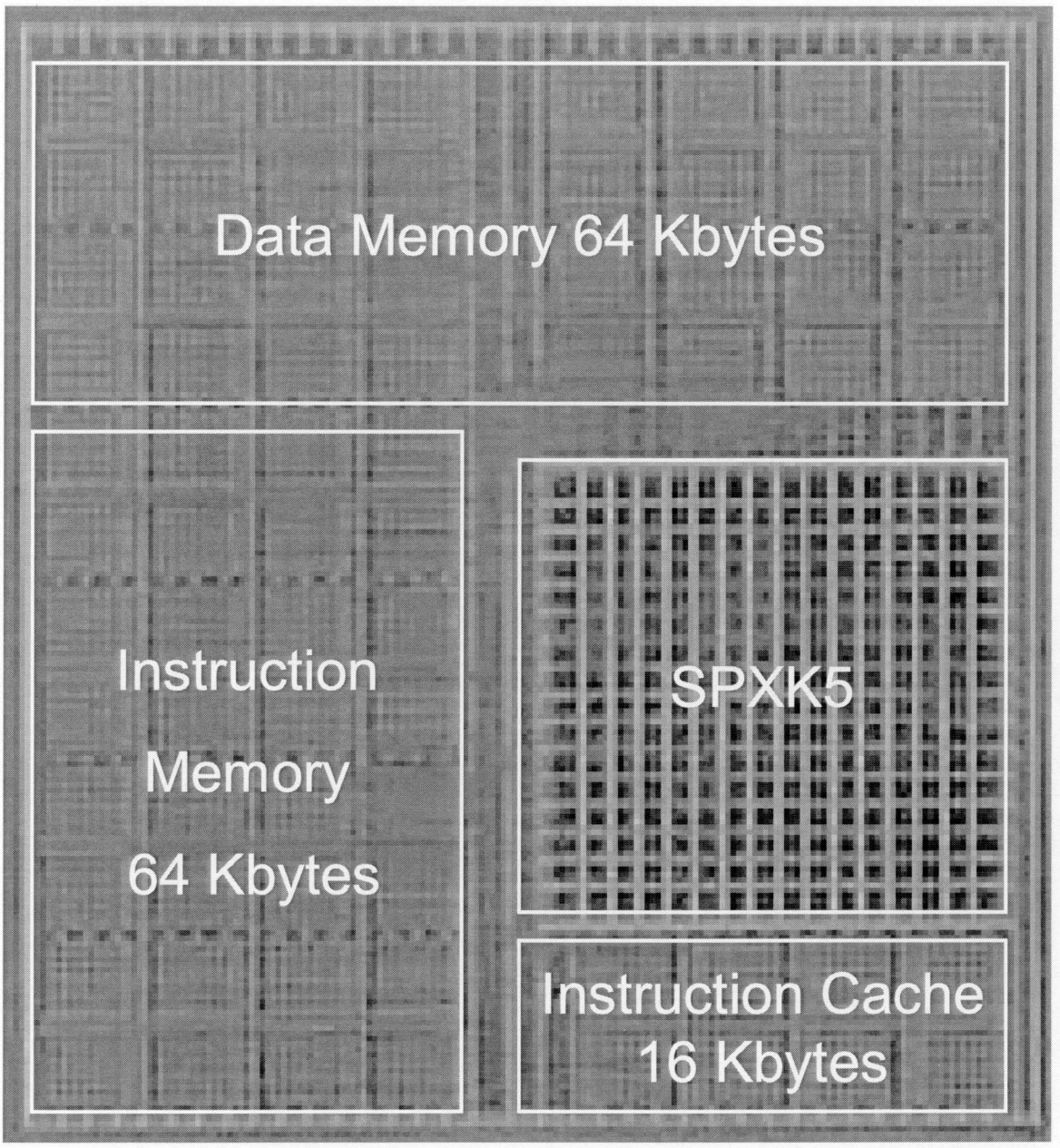


Figure 2.3: Chip micrograph of the SPXK5 and its memory banks.

compilers to generate more efficient programs. Moreover, a C-like algebraic assembly language has been developed to help programmers to develop highly parallelized programs easily.

Table 2.2 summarizes the SPXK5 instruction set. Each instruction is executed by its corresponding functional unit, e.g., general-purpose register addition is executed by either of the ALUs.

MAC instructions include 16- $\times$ 16-bit multiplications and 40(16)-bit $\pm$ 16- $\times$ 16-bit multiply-accumulate operations. All 16-bit $\pm$ 16- $\times$ 16-bit multiply-accumulate operations may involve rounding during accumulation. Each of the general-purpose registers can be used as an accumulator for these instructions. Either 16-bit portion of a general-purpose register can also be assigned for use as an accumulator or as a source operand.

ALU instructions include general arithmetic, shift, and logical operations. They also include such media instructions as parallel addition/subtraction, parallel absolute addition, parallel shift, format conversion between 8-bit and 16-bit, and parallel maximum/minimum operations.

DAU instructions include both load/store and such address register modifications as DP register addition/subtraction. Load/store operations on data stored in the local memory connected directly to the SPXK5 are executed in a single cycle.

SCU instructions include register transfers between general-purpose registers and other registers (e.g., address registers, offset registers, system registers), branch, subroutine call/return, zero overhead looping, conditional execution, and other program sequence control operations. Here, zero overhead loops can be nested up to four levels and will have three delay cycles through which users can issue instructions. Instructions fetched within the three cycles after a loop instruction are executed before the loop begins and are not part of the loop. The only instructions that cannot be issued in the delay cycles are branch, subroutine call/return, and a new loop instruction. Branch and subroutine call/return instructions involve 3 to 5 cycles of latency. Conditional execution can be applied to all ALU instructions and to register transfers between general-purpose registers and address registers.

2.3.1 Variable length instruction packet

All SPXK5 instructions are either 16 or 32 bits long. Instruction codes that include instructions to be executed in parallel are referred to as an instruction packet. The total length of instructions in an instruction packet can vary from 16 bits to 64 bits, i.e., they will be multiples of 16. This is in contrast to a conventional VLIW architecture, which requires that the length of each instruction packet be the same. Variable-length instruction packets allow the elimination of the redundant non-operations (e.g. NOPs) used to fill packets, thus producing higher code density. Each instruction code has a field to indicate its execution unit, code length, and parallel execution. Figure 2.4 shows instruction packets containing from one to four instructions.

2.3.2 Saturation mode

The SPXK5 has a saturation mode for all arithmetic operations (addition, subtraction, multiply-accumulate operation, etc.) in order to efficiently implement such media processing as speech CODEC, video CODEC, etc., and the saturation mode can be turned off/on for each destination register individually. Those arithmetic operations in the saturation mode are compliant with the speech CODEC standards of the European Telecommunications Standards Institute (ETSI). When

Table 2.2: Instruction set summary.

Instruction description	Syntax example
MAC unit operations	
Multiply	`R0=R1H*R2H`
Multiply-accumulate	`R0=R0+R1H*R2H`
Multiply-accumulate with rounding	`R0H=maddr(R1H,R2H)`
ALU operations	
Add or subtract two operands	`R0=R1+R2`
Absolute value	`R0=abs(R1)`
Various logical operations	`R0=R1 and R2`
Negate	`R0=-R1`
Clip and/or round	`R0=clip(R1)`
Clear an operand register	`clr(R0)`
Divide	`R0/=R1`
Exponent	`R0=exp(R1)`
Compare two operands	`R0=lt(R1,R2)`
Pack two values in a register	`R0=packv(R0)`
Parallel add/subtract	`R0=padd(R0,R1)`
Parallel maximum/minimum	`R0=pmax(R0,R1)`
Parallel shift	`R0=psra1(R0)`
Shift	`R0=R1 sra R2`
Sign/zero extension	`R0=signext(R0HL)`
Unpack two values in a register	`R0=unpackv(R0)`
DAU operations	
Add/subtract DP registers	`DP0=DP1+DP2`
Modify a DP register	`DP0=DP0+DN0`
Sign/zero extension	`DP0=signext(DP0)`
Load with post modification	`R0=*DP0++`
Load with pre modification	`R0=*(DP0+`*N*`)`
Direct addressing load	`R0=*(DBASE+`*N*`)`
Store with post modification	`*DP0++=R0H`
Store with pre modification	`*(DP0+`*N*`)=R0H`
Direct addressing store	`*(DBASE+`*N*`)=R0H`
SCU operations	
Branch	`Jmp label`
Subroutine call/return	`call label`
Zero overhead loop	`loop` *N*
Move	`DP0=R0HL`
Conditional execution	`if (R0==0)`
Kernel state control	`halt`

Instruction packet #1	`clr(R0)`	`clr(R1)`	`R2=*DP0`	`R3=*DP1`
#2	`R0=R0+R2`	`R1=R1+R3`	`R2=*DP2`	
#3	`R0=R0+R2`	`R1=R1+R2`		
#4	`R0=R0+R1`			

Figure 2.4: Instruction packets containing one to four instructions.

the saturation mode is turned on for the destination register of an arithmetic instruction, the result of the instruction will be saturated. This destination-register-dependent saturation mode has two benefits: it enables simultaneous execution of instructions both with and without the saturation mode, and it allows for a reduction in the number of instructions whose results are saturated, which leads to reduced instruction code length, i.e., most instructions can be expressed in 16 bits.

2.3.3 Special instructions for particular applications

The VLIW-based architecture of the SPXK5 accelerates applications by utilizing instruction-level parallelism. It also contains a number of SIMD instructions so as to take advantage of data-level parallelism [17] as well. In addition, a number of application-specific instructions are included in its instruction set for image/video processing and Viterbi decoding. These special instructions are executed on each of the two ALUs. Although this kind of enhancement could be implemented on a co-processor, the co-processor approach tends to be less flexible than the instruction-level approach, and it also tends to increase chip area. Here, rather, the logic blocks dedicated to these special instructions occupy only 2.5% of the DSP core, and their power dissipation can be reduced by completely stopping the switching activity of those logic blocks when they are not being used.

Eight special instructions were carefully selected from among the great variety of conceivable SIMD instructions so as to provide sufficient coverage without significantly increasing either the length of instruction encoding fields or the total code size. Since all special instructions share eight general-purpose registers with all the other instructions, it is easy for them to use the results of other instructions, and for other instructions to use theirs. The eight special instructions are briefly described below.

PADD and PSUB instructions perform two add operations or two subtract operations, respectively, in parallel. PADD instruction adds 16-bit values placed in the high portions of source and destination registers, and also adds 16-bit values placed in the low portions of source and destination registers at the same time. The same thing is done for PSUB instruction. These instructions are general SIMD instructions and are applicable to such applications as video encoding/decoding, Viterbi decoding, FFT, etc.

PSHIFT instruction shifts right two 16-bit values placed in the high and low portions of a destination register, and rounds each of the values. The amount of shift is specified in the instruction format. This instruction is also a general SIMD instruction and is useful for motion compensation

in video encoding/decoding and for scaling in FFT.

PADDABS instruction calculates the absolute values of two 16-bit values placed in the high and low portions of a source register and adds those absolute values to two 16-bit values placed in the high and low portions of a destination register. This instruction is useful for motion estimation in video encoding.

PACKV instruction packs two signed 16-bit values into two unsigned 8-bit values. The two 8-bit values are then packed and stored in the low portion of a destination register. If a signed 16-bit value is beyond the range of an unsigned 8-bit value (i.e., greater than 255 or less than 0), the saturated value (255 or 0, respectively) is stored. This instruction is effective for motion compensation in video encoding/decoding for storing calculated pixel values into memory.

UNPACKV instruction unpacks the two packed unsigned 8-bit values placed in the low portion of a destination register to the high and low portions of a destination register. This instruction is used in video encoding/decoding for reading pixel values stored in memory.

PMAX and PMIN instructions perform two maximum operations or two minimum operations, respectively, in parallel, and also perform specific bit-manipulation functions for both Viterbi decoding and maximum/minimum index search.

2.4 Application benchmarks

This section presents processing performance results for the SPXK5. Table 2.3 lists the benchmarks for the μPD77210 (the latest product employing the μPD7701x architecture), the SPXK5, Texas Instruments C55xTM [14, 18], and Intel/ADI Micro Signal Architecture (MSA) [15] for several basic applications: finite impulse response (FIR) filtering, infinite impulse response (IIR) filtering, least mean square (LMS) adaptive filtering, fast Fourier transformation (FFT), maximum value/index search, and Viterbi decoding. As shown in the table, the SPXK5 is at least twice as fast as the μPD77210 for all benchmarks other than IIR filtering. Since the SPXK5 has two MAC units, it can perform FIR filtering twice as fast as DSPs with only one MAC unit. Furthermore, the SPXK5 is highly optimized so as to execute more efficiently the basic applications compared does an off-the-shelf low-power DSP core such as C55x.

The following sections addresses three application examples: delayed least mean square adaptive filtering, Viterbi decoding, and MPEG-4 video encoding/decoding, all important applications for handheld terminals. Delayed least mean square adaptive filtering is applied in echo cancellation, channel equalization, etc.; Viterbi decoding is one of the most commonly used forward-error-correction techniques in wireless communication systems; MPEG-4 video encoding and decoding are necessary for videophones. These three examples illustrate how the 4-way VLIW architecture, dual MAC units, and the eight special instructions on the SPXK5 can be efficiently utilized.

2.4.1 Delayed least mean square adaptive filtering

The least mean square (LMS) algorithm employed in transversal filter structures is widely used in many digital signal processing applications, e.g., echo cancellation, channel equalization, etc. [19]. The input, output, and desired signals at time index i are defined as, respectively, x_i, y_i, and d_i, and

Table 2.3: General DSP function benchmark comparison.

Benchmarks	Cycle counts			
	μPD77210	SPXK5	TI C55x *1	Intel/ADI MSA *2
FIR filtering (*N* samples, *T* taps)	*NT*	*NT*/2	*NT*/2	*NT*/2
IIR filtering (*N* samples, *B* biquads)	5*NB*	3*NB*	3*NB*	2.5 *NB*
LMS adaptive filtering (*N* samples, *T* taps)	3*NT*	*NT*	2*NT*	1.5 *NT*
Maximum value search (*N* samples)	2*N*	*N*/4	*N*/2	*N*/2
Maximum index search (*N* samples)	3*N*	*N*/4	*N*/2	*N*/2
Radix-2 complex FFT butterfly	8	3	5	3
256-point complex FFT	9196	2944	4786	3176
ACS in Viterbi decoding	5	1	1	1
Viterbi decoding for GSM (*K*=5, *R*=1/2)	19478	6048	-	6357

*1: sources http://www.ti.com and [17]

*2: source [9]

the j-th filter coefficient as w_j. The LMS adaptive filtering can then be formulated as

$$y_i = \sum_{j=0}^{T-1} w_j \cdot x_{i-j}, \tag{2.1}$$

$$e_i = d_i - y_i, \tag{2.2}$$

$$w_j = w_j + 2\mu \cdot e_i \cdot x_{i-j}, \qquad j = 0, 1, 2, \ldots, T-1, \tag{2.3}$$

where μ is a small positive constant referred to as the step size, and T is the number of filter taps. In this algorithm, the estimated signal y_i in each data interval is calculated and subtracted from the desired signal d_i. The error is then used to update the coefficients before the next sample arrives.

For algorithmic or hardware architectural reasons, however, when updating coefficients in certain practical applications, there may be a delay in the LMS algorithm. Here, an LMS algorithm with such a delayed coefficient adaptation is dealt with as an example. This kind of LMS algorithm is referred to as a delayed LMS (DLMS) algorithm [20]. This DLMS algorithm gains an advantage in acceleration for both Equations (2.1) and (2.2) by using the previous error signal e_{i-1} in Equation (2.3) to update each coefficient, rather than using current error signal e_i. This allows both (2.1) and (2.3) easily to be combined into a single loop, which in turn makes it possible to share load operations in terms of both input signals and coefficients, and unified-loop implementation of (2.1) and (2.3) is faster than individual-loop implementation.

Implementation of the DLMS algorithm

Figure 2.5 shows a C program of the DLMS algorithm. In the innermost loop of the figure, coefficients are updated using previous error signals while outputs are also calculated. Here, it is assumed that all signals are fixed-point and that input signals `x[i-j]`, coefficients `w[j]`, and the error signal err are all expressed as 16-bit values in a Q15 format. Note that, in a Qn format, n bits are used to represent the two's complement fractional portion. Output signal `y[i]` is assumed to be 40-bit values in a Q31 format. Repetitive accumulations are required for the calculation of each of the outputs, and ideally, to avoid overflow at that time, each should be calculated within a 40-bit general-purpose register. Each coefficient, on the other hand, will be updated with only a single accumulation, after which the result will be written back into memory. When the coefficients are expressed as 16-bit values, this kind of MAC operation will not require a 40-bit accumulator because saturation and rounding in the accumulation allow sufficient accuracy to be maintained.

As previously noted, the SPXK5 has a useful saturation mode for arithmetic operations. In addition, its instruction set includes 16-bit MAC instructions with rounding. One of these instructions is referred to as the MADDR instruction. It occupies either the high or low 16-bit portion of a general-purpose register being used as an accumulator. Figure 2.6 shows two possible MADDR instruction patterns, in which each has a different 16-bit accumulator, either r4H or r4L, respectively, neither affecting the other. With this instruction, two coefficients placed in a general-purpose register can be updated individually. Thus, the number of registers for updating coefficients can be halved and optimizing techniques such as loop unrolling and software pipelining can be applied by utilizing the registers that have consequently become vacant. This instruction can be used effectively when an application does not require significant repetition of a MAC operation in its innermost loops, e.g., coefficient updating in LMS algorithms and butterfly operations in FFT.

```
1:   err=0;
2:   for ( i=0 ; i<N ; i++ ) {
4:       for ( j=0 ; j<T ; j++ ) {
5:           y[i] += w[j] * x[i-j];
6:           w[j] += err * x[i-j-1];
7:       }
8:       e[i] = d[i]-y[i];
3:       err = 2 * u * e[i];
9:   }
```

Figure 2.5: C program for the Delayed LMS algorithm.

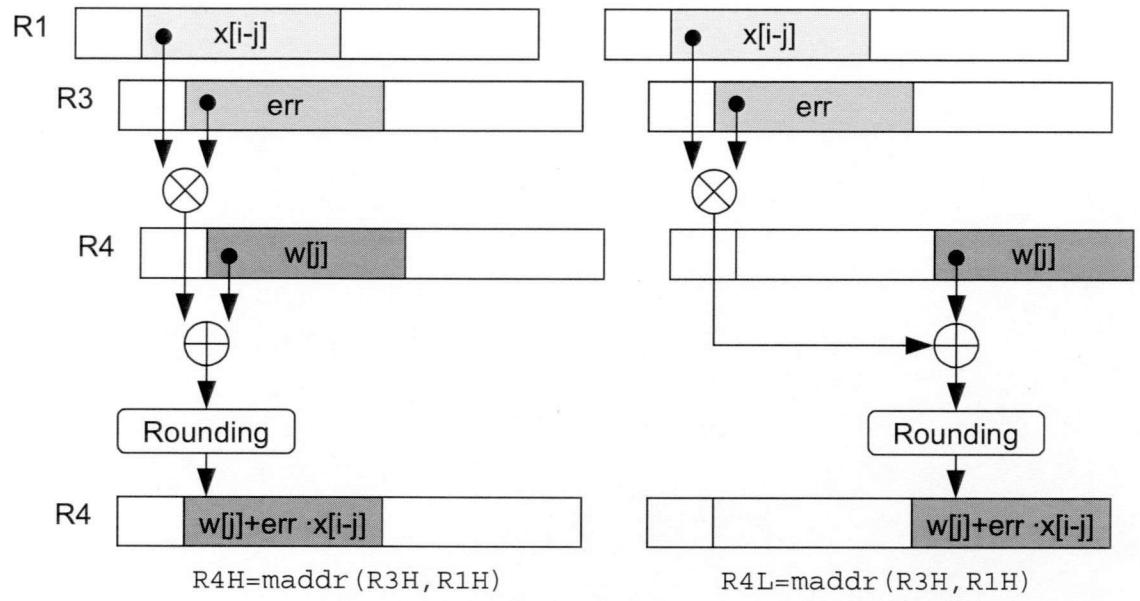


Figure 2.6: Two possible patterns of MADDR 16-bit MAC instruction.

Operations in the innermost loop

Figure 2.7 shows an assembly code segment for the innermost loop of the DLMS algorithm in Figure 2.5. Each of lines 1 through 4 and 6 through 11 represents an instruction packet. Lines 5 and 12 are loop brackets to indicate, respectively, the beginning of the loop and the end of the loop. Here, each packet contains one instruction. Three instruction packets after line 1 (i.e., lines 2 through 4) are not part of the loop and are executed before the loop begins. The innermost loop in the figure (i.e., lines 6 through 11) contains five actual operations, and line 10 contains a non-operation that provides the time required to wait for the preceding MAC result. The five operations include two MAC operations, two 16-bit load operations, and one 16-bit store operation. One of the MACs is an ordinary MAC instruction 40-bit + 16- × 16-bit (`R0+=R4L*R1H`), and the other is an MADDR instruction 16-bit + 16- × 16-bit (`R4L=maddr(R3H,R1H)`). The load operations in both the third and fifth lines use address register DP4 in a post-increment mode (`R4L=*DP4++`), and DP0 in a post-decrement mode (`R1H=*DP0--`). The store operation in the eighth line uses address register DP5 in a post-increment mode (`*DP5++=R4L`). If the load and store operations are reduced to two by utilizing 32-bit load/store instructions, the number of operations in the innermost loop will become four, so that through the optimization techniques, these operations can be executed at 1 cycle/tap. Note that the SPXK5 can execute up to four instructions during the same clock cycle.

Optimization of the DLMS algorithm

In order to accelerate the innermost loop, 32-bit load/store instructions were used, rather than 16-bit load/store instructions, to read and write coefficients. In this case, each 32-bit load instruction reads

```
1:    loop T                      ;
2:        R1H=*DP0--              ; load x[i-1]
3:        NOP                     ;
4:        NOP                     ;
5:    {
6:        R4L=*DP4++              ; load w[j]
7:        R0+=R4L*R1H             ; y[i]+=w[j]*x[i-j]
8:        R1H=*DP0--              ; load x[i-j-1]
9:        R4L=maddr(R3H,R1H)      ; w[j]+=err*x[i-j-1]
10:       NOP                     ; wait for w[j]
11:       *DP5++=R4L              ; store w[j]
12:   }
```

Figure 2.7: Innermost loop code segment of an assembly program corresponding to Figure 2.5.

two adjacent coefficients, `w[j]` and `w[j+1]`, into a general-purpose register and each 32-bit store instruction writes back into the memory two coefficients that had been placed in a register. As a result, the total number of 32-bit load/store instructions in terms of coefficients in the innermost loop can be reduced by half, and the number of load/store instructions per tap can be reduced to two.

Figure 2.8 shows the unrolled innermost loop of Figure 2.5. In Figure 2.8 the first (leftmost) of the four columns represents the calculation of output `y[i]`, the second the updates of coefficients, the third load/store operations for coefficients `w[j]` and `w[j+1]`, and the fourth load operations for inputs `x[i-j]`. Here it is assumed that a 32-bit load operation is used to read two coefficients at a time, and that, similarly, a 32-bit store operation is also used to write two coefficients at a time back into the memory. In the third column, 32-bit load and store operations appear alternately, which allows each clock cycle to be filled with four operations.

Figure 2.9 shows the assembly code segment corresponding to the unrolled operations in Figure 2.8, minus the prologue and epilogue. Here, the delay cycles of the loop instruction are ignored and the focus for analysis is set on the inside of the loop. Each of the four columns corresponds to a column in Figure 2.8. Note that in the third column, the 32-bit load and store instructions in the post-increment mode are denoted as, respectively, Rn=*DPm.d++ and *DPm.d++=Rn, where m and n are arbitrary integers between 0 through 7. In the figure, sixteen multiply-accumulate operations, for updating eight coefficients and for calculating the output `y[i]` on eight taps, are simultaneously executed in the loop brackets, i.e., `loop T/8 {...}`. The inside of the loop is executed T/8 times. While the output is calculated within general-purpose register R0, eight 16-bit portions of four general-purpose registers, R4-R7, are used as 16-bit accumulators to update

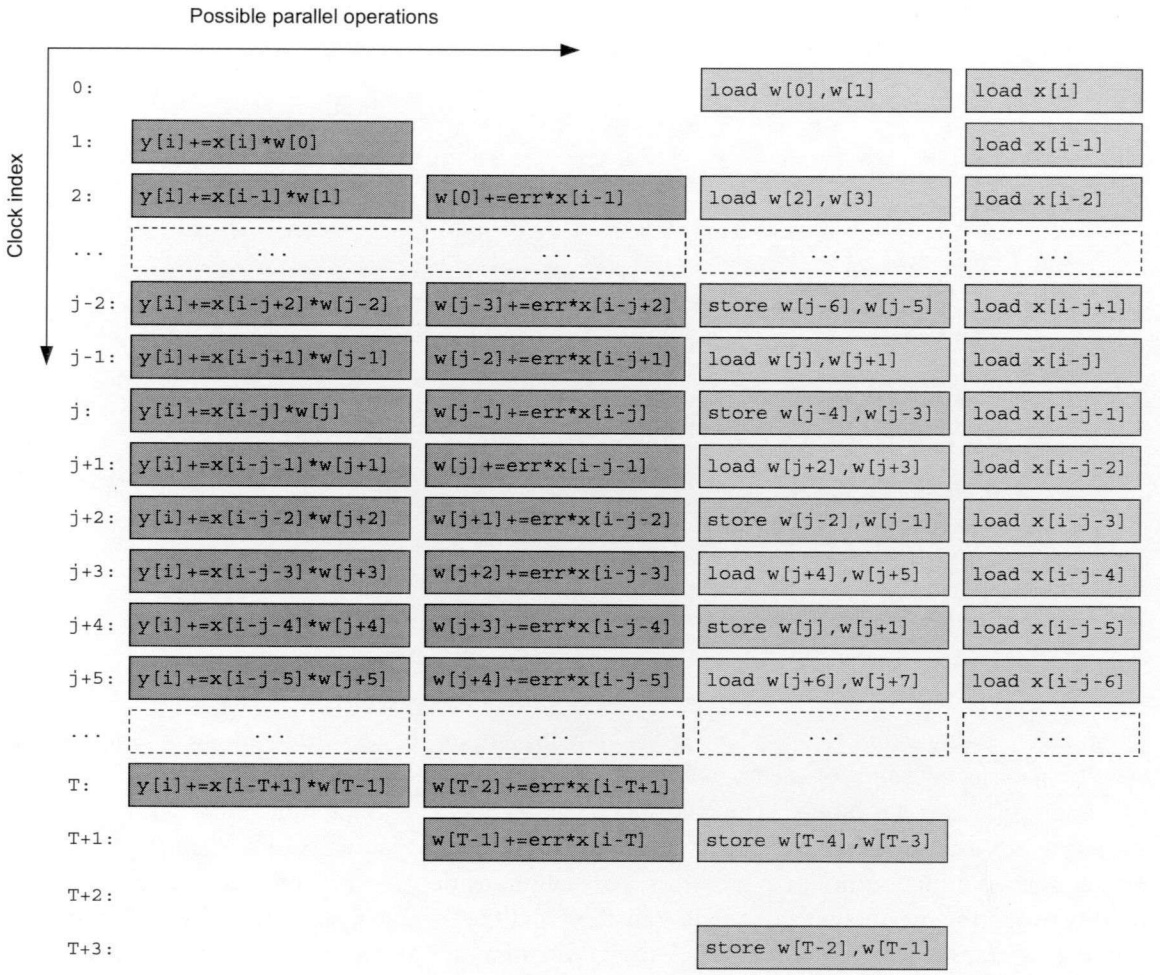


Figure 2.8: Unrolled innermost loop of Figure 2.5.

eight coefficients. This code scheduling is possible because of a highly orthogonal instruction set, single-cycle throughput MAC operations, single-cycle load/store operations, and efficient register handling. Consequently, the DLMS algorithm can be implemented on the SPXK5 at 1 cycle/tap. Since the same strategy of using a MADDR instruction can also be applied to the innermost loop of regular LMS algorithms (non-delayed), non-delayed LMS algorithms can be implemented on the SPXK5 at 1 cycle/tap.

2.4.2 Viterbi decoding

The Viterbi decoding algorithm is a maximum-likelihood decoding algorithm for convolutional codes and is commonly used in forward error correction because it is simple to implement and

```
1:   loop T/8;
2:   {
3:      R0+=R5L*R1H,  R4H=maddr(R3H,R1H),  *DP5.d++=R7,  R1H=*DP0--;
4:      R0+=R5H*R1H,  R5L=maddr(R3H,R1H),  R6=*DP4.d++,  R1H=*DP0--;
5:      R0+=R6L*R1H,  R5H=maddr(R3H,R1H),  *DP5.d++=R4,  R1H=*DP0--;
6:      R0+=R6H*R1H,  R6L=maddr(R3H,R1H),  R7=*DP4.d++,  R1H=*DP0--;
7:      R0+=R7L*R1H,  R6H=maddr(R3H,R1H),  *DP5.d++=R5,  R1H=*DP0--;
8:      R0+=R7H*R1H,  R7L=maddr(R3H,R1H),  R4=*DP4.d++,  R1H=*DP0--;
9:      R0+=R4L*R1H,  R7H=maddr(R3H,R1H),  *DP5.d++=R6,  R1H=*DP0--;
10:     R0+=R4H*R1H,  R4L=maddr(R3H,R1H),  R5=*DP4.d++,  R1H=*DP0--;
11:  }
```

Figure 2.9: Accelerated assembly code segment for the DLMS algorithm.

offers large coding gain [21]. To decode the encoder output, the algorithm attempts to recreate all state transitions possible for the encoder and to choose the most likely set of state transitions for use as original encoder inputs. The path metric of each state and the branch metric of each state transition are used as criteria for this selection. A path metric is the likelihood ratio between the original encoder inputs and the recreated set of state transitions. A branch metric is the likelihood ratio between the original encoder state transition and the recreated state transition. The structure of the state transitions is referred to as a trellis diagram, and the trellis diagram for an $R = 1/n$ convolutional code, where R is a coding rate and n is an integer larger than 1, can be subdivided into a number of basic modules because of the inherent symmetry of the trellis structure. These basic modules can be represented as state transitions between two old states $(m, m+M/2)$ in stage k and two new states $(2m, 2m+1)$ in stage $k+1$, where $m = 0, 1, ..., M/2-1$ shown in Figure 2.10.

A path metric at state m in stage k is defined as $p_{m,k}$, and also define a branch metric as bm. The two path metrics in stage $k+1$, $p_{2m,k+1}$ and $p_{2m+1,k+1}$ may be expressed as

$$p_{2m,k+1} = \max[p_{m,k} + bm, p_{m+M/2,k} - bm], \tag{2.4}$$

$$p_{2m+1,k+1} = \max[p_{m,k} - bm, p_{m+M/2,k} + bm], \tag{2.5}$$

where $\max[x, y]$ represents a larger value of either x or y. The operations represented by Equations (2.4) and (2.5) are referred to as add-compare-select (ACS) operations. The speed of ACS operations is crucial to the successful implementation of Viterbi decoding on DSPs because the operations are repeated millions of times to recreate all state transitions.

With respect to ACS operations, the SPXK5 has three useful SIMD instructions for calculating path metrics efficiently, PADD, PSUB, and PMAX. Figure 2.11 illustrates how to use these

SIMD instructions to perform two ACS operations. Here, both path metrics and branch metrics are assumed to be signed 16-bit integers. First, PADD and PSUB calculate two candidate values for one of two path metrics at stage $k+1$ in (2.4) and (2.5), respectively. Specifically, the first two candidates are $p_{m,k} + bm$ and $p_{m+M/2,k} - bm$ for $p_{2m,k+1}$, while the second two candidates are $p_{m,k} - bm$ and $p_{m+M/2,k} + bm$ for $p_{2m+1,k+1}$. The first two candidates are placed into the high and low 16-bit portions, respectively, of register R0; the same is done for the second two candidates with respect to register R1. Next, PMAX compares the two candidates in each register and selects the largest from each. As a result, $p_{2m+1,k+1}$ and $p_{2m,k+1}$ are obtained and both of them are placed in register R1. PMAX also stores two 1-bit selection flags in a 32-bit shift register, i.e., a Viterbi history register (VHR), in order to indicate which candidate is larger in each of the two comparisons. After calculating path metrics at all stages, these flags are used to determine the maximum-likelihood surviving path.

Figure 2.12 shows an assembly code segment for four ACS operations, represented in four lines. Before execution, in both the first and third lines, registers R0 and R2 have branch metrics, and registers R1 and R3 have path metrics. The SPXK5 can execute the PADD, PSUB and PMAX instructions in two cycles to perform two ACS operations, i.e., 1 ACS/cycle.

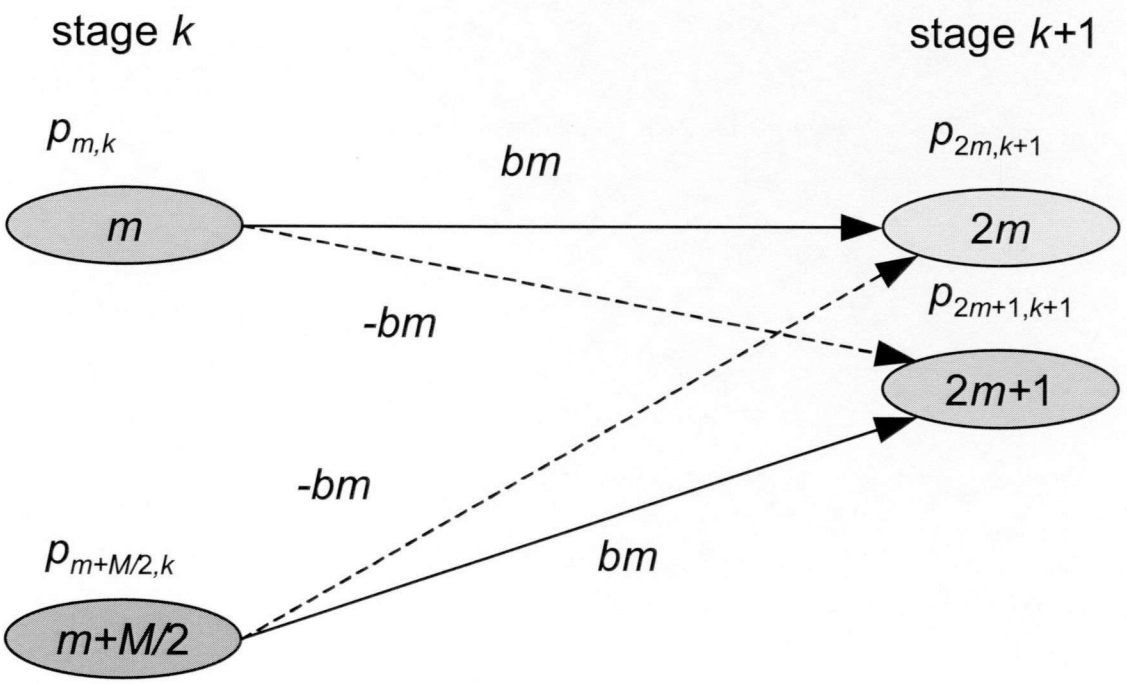


Figure 2.10: Add-compare-select operations.

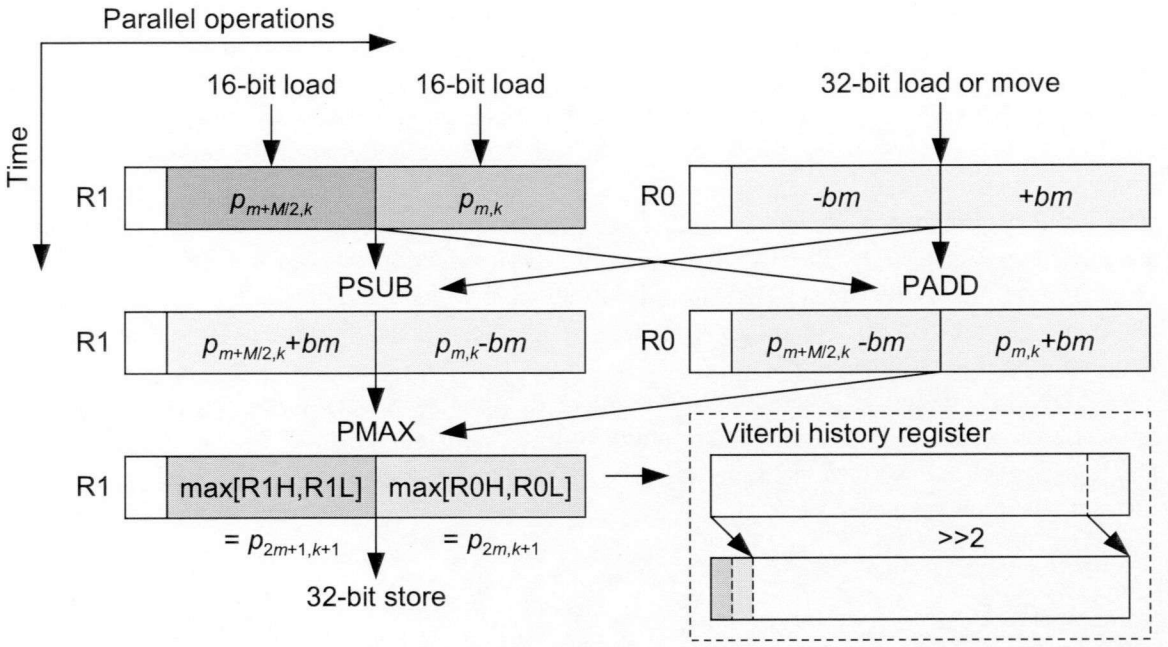


Figure 2.11: ACS operations on the SPXK5.

```
1:  R0=padd(R1),  R1=psub(R0),  R3L=*DP0++,  *DP4.d++=R3;  store pm(2m,k+1) pm(2m+1,k+1)
2:  R1=pmax(R0),  R2=R4,        R3H=*DP1++                ;
3:  R2=padd(R3),  R3=psub(r2),  R1L=*DP0++,  *DP4.d++=R1;  store pm(2m+2,k+1) pm(2m+3,k+1)
4:  R2=pmax(R3),  R0=R5,        R1H=*DP1++                ;
```

Figure 2.12: Assembly code segment for four ACS operations.

2.4.3 MPEG-4 video encoding and decoding

Here it is shown how media instructions are effective for the core functions of a video encoder/decoder algorithm. In previous work, the H.263 video CODEC (176x144 pixels, 7.5 frames/sec) have been already implemented on a μPD7701x architecture at 50 MHz in 1998 [22] and the MPEG-4 video CODEC SP@L1 (176x144 pixels, 15 frames/sec) on the μPD77210 at 47.1 MHz in 2001 [23]. The target for the SPXK5 implementation was the MPEG-4 video CODEC SP@L2 (352x288 pixels, 15 frames/sec), which is expected to be a key application in next generation mobile terminals. The main components in the MPEG-4 video CODEC are motion estimation (ME), discrete cosine transform (DCT), inverse DCT (IDCT), motion compensation (MC), quantization (Q), inverse quantization (IQ), variable length code coding (VLC), and variable length code decoding (VLD). Among them, SIMD instructions are effective for both ME and MC.

Motion estimation (ME)

ME is one of the most heavily demanding operations in video encoders. It uses block-matching techniques to search a motion vector for a desired macroblock. In a block-matching algorithm, both the mean absolute error (MAE) and the mean square error (MSE) between a current macroblock and a reference macroblock are feasible as block-matching criteria. The MAE and the MSE are defined as:

$$\mathrm{MAE} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |a_{m,n} - b_{m,n}|, \tag{2.6}$$

$$\mathrm{MSE} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (a_{m,n} - b_{m,n})^2, \tag{2.7}$$

where $a_{m,n}$ and $b_{m,n}$ are pixel values in, respectively, current and reference macroblocks. In the video CODEC implementation on the μPD77210, MAC instructions were used to calculate the MSE values that would serve as block matching criteria. The MSE for 1 pixel could be calculated in 2 cycles on the single MAC architecture. By way of contrast, for the SPXK5 implementation, MAE values were used for the criteria and they were calculated with two SIMD instructions, PSUB and PADDABS. The MAE for 1 pixel could be calculated in 0.5 cycles on the SPXK5. Figure 2.13 shows an MAE calculation flow diagram. A two-step search algorithm were used, which was similar to that used in [22]. Overall, ME in the SPXK5 architecture is 2.3 times faster than in the μPD77210.

Motion compensation (MC)

One of the core functions in MC is 4-pixel interpolation, in which averages of every 4 pixel values are calculated. By using three of the eight special instructions, PADD, PSHIFT, and UNPACKV, an average can be calculated in 1.5 cycles. PACKV instruction can also be used effectively to enable MC to store those averages into memory. Overall, MC is 1.8 times faster in the SPXK5 architecture than in the μPD77210.

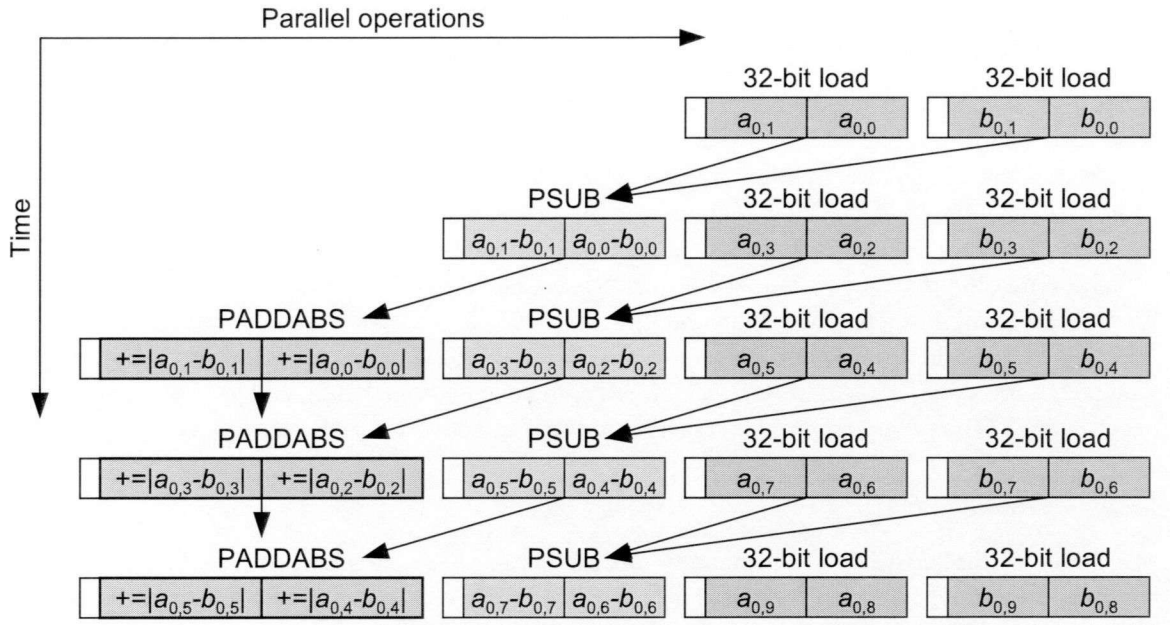


Figure 2.13: MAE calculation flow diagram for the SPXK5.

Discrete cosine transform (DCT)

In a previous study, a fast 8-point DCT algorithm was introduced, which required 23 MAC operations and 12 additions [16]. This algorithm was implemented in 35 cycles on the μPD77210. On the SPXK5, the same algorithm has been able to be implemented in only 15 cycles, using PADD and PSUB instructions in combination with two MAC units and two ALUs. The DCT in the SPXK5 architecture is 2.3 times faster than it was in the μPD77210.

Performance of MPEG-4 video CODEC on the SPXK5

On the SPXK5, MPEG-4 video CODEC SP@L2 (352x288 pixels, 15 frames/sec) has been implemented. Both encoding and decoding can typically be conducted at less than 105 MHz, with a power consumption of 21 mW at 0.9 volts. Figure 2.14 shows a cycle-count comparison for a typical case of MPEG-4 video encoding and decoding SP@L2 on both the SPXK5 architecture and the μPD77210. In order to focus on the difference between the core architectures, the respective overheads for external memory accesses have been excluded. As may be seen, the SPXK5 architecture is 1.8 times faster.

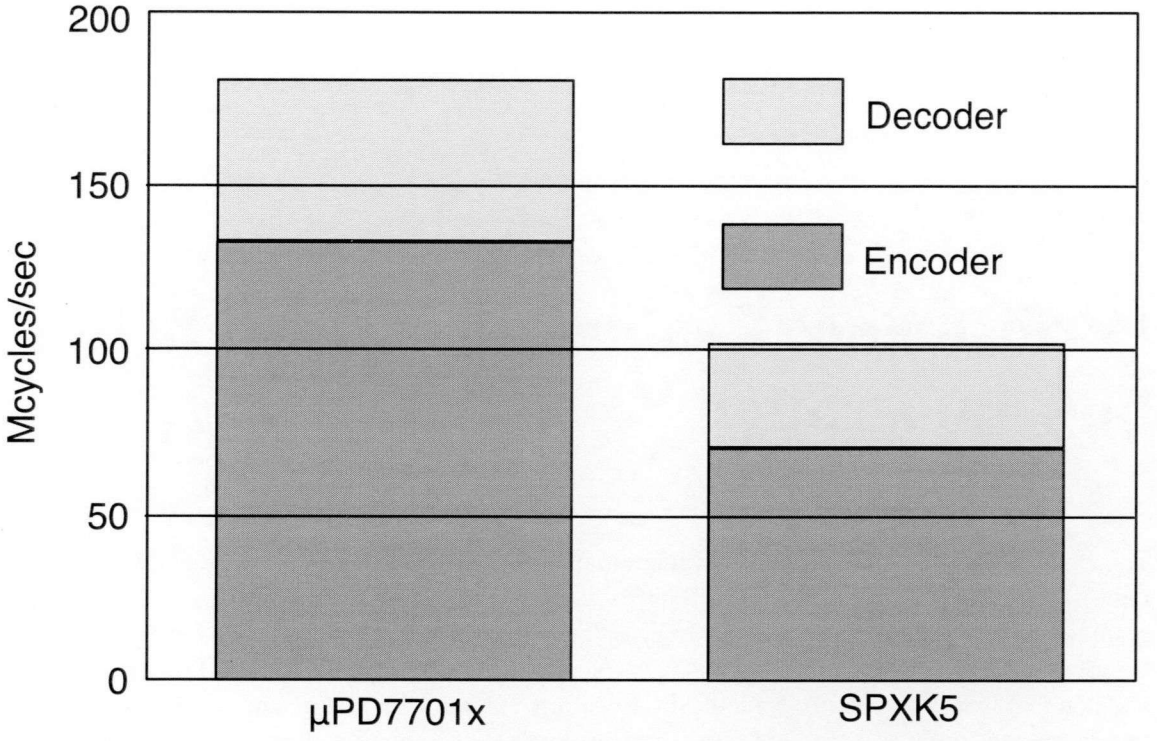


Figure 2.14: MPEG-4 encoder/decoder cycle counts (SP@L2, 352x288 pixels, 15 frames/sec).

2.5 Summary

A new-generation, general-purpose DSP core, the SPXK5 has been developed. It is a 4-way VLIW DSP core designed for multimedia applications on handheld terminals. Many low-power consumption features of the VLIW-based architecture make it possible for the SPXK5 to achieve both high performance and low-power consumption. These features include compact register files, fewer restrictions in terms of parallel instructions, single-cycle operations for most instructions, software-based instruction scheduling, and an instruction set based on 16-bit encodings, etc. In addition, a set of eight special purpose instructions makes it possible to implement multimedia applications efficiently on the SPXK5 with no significant increase of its chip area.

Chapter 3

Performance Evaluation Method Before SoC Integration

This chapter provides a performance evaluation of audio and video CODECs by using a method for rapidly verifying and evaluating overall performance on real-time workloads of system LSIs integrated with SPXK5SC DSP cores. The SPXK5SC have been developed as a DSP core well-suited to system LSIs. Despite the fact that it is very important to evaluate the overall performance of target LSIs on real workloads before actual LSI fabrication, software simulators are too slow to deal with real workloads and full hardware prototyping is unable to respond well to design improvements. Therefore, a hardware emulation system has been developed to be used for system LSIs integrated with a SPXK5SC DSP core in order to evaluate the overall performance of audio/video CODEC on a target system. The emulation system using a DSP core TEG, which has a bus interface, and an FPGA is suitable for overall system evaluation on real-time workloads as well as architectural investigation. This chapter discuss the use of the emulation system in evaluating performance during AV CODEC execution. In addition, an architecture design based on the emulation system is also described.

3.1 System performance evaluation for LSIs using processors

As LSI designs have grown larger, the biggest development bottleneck has become LSI operational verification at the design level. On the system LSIs integrated with a large number of hardware components, the estimation of overall performance on real workloads becomes difficult. System LSIs now include different intellectual properties (IPs) such as CPUs and DSPs, and those IPs are connected to a backbone bus on the system LSIs. As the number of masters on the bus increases, since different kinds of data are transferred through the bus, the estimation of traffic on the bus becomes difficult. Access conflicts among bus masters could degrade the CPUs performance because the conflicts would disturb cache access of CPUs.

It is very important to evaluate the overall performance of target LSIs on real workloads before actual LSI fabrication. However, software simulators, while relatively inexpensive, versatile, and accurate, are too slow to deal with real-time workloads, e.g., audio/video signals, and full hardware prototyping is unable to respond well to design improvements. In this regard, as an intermediate

approach between software simulation and full hardware prototyping, hardware emulation using FPGAs appears promising [24].

This chapter provides a method for rapidly verifying and evaluating overall performance on real-time workloads of system LSIs integrated with SPXK5SC DSP cores. The SPXK5SC is a DSP core well-suited to system LSIs, and is composed of the SPXK5 core [25], a 64 Kbyte instruction memory, a 64 Kbyte data memory, a 16 Kbyte instruction cache, and an AMBA AHB interface. Its high performance and low-power consumption make it suitable for use in mobile terminals.

Here, an emulation system has been developed using an FPGA and an SPXK5SC core TEG (test element group), which has a system bus interface for it to be connected to the system backbone bus implemented on the FPGA. The FPGA is used to emulate the system bus and peripherals of target LSIs, making it possible to verify and evaluate the operations of hardware blocks in real situations and to develop improved designs.

Since the DSP core should be implemented as a discrete chip on the emulation system, the developed method is far faster than software simulators and can handle real-time audio/video signals. In addition, since new hardware components can be added to the FPGA to reflect recent additions to hardware designs, the developed method is able to respond well to design improvements. While the system bus and the peripherals implemented on the FPGA are operated at slower frequencies than those of target LSIs (roughly 1/3), since the frequency ratios between them can be made the same, the emulator essentially becomes a scaled-down version of the target LSI, enabling estimation of the overall performance before actual LSI fabrication. The method using a DSP core TEG and FPGA also makes it possible that architectures of system LSIs are investigated in real situations.

AV CODEC software for the SPXK5SC has been developed on the emulation system, and its overall performance (e.g., number of cycles, bus traffic, etc.) has been evaluated on real-time audio/video signals. Based on the evaluation results obtained with the emulation system, a new DSP integrated with the SPXK5SC core has been designed and fabricated. This chapter discusses the evaluation and architecture design based on the emulation system.

3.2 SPXK5 and SPXK5SC

The SPXK5 is a 16-bit general-purpose DSP core based on a VLIW architecture shown in Figure 3.1. It contains register files, buses, control blocks, and seven functional units. These functional units consist of two multiply-accumulate (MAC) units for 16-bit by 16-bit multiplication and 40-bit accumulation; two arithmetic logical units (ALU) for addition and subtraction, shift, and logical operations; two data address units (DAU) for load and store; and one system control unit (SCU) for branch, zero overhead looping, and conditional execution.

The SPXK5 instructions are either 16- or 32-bit wide and can be grouped into up to 64-bit instruction packets to be executed within a cycle. Users are free to choose instructions for parallel issue so long as the number and total length of those instructions fall within these limits. In addition, all arithmetical, logical, and store operations other than MAC operations are single-cycle operations. These fewer restrictions on parallel instructions and single-cycle operations are very important features in the SPXK5 instruction set architecture because they allow users and compilers to generate more efficient programs. Moreover, a C-like algebraic assembly language has been developed, which helps programmers to create highly parallelized programs more easily.

A number of application-specific instructions are included in the SPXK5 instruction set for image/video processing and Viterbi decoding. These instructions are parallel operations for two 16-bit data items in registers (e.g., parallel additions, parallel subtractions, parallel shifts, parallel absolute additions, and parallel maximum/minimum operations). Application-specific instructions are executed on each of the two ALUs.

Outside the SPXK5, instructions and data are stored separately (i.e., in different banks) to make sure that the SPXK5 can concurrently access both of them. Memory banks for instruction storage are connected to a single 64-bit instruction bus, while memory banks for data storage are connected to both of two 32-bit data buses . With three buses, the SPXK5 is able to access three different memory banks at the same time.

In developing the SPXK5SC core, a 64 Kbyte instruction memory, a 64 Kbyte data memory, a 16 Kbyte instruction cache, and an AMBA AHB interface have been added to the SPXK5 shown in Figure 3.2. The instruction and data memory are sub-divided into 4 Kbyte memory banks. Memory banks accessed by the SPXK5 are activated, and the others are suspended, in order to reduce power consumption. With the SPXK5SC, instructions may reside in the internal 64 Kbyte instruction memory or in an external memory. When instructions reside in an external memory, the instruction cache buffers the most recent instructions accessed by the SPXK5SC. The instruction cache is a direct-map cache and holds up to 16 Kbytes, 128 lines, and 128 bytes per line. Users can improve overall system performance by allocating instructions and data frequently used by the SPXK5SC into the SPXK5SC internal memory. A 32-bit cycle counter is equipped with the SPXK5SC. It counts the number of cycles and can be used for purposes of code profiling.

For a DSP core to be used in a system LSI, it needs a high-throughput interface to exchange information between hardware blocks in the system LSI. Although some conventional system LSIs use shared memory for that purpose, most recent system LSIs prefer the flexibility offered by a system backbone bus, which makes the connection of peripherals much easier. In order to allow the SPXK5SC to access the components in an LSI equipped with a backbone bus, an AMBA AHB interface is employed. The AMBA AHB on-chip bus specification is used for high-performance system modules [26]. The interface also enables other components to access memory in the SPXK5SC, and it provides a seamless interface to system LSIs that integrate the DSP cores, CPUs, and peripherals.

3.3 AV CODEC prototype system

The AV CODEC emulation system consists of an SPXK5SC evaluation board, an AV interface board, a laptop PC for control, a monitor, and a camera shown in Figures 3.3 and 3.4. The SPXK5SC evaluation board contains an SPXK5SC TEG, an FPGA, and 3-Mbyte SRAM. The operating frequency of the SPXK5SC TEG can be set to be a multiple of the system bus frequency, up to 200 MHz. The FPGA is used to emulate an AMBA AHB system bus and such peripherals for the SPXK5SC as an EBIF (external bus interface), a DMAC (direct memory access controller), an ASIO (audio serial input/output), and a VIF (video interface). The SPXK5SC TEG, the peripherals, and 3-Mbyte SRAM are connected to an AHB.

The AV interface board includes AD/DA converters for real-time video and speech signals. These converters are controlled by an FPGA in the AV interface board. The FPGA outputs digital

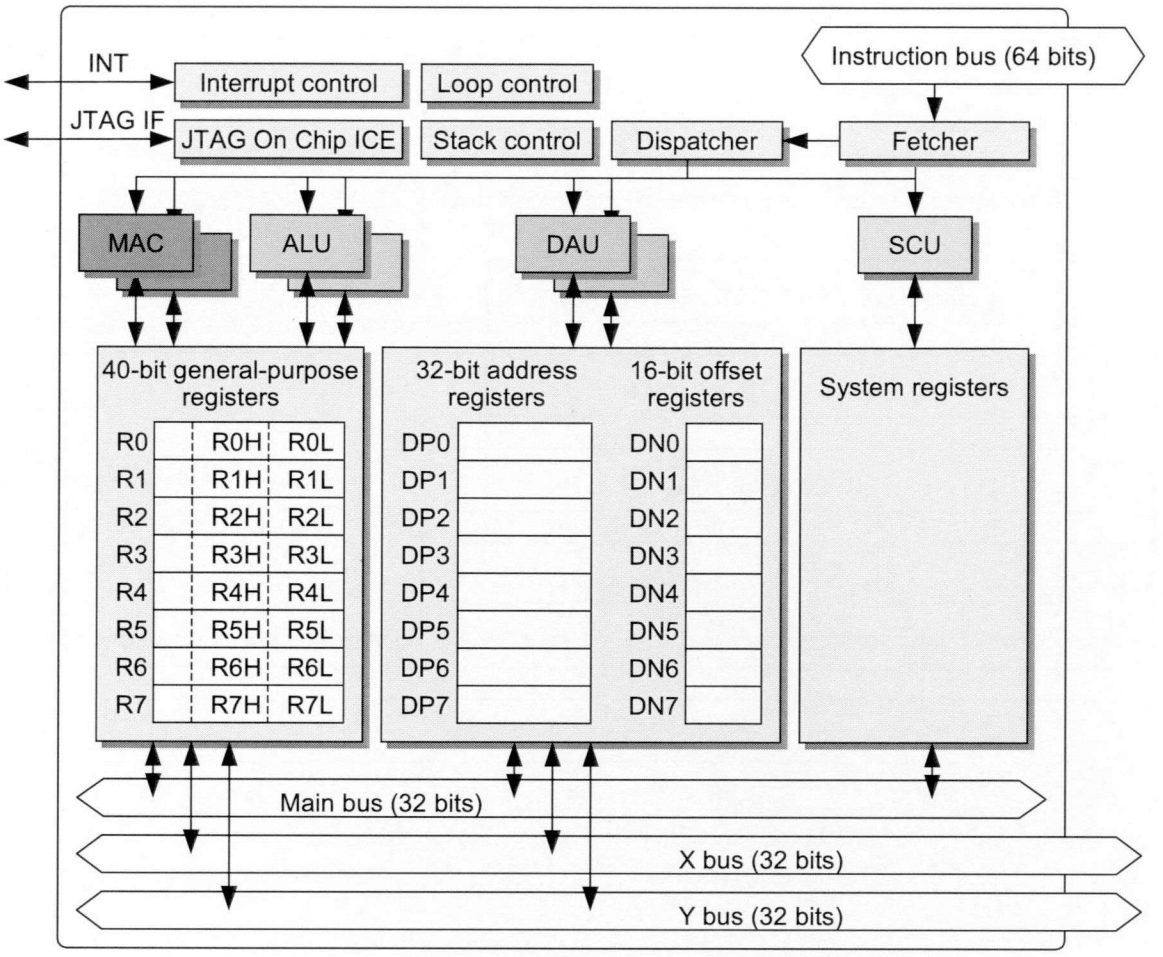


Figure 3.1: Block diagram of the SPXK5.

video signals to the VIF, and speech signals to the ASIO. The VIF captures video signals single-frame by single-frame at up to 30 frames/sec. The captured images are stored in the memory of the evaluation board. The VIF also output images stored in the memory to the AV interface board. The ASIO receives and outputs speech signals.

The DMAC transfers data between regions in the SPXK5SC memory map without intervention by the SPXK5SC itself. It allows movement of data (to and from peripherals, as well as to and from the internal memory of the SPXK5SC) to occur in the background of DSP operations. The DMAC has eight independent programmable channels, including two dedicated channels to transfer data representing rectangular regions of images.

In order to monitor the system bus, a bus traffic monitor is implemented in the FPGA. It counts the number of active cycles for each bus master in the AHB. Bus traffic is a major concern in designing systems, and the monitor provides an effective way to measure such traffic.

Minimizing memory access latency is crucial in maximizing overall LSI performance. If the operating frequency of the SPXK5SC were much faster than the AHB frequency, the SPXK5SC's

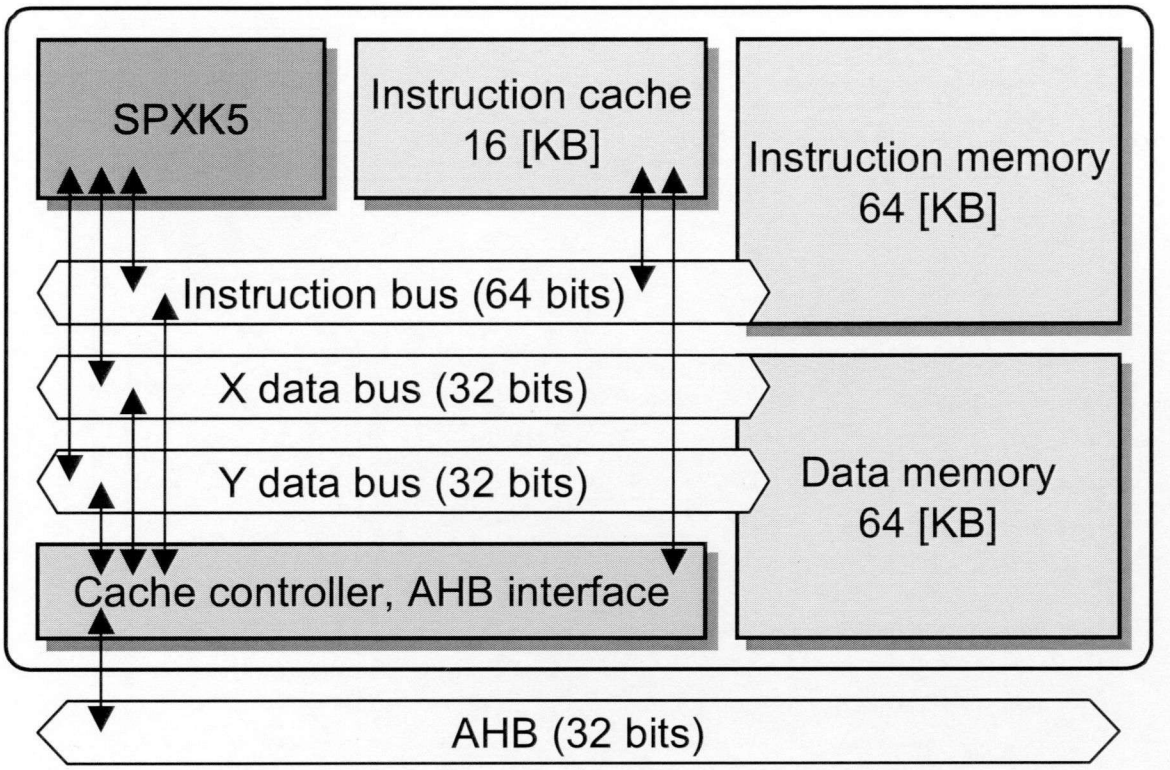


Figure 3.2: Block diagram of the SPXK5SC.

memory access to SRAM1 and SRAM2 (which are connected to the AHB) would create excessive access latency. The frequency ratio between the SPXK5SC and the AHB needs to be low if external memory access latency is to be kept low. In this work, the maximum frequencies of the SPXK5SC and AHB on target LSIs are 250 MHz and 100 MHz, respectively. The SPXK5SC frequency must be a multiple of the AHB frequency, here from 2:1 to 4:1; a frequency ratio 3:1 (i.e., a 250-MHz SPXK5 and a 83-MHz AHB,) is expected to provide the best performance. Since, for most accurate results, the AHB frequency in the emulation system should be kept at its maximum value (25 MHz), adjustments in frequency are made by changing the SPXK5SC frequency. In terms of an SPXK5SC's memory access latency, a 75-MHz SPXK5SC and a 25-MHz AHB on the emulation system are virtually equivalent to the combination of a 250-MHz SPXK5SC and a 83-MHz AHB on a target LSI.

3.4 Implementation of AV CODEC

MPEG-4 video CODEC [7, 8, 27] and AMR speech CODEC have been implemented on the emulation system shown in Table 3.1. The two work in a loop-back manner. That is, the video and speech CODEC follow the following procedure: (a) speech and video data from the AV interface

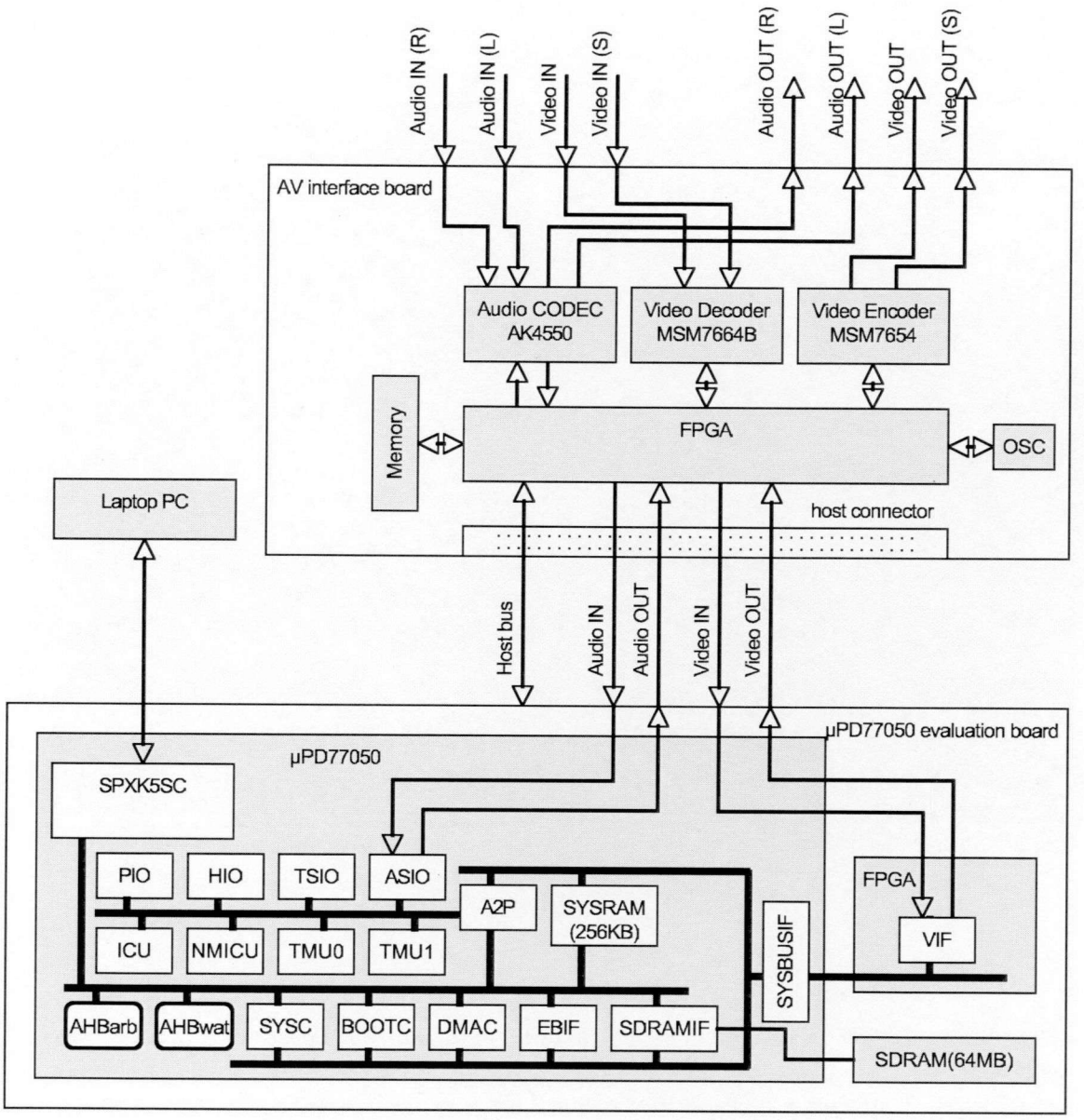


Figure 3.3: Block diagram of the AV CODEC prototype system.

board are encoded by the SPXK5SC, and the encoded stream is stored in an SRAM; (b) the stream is decoded by the SPXK5SC, and the decoded speech and video data are output through the AV interface board. These two procedures are continuously performed in real time without buffering any video/speech signals for emulation.

The size of all instructions and data for the AV CODEC is so large that it cannot fit into the internal memory of the SPXK5SC, and these instructions and data are placed separately in four

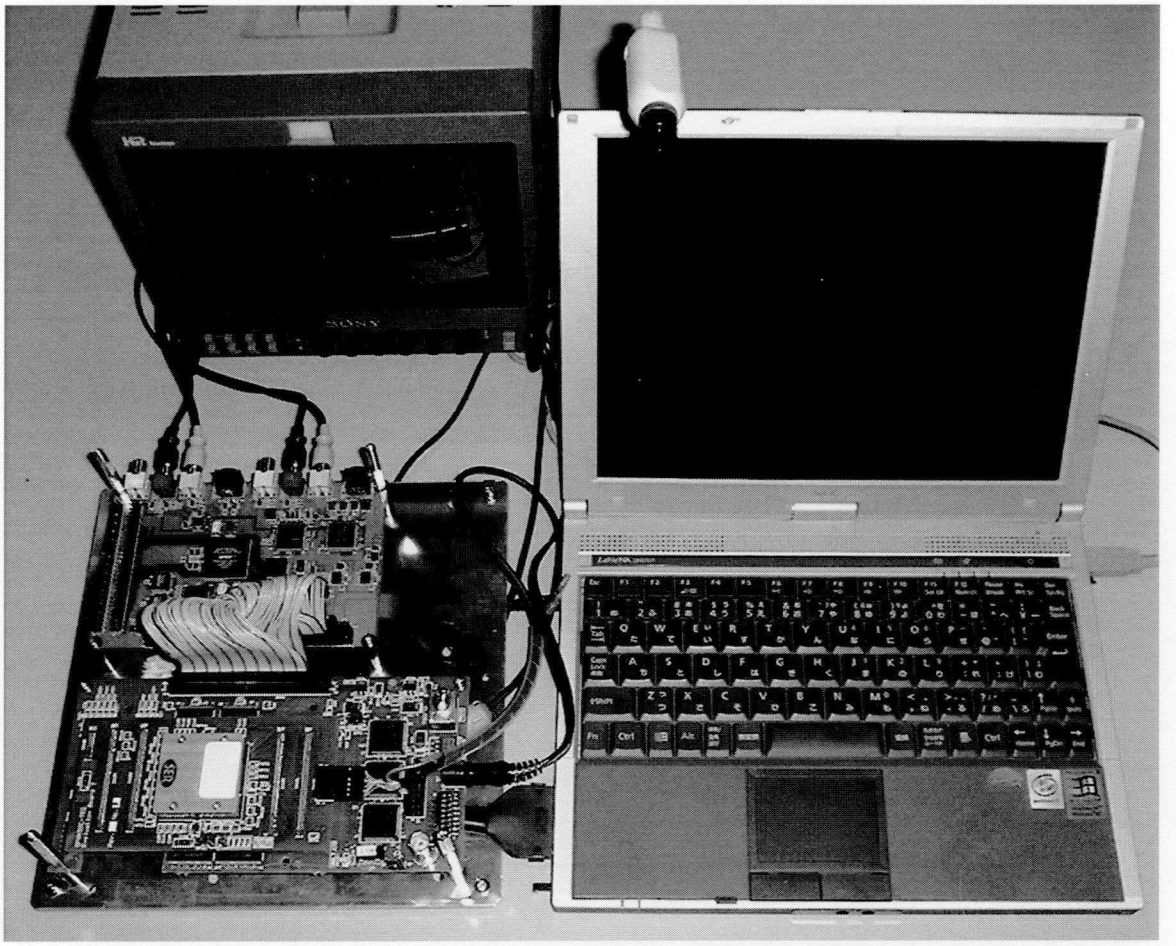


Figure 3.4: AV CODEC prototype system.

different memory areas shown in Table 3.2. In order to minimize the need for SPXK5SC external memory access, instructions and data accessed frequently are allocated to the internal memory of the SPXK5SC. Instructions allocated to an external memory are accessed by the instruction cache of the SPXK5SC. Data transfer between the SPXK5SC internal data memory and the external memory is controlled by the DMAC. The only external-memory-stored data that is directly accessed by the SPXK5SC itself are constant data for AMR speech CODEC and coded bit stream of the video CODEC.

3.5 Performance evaluation

The bus traffic and number of cycles were measured during AV CODEC execution under conditions described in Table 3.3, and also Figures 3.5 and 3.6. During AV CODEC execution at 75-MHz

Table 3.1: Functional specification of the AV CODEC prototype system.

SPXK5SC TEG	instruction memory and cache: 64 [KB] and 16 [KB] data memory: 64 [KB] AHB interface
SRAM1	1 [MB] (no wait)
SRAM2	2 [MB] (1-cycle wait)
MPEG-4 video CODEC	MPEG-4 visual simple profile image size: up to CIF (352x288) instruction size: 89.4 [KB] data size: 53 [KB] frame memory: 920 [KB]
AMR speech CODEC	bit rate: 12.2 - 4.75 [Kbps] instruction size: 65 [KB] data size: 47 [KB]

Table 3.2: Memory allocation of the prototype system.

SPXK5SC inst. memory	42.6 [KB] (video)	22.9 [KB] (speech)
SPXK5SC data memory	52.9 [KB] (video)	11.7 [KB] (speech)
SRAM1 (no wait)	46.8 [KB] (video, inst.) 920 [KB] (video, frame buffer)	42.0 [KB] (speech, inst.)
SRAM2 (1-cycle wait)	49.2 [KB] (video, bit stream)	35.4 [KB] (speech, data)

SPXK5SC and 25-MHz AHB, the DSP load for the emulation was 75%, and 30% of the AHB bandwidth, i.e., 33 Mbytes/sec, was used. At 100-MHz SPXK5SC and 25-MHz AHB, the DSP load was 62%, and the same 30% of the AHB bandwidth was used.

As may be seen in Figure 5, the respective number of average cycles per second for encoding and decoding in case of 100-MHz SPXK5SC and 25-MHz AHB is a bit larger than that of 75-MHz SPXK5SC and 25-MHz AHB. This means that the frequency ratio between the SPXK5SC and the AHB can affect average cycles for encoding and decoding. The DSP core occasionally accesses the external memory during encoding and decoding, and the DSP core has to wait sometimes until the DMA finishes transferring data between the external memory and the internal memory of the DSP core. These data accesses related to the external memory can take a number of cycles at a time. So, larger frequency ratio between the SPXK5SC and the AHB increases average cycles for encoding and decoding. Further, as may be seen in Figure 3.6, at an unchanging AHB frequency of 25 MHz, changes in the SPXK5SC frequency did not affect bus traffic. Since it was necessary to map most of the constant data for AMR speech CODEC on an external memory shown in Table 3.2, the number of cycles for AMR speech CODEC is higher than desirable; if all instructions and data for AMR speech CODEC could be mapped on the internal memory of the SPXK5SC, the number of cycles could be held to 16 Mcycles or lower.

The performance of the AV CODEC was estimated on a 250-MHz SPXK5SC and a 83-MHz AHB for a target LSI, maintaining a frequency ratio of 3:1 in the emulation system. Cycle counts

and bus traffic for video CODEC depend on the number of pixels per frame and the frame rate. Cycle counts and DMAC bus traffic increase roughly in proportion to the number of pixels and the frame rate. VIF bus traffic increases only in proportion to the number of pixels. As may be seen in Figure 3.6, SPXK5SC bus traffic is mainly composed of instruction cache access and memory access used in the AMR speech CODEC. That is, it may be assumed here that SPXK5SC traffic is independent of the number of pixels or frames of the video CODEC. Further, although DMAC bus traffic includes memory access for AMR speech CODEC, that portion is so small that can be ignored. In addition, it may also be assumed that the frame buffer for the video CODEC is mapped on SRAM1 and can be accessed without wait cycles, and that the video bit rate in this estimation increases in proportion to image size and frame rate. As image size increases, SRAM1 will eventually be unable to contain the frame buffer, and the frame buffer will be placed in an SDRAM instead of SRAM1.

As may be seen in Table 3.4, which lists estimated cycle counts and bus traffic for the AV CODEC for various image sizes and frame rates, the combination of a 250-MHz SPXK5SC and a 83-MHz AHB is able to handle up to QVGA 30fps video CODEC and has the capability to execute AMR speech CODEC. In this case, however, bus occupation is nearly 50%, which means that there will be excessive conflict among bus masters, and some remedy, such as a multi-layer AHB, will be needed.

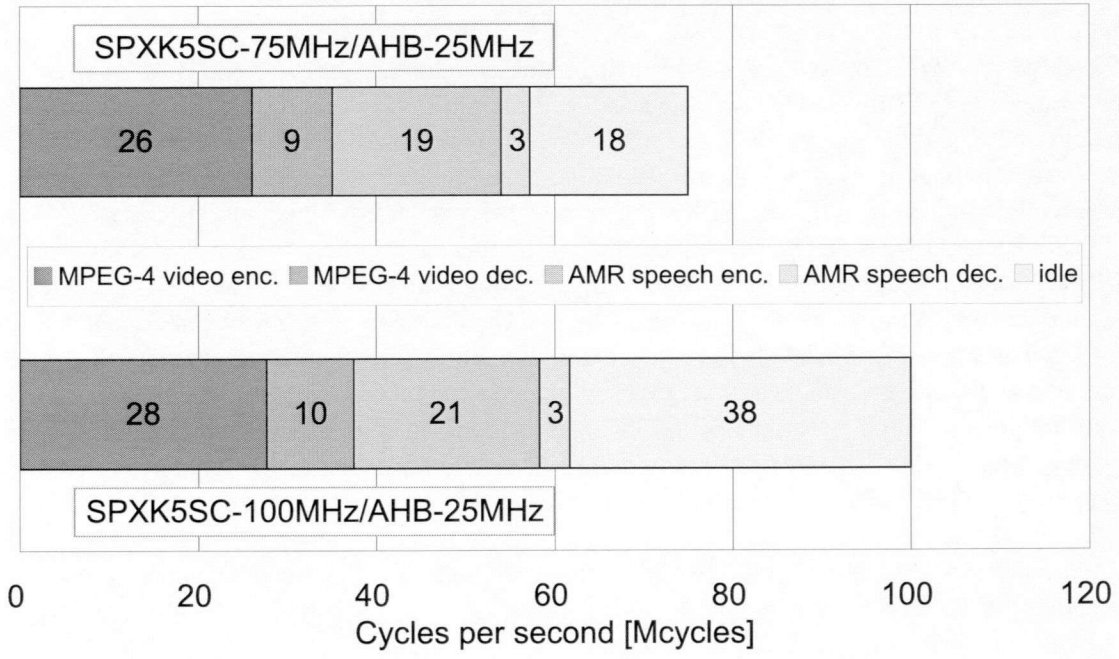


Figure 3.5: Average cycles per second of each function during AV CODEC execution.

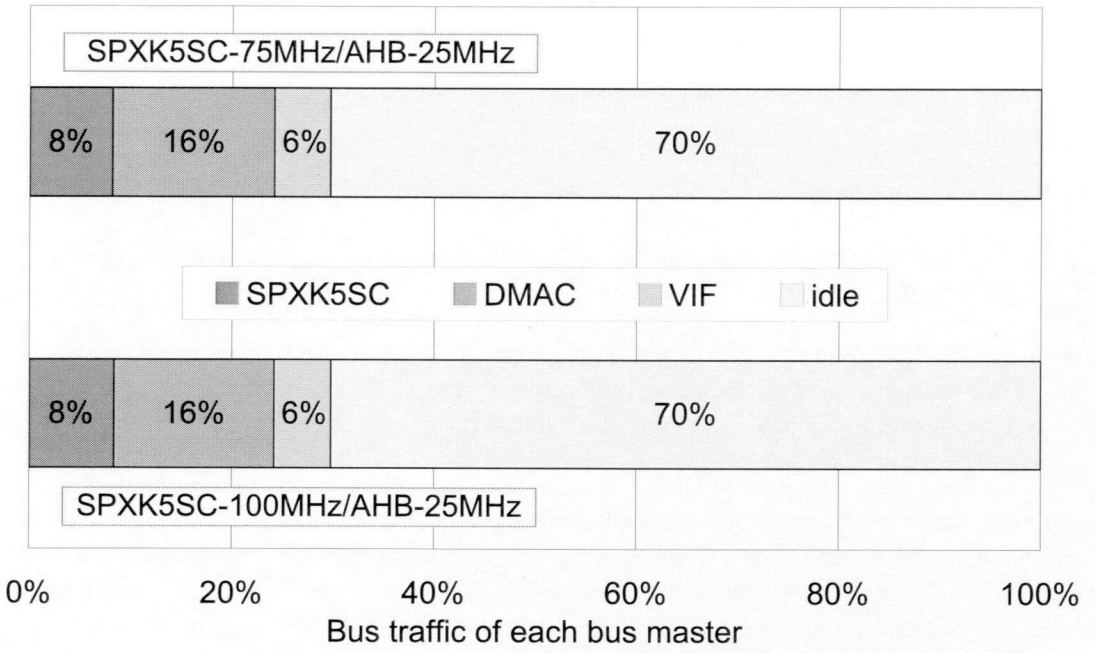


Figure 3.6: Average bus traffic per second of each bus master during AV CODEC execution.

Table 3.3: AV CODEC parameters for performance evaluation.

Video frame size and rate	QCIF (176x144), 15 [fps]
Video bit rate	64 [Kbps]
Speech bit rate	12.2 [Kbps]

Table 3.4: Estimated cycle counts and bus traffic of the AV CODEC on 250-MHz SPXK5SC and 83-MHz AHB.

	Cycle count [Mcycles]	Traffic [Mbytes/sec]	Occupation of 83-MHz AHB [%]
QCIF 15fps	57	30	9%
QVGA 15fps	128	73	22%
CIF 15fps	163	94	28%
QCIF 30fps	92	51	15%
QVGA 30fps	235	138	42%
CIF 30fps	303	180	54%

3.6 Architecture design based on the emulation system

Through the careful observation of the AV CODEC operation implemented on the emulation system, the following results have been found out:

- The 250-MHz SPXK5SC DSP core is able to perform QVGA-30fps video CODEC.
- DMAC bus traffic and VIF bus traffic occupy most of overall bus traffic during QVGA-30fps video CODEC.
- Since the QVGA-30fps AV CODEC use 42% of 83-MHz AHB bandwidth, a system LSI having the same architecture as the emulation system might not be able to handle well overall bus traffic during QVGA-30fps video CODEC due to access conflicts between the DMAC and VIF.

These mean that the computational performance of the DSP core would be sufficient for QVGA-30fps video CODEC but the bus transfer capability of the architecture might be insufficient. In order to efficiently utilize the system bus bandwidth, DMAC bus traffic and VIF bus traffic in that architecture should be separated. Since both the DMAC and VIF deal with image data stored in memory, the image data should also be placed in separated memory banks to make it sure that the DMAC and the VIF can access them concurrently. Based on these investigations, a DSP has been designed so as to be suited to AV CODEC.

The emulation system has been used to develop the μPD77050, the first DSP integrated with the SPXK5. The μPD77050 consists of a SPXK5SC DSP core, a 256 Kbyte SRAM, a 64 Kbyte ROM, and several selected peripheral components integrated together by two 32-bit AMBA AHBs: DSPBUS and BRGBUS shown in Figure 3.7 and Table 3.5. The SPXK5SC in the μPD77050 can operate at up to 250 MHz, and the two AHBs at up to 83 MHz. Both instructions and data can be stored in the single 256 Kbyte SRAM, which can be sub-divided into 4 Kbyte memory banks like the internal memory of the SPXK5SC.

The peripheral components can be divided into two groups, PBUS peripherals and AHB peripherals. The PBUS peripherals contain such basic functions as an interrupt control unit, serial and parallel input/output, and timers. These peripherals do not require high throughput. The AHB peripherals, on the other hand, do require high throughput. They contain a direct memory access controller, two memory interfaces (EBIF and SDRAMIF), and an external AHB interface (SYSBUSIF). The SYSBUSIF acts as a system bus bridge between the BRGBUS and an external AHB. Note that the VIF is not included in the DSP; it can be connected to the DSP via the SYSBUSIF.

The SPXK5SC and each of the AHB peripherals other than the SYSBUSIF can be a master on the DSPBUS but not on the BRGBUS; only the SYSBUSIF can be a master on the BRGBUS. Because there are two AHBs, the external bus masters through the BRGBUS will not conflict on the BRGBUS with the SPXK5SC or the AHB peripherals. The idea of using a multiple bus has come from the experience gained in using the emulation system. The two AHBs help peripheral components of the μPD77050 to use the bus bandwidth more efficiently.

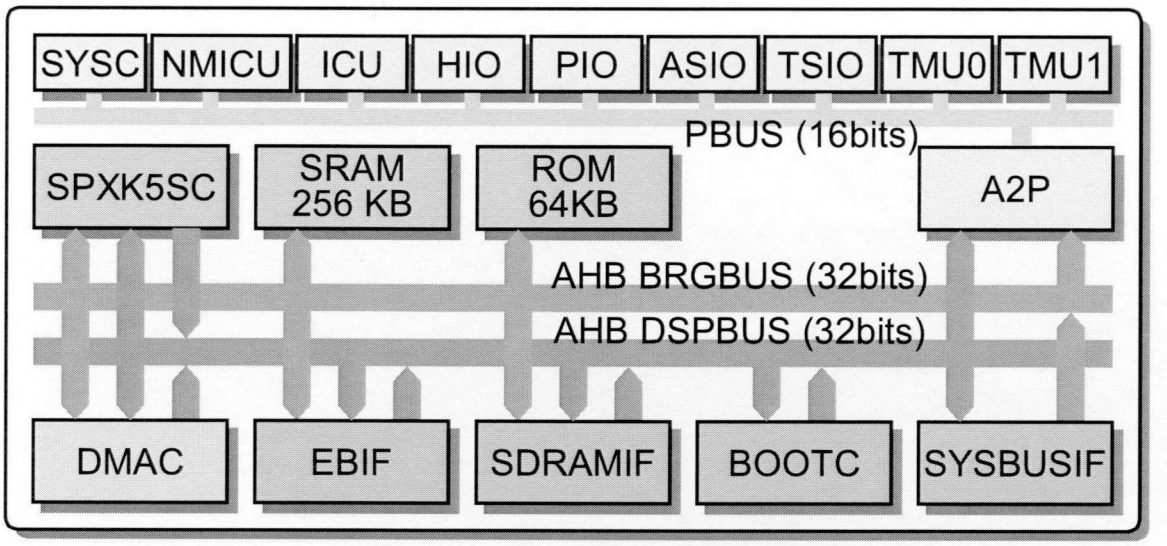


Figure 3.7: Block diagram of the μPD77050.

3.7 AV CODEC performance of μPD77050

Figure 3.8 shows an actual AV CODEC performance of the μPD77050, which was measured on a sample of the μPD77050. In this measurement, the DSP core was set to operate at 240 MHz and its internal AHB buses to operate at 120 MHz. Outside of the DSP, SDRAMs were connected to the DSP and operated at the same frequency of the AHB buses (120 MHz). The video frame buffer was placed onto the SDRAMs. As may be seen, the measured performance in Figure 3.8 is very close to the estimated performance in Table 3.4.

In previous work, an MPEG-4 video CODEC was implemented on the μPD77210, which is a single MAC DSP [23]. In that implementation, QCIF 15fps CODEC, for example, was processed at 47 Mcycles/sec and QVGA 15fps CODEC at 154 Mcycles/sec on the μPD77210. According to [28], TMS320C5510, which is a dual MAC DSP of Texas Instruments, needs 58 Mcycles/sec to perform QCIF 15fps CODEC. On the other hand, the respective cycle count for QCIF 15 fps and QVGA 30fps has been reduced to 36 Mcycles/sec and 115 Mcycles/sec on the μPD77050. That is, this means that the μPD77050 can perform MPEG-4 video CODEC with 25% less cycles than the μPD77210 does or 38% less cycles than the TMS320C5510 does. In addition, since the maximum operating frequency of the μPD77210 was 160 MHz, it could not afford to deal with QVGA 15fps CODEC and AMR speech CODEC at the same time. However, the μPD77050 can deal with them concurrently and there is still sufficient computational power available for other tasks such as video deblocking filter or speech noise reduction.

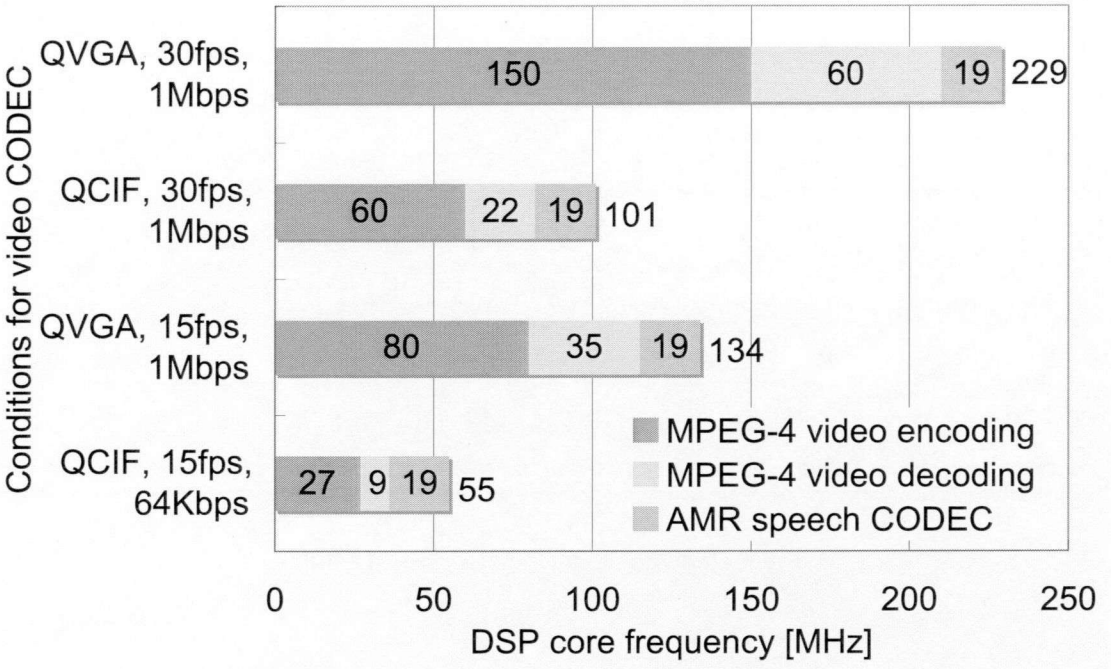


Figure 3.8: AV CODEC performance of μPD77050.

3.8 Summary

A method has been developed for using a DSP core TEG and FPGA-based hardware emulation that is capable of handling real-time workloads and can rapidly verify and evaluate target LSIs integrated with SPXK5SC cores. The presence of the SPXK5SC system bus interface and the use of FPGA devices allowed us rapid development of the real-time emulation system. The developed method would be applicable to other DSP cores that have a system bus interface. It has been found that the SPXK5SC can be used with AV CODEC (QVGA 30fps). The emulation system using a DSP core TEG, which has a bus interface, and an FPGA is suitable for overall system evaluation on real-time workloads as well as architectural investigation. In addition, the emulation system makes it possible to verify the operation of peripherals inside a μPD77050 and to evaluate AV CODEC software on a DSP for actual situations. Evaluation results obtained with the emulation system have been used as feedback in the development of a μPD77050.

Table 3.5: The μPD77050 specification.

Frequency	250MHz@1.5V, 180MHz@0.9V	
Power	0.23mW/MIPS@1.5V 0.09mW/MIPS@0.9V	
Memory	Instruction cache	16 [KB]
	Instruction RAM	64 [KB]
	Data RAM	64 [KB]
	SRAM	256 [KB]
	ROM	64 [KB]
Peripherals	DMAC	8-channel direct memory access controller
	EBIF	16-bit external bus interface, 16-MByte total addressable external memory space
	SDRAMIF	Single data rate SDRAM interface, 16 bits or 32 bits
	SYSBUSIF	External AHB interface
	BOOTC	Boot controller
	SYSC	System controller (clock control, power control, memory configuration)
	TSIO	Time-division serial input/output, 8 bits or 16 bits
	ASIO	Audio serial input/output, 32 bits or 64 bits
	HIO	Host input/output, 8 bits or 16 bits
	PIO	Sixteen general-purpose input/output pins
	TMU	Two 16-bit time units, 5 clock sources available
	ICU	Interrupt control unit, 16 maskable external triggers available
	NMICU	Non-maskable interrupt control unit

Chapter 4

Software Development Tool Generation for Architecture Design

This chapter addresses a method of software development tool generation suitable for instruction set extension of existing embedded processors. The key idea in the method proposed in this chapter is to enhance a base processor's toolchain by adding plugins, which are software components that handle additional instructions and registers. The proposed method generates a compiler, assembler, disassembler, and instruction set simulator. Generated compilers with the plugins provide intrinsic functions that are translated directly into the new instructions. To demonstrate that the proposed method works effectively, this chapter presents an experimental result of the proposed method in the study of adding SIMD instructions to the embedded microprocessor V850. In the experiment, by using intrinsic functions, the compiler generated efficient code with only 7% increase in the number of instructions against the hand-optimized assembly codes.

4.1 ASIPs and their software development tools

Application specific instruction set processors (ASIPs) are increasingly employed in embedded systems for multimedia and mobile wireless applications because they are programmable and provide better performance for applications. The flexibility of software allows designers to make late design changes or additions, and the instruction sets tuned for target applications improve performance. To design ASIPs, people recently use design tools to generate hardware description language (HDL) source codes or software development tools. There have been many previous approaches in terms of exploring both the processor architecture and instruction set architecture (ISA) for ASIPs using architecture description languages (ADLs) such as nML, LISA, and EXPRESSION [29–32].

One of the main challenges in making the best use of ASIPs is to provide software development tools such as assemblers and C compilers at the early stages of architecture design. To provide an optimized C compiler at early stages of architecture design, most ASIPs consist of application specific functional units and a base processor, which has a fundamental instruction set. Such ASIPs consisting of application specific functional units and a base processor are accepted in the market and are commercially available from several sources such as Tensilica. The design tools for the ASIPs have also been released by CoWare Inc., Target Compiler Technologies, and ASIP Solutions.

Looking closer at tool generation processes, most of these ASIP design tools and previous related works have focused on generating whole part of the tools from scratch and have not considered reusing existing tools for base processors or manually improving the generated tools. If existing processors are used as the base processors for the ASIPs, a complete set of tools is already available. The tools would include a hand-optimized compiler or a simulator equipped with features such as particular performance profiling. These features may not be always available on the tools generated by using the conventional methods.

Another important aspect on the tool generation for ASIPs is fundamental toolchain used for the tool generation. While most of ASIP design tools and previous related works [31–34] have been developed based on their own compilers, simulators, or binary tools, some of the conventional approaches [35–39] have used the GNU toolchain, which is an open-source and a de-fact toolchain in the field of embedded software development. Since the GNU toolchain supports many kinds of processors, it is very suitable to generate software development tools for the ASIPs based on existing embedded processors. However, excepting the tool generation method for the Xtensa [39], the conventional tool generation methods targeted at the GNU toolchain have mainly focused on part of the toolchain like GNU Binutils not the whole part. Although the tool generation method for the Xtensa can generate all the tools based on the GNU toolchain, its target processor is limited to the Xtensa and does not support any other processor architectures. Therefore, a new method of generating tools based on the GNU toolchain is required.

This chapter describes a method of software development tool generation for instruction set extension of existing embedded processors. The key idea behind the proposed method is to enhance a base processor's toolchain by adding plugins, which are software components that handle additional instructions and registers added to the base processor. Plugins are generated from the specification information of the additional instructions and registers. The only modification that needs to be made to the base processor's toolchain is to provide the sockets to accept the plugins. This chapter overviews the proposed approach in terms of how the specifications for additional resources are described, what plugins are generated to handle the additional resources, and what modifications need to be made to the base toolchain to accept plugins. In addition, the experimental results on the study of adding SIMD extension to the V850 microcontroller are presented.

The rest of this chapter is organized as follows. Section 2 introduces the proposed tool generation method. Section 3 gives an outline of the ISA description language for the proposed tool generation. Section 4 describes an experimental result. Section 5 discusses the difference between the proposed method and related work. Finally, section 6 concludes this chapter.

4.2 Software development tool generation using plugins

The goal of tool generation addressed in this chapter is to provide an efficient way of generating the software development tools for instruction enhanced processors, which are based on existing microprocessors. To achieve this goal, a method using plugins is chosen, where software components are added as plugins to the existing toolchain to handle additional features, e.g., for parsing, encoding, or decoding new instructions. The plugins allow the existing toolchain to continue to be used and to be enhanced its functionalities for new instructions by adding more plugins. As a result, the proposed method can generate a compiler, assembler, disassembler, linker, and simulator

for instruction enhanced processors.

4.2.1 Tool generation flow

Figure 4.1 shows the concept flow underlying the proposed method for tool generation. The flow begins with a specification document written in extensible markup language (XML). The XML document contains an additional ISA specification for the target processor. Here, it has been assumed that the target processor's ISA consists of the base processor's instruction set and the additional instruction set which is written in the XML document. The additional ISA specification includes the base processor's name, the base processor's register information, and the specifications for new registers and instructions added to the base processor. However, it does not include the definition of the base processor's instruction set. Designers write this XML document for their target processor and feed the document to the tool generator.

The tool generator, then, generates the plugins to be added to the software development tools of the base processor (base tools). The base tools are modified once in advance so that they have sockets that can be connected to the plugins, the plugins can be integrated with the base tools, and the integrated software can become enhanced tools for the target processor. The enhanced tools with the plugins have four main features of assembling, disassembling, relocating, and executing the new instructions, and they also provide intrinsic functions translated by the enhanced compiler directly into the new instructions.

4.2.2 Structure of the generated plugins

What distinguishes the proposed method from conventional methods is to propose plugins produced from templates and parameters and to easily add the support for new instructions to existing toolchain for base processors. This work is a very complicated task and requires deep knowledge about the GNU toolchain if you do manually the same things. Since the templates are designed so as to support a variety of instructions, the specification of desired new instructions can be described in a simple format. In addition, generating plugins from templates and parameters increases readability of the plugins and decreases possibility of involving errors in the plugins.

Each command of the GNU toolchain such as gas has an internal flow in which it identifies one instruction and another extracted from input programs and performs particular operations on the instruction. The operations performed to instructions can be composed of several fundamental sub-operations. Field structures and syntaxes of instructions determine which sub-operation is used and how sub-operations are mixed. The proposed method represents a combination of sub-operations as a template and parameters, which will become a plugin. Templates are code fragments commonly used for every instruction, and parameters are a variety of information items on instructions. The parameters drive the templates to perform particular operations. Such generation scheme allows the structure of the generated plugins to be easily understood and to be verified if the generated plugins work correctly. If different control code fragments were generated for different instructions, it could make the tool generator complicated and make it difficult to verify the generated plugins since the generated plugins would differ greatly depending on instructions.

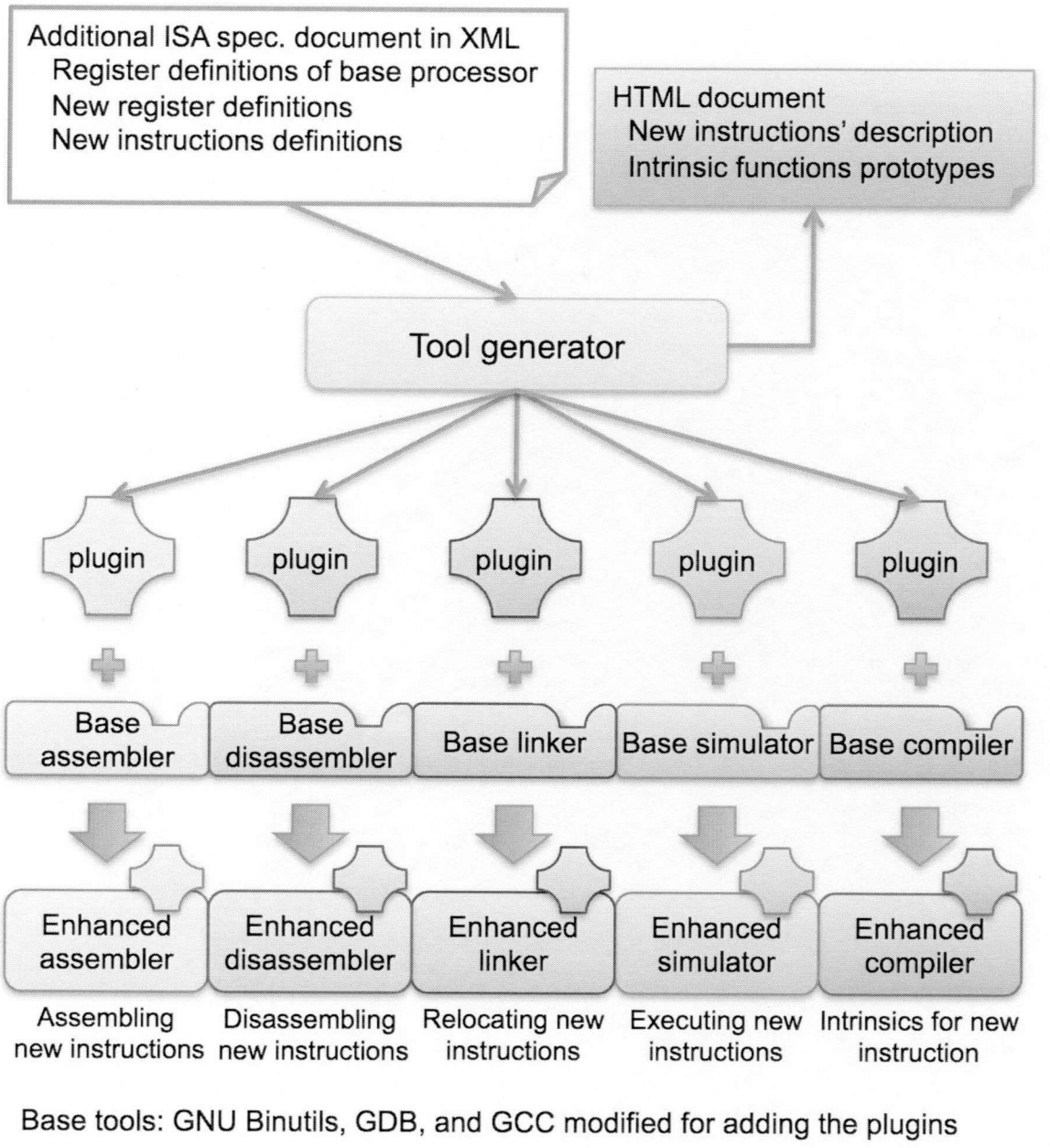


Figure 4.1: Concept flow underlying tool generation.

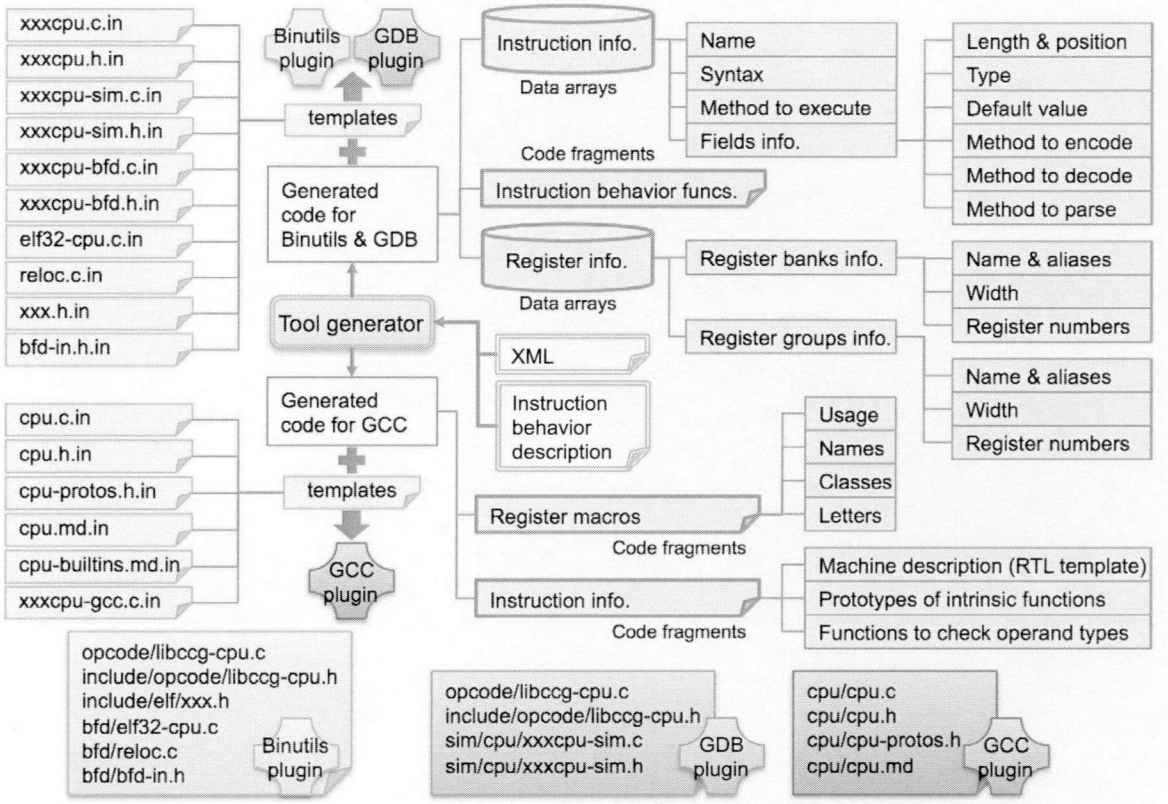


Figure 4.2: Generating plugins for GNU Binutils, GDB, and GCC.

Figure 4.2 shows the process to generate the plugins for the GNU toolchain in detail. The tool generator generates the plugins by adding data arrays and code fragments into pre-described template files. While the template files are commonly used, the data arrays and code fragments vary depending on XML documents and instruction behavior description files.

Template files have common functions used to handle any new instructions and registers. For example, the common functions include functions for encoding, decoding, assembling, and disassembling new instructions. These common functions perform their operations according to the information on new instructions and registers. The information is generated into plugins as structure data arrays, which are used as a database on new instructions and registers.

For GNU Binutils and GDB, instruction information and register information required for assembling, disassembling, linking and executing are generated as data arrays, and the body of instruction behavior functions are generated as code fragments. The information for each of instructions includes a name, syntax in assembly language, pointer of the method to execute the instruction, and the information on the instruction fields. Each of the instruction fields information further includes its length and position in the instruction codeword, a field type to specify what the field represents (register, immediate, or operation code), a default value expression of the field, and three pointers to methods for encoding/decoding/parsing.

For the GCC, register macros and instruction information are generated as code fragments. On the GCC, the register-related macros give the registers information such as their usage, names, classes, and letters to represent registers. The instruction information includes machine description, prototype definitions of the intrinsic functions, and functions to check operand types for the instructions.

4.2.3 Internal tool flow working with the plugins

The base tools and the plugins work together on the enhanced tools for the target processor, as shown in Figure 4.3, which is a simplified flow chart on how the plugins are employed to process each of instructions for the target processor. On the enhanced tools for the target processor, instructions are processed through either a conventional procedure or an additional procedure. First, the enhanced tools determine whether a given instruction is of new instructions added to the base processor or not. Then, the enhanced tools process the instruction through an appropriate procedure. The conventional procedure is a path to handle the instructions of the base processor, and the additional procedure, which is a feature provided by the plugins, is a path to handle the new instructions added to the base processor. In order to make the enhanced tools behave in this way, the base tools need to be modified at once so as to work with the plugins.

4.2.4 Assembling and encoding new instructions

Here, how the enhanced assemblers process new instructions is outlined. The parser plugin built in the enhanced assemblers can handle any one of three instruction syntax styles in assembly language: a mnemonic style, a function style, and an algebraic style. In the mnemonic style, a mnemonic comes first and is followed by several operands. In the function style, input operands and an output operand are denoted respectively as function arguments and a return value. In the algebraic style,

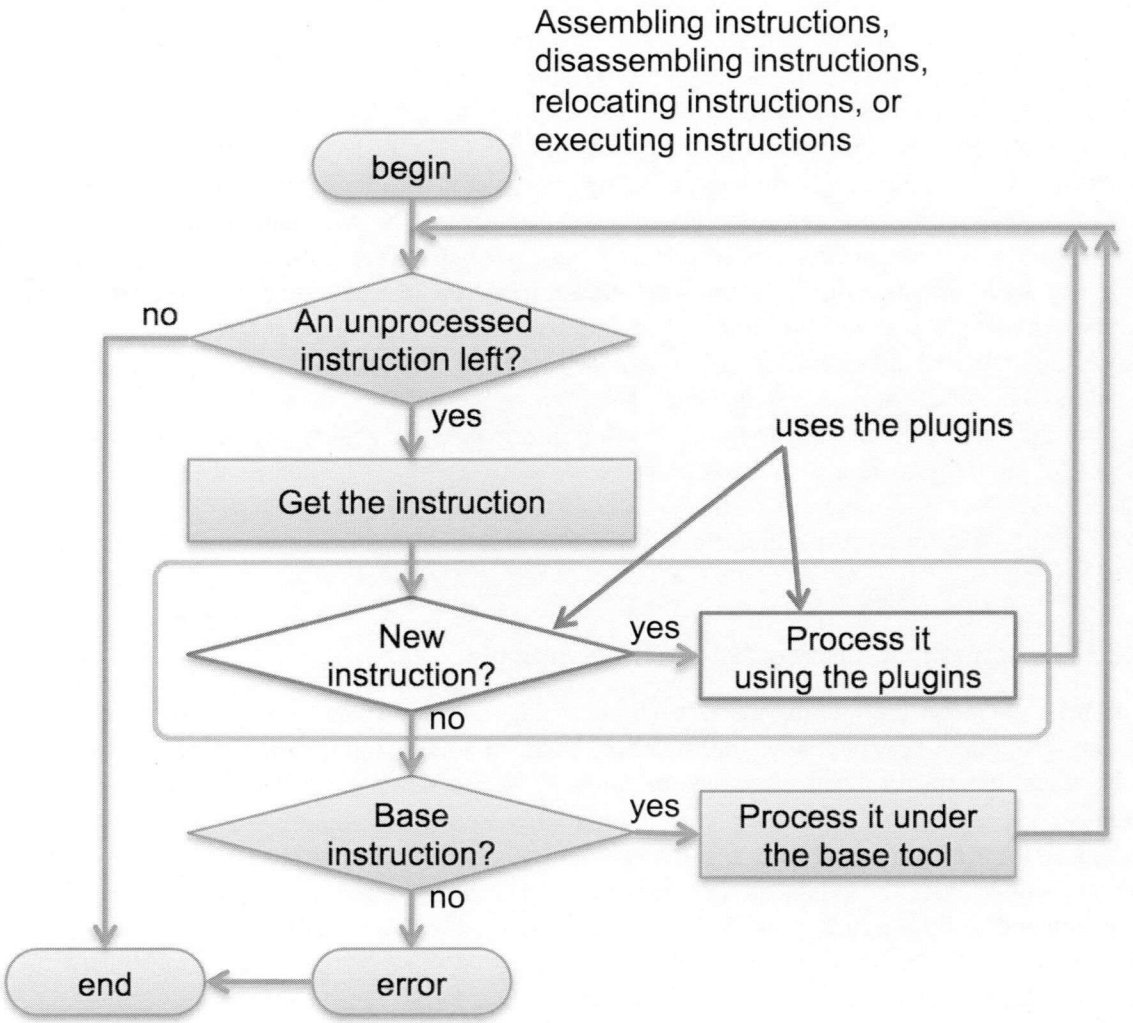


Figure 4.3: Tool internal flow for enhanced assemblers, disassemblers, linkers, and simulators working with plugins.

operators such as '+', '-', and '*' denote instructions' operations and '=' is used to specify a destination operand.

Mnemonic style:	`add3i r4,r5,0x22`
Function style:	`r4=add3i(r5,0x22)`
Algebraic style:	`r4=r5+0x22`

The mnemonic style is the most major syntax style, and the latter two styles are used favorably for digital signal processors to make it easy to understand their assembly codes. Supporting these three styles contributes to increasing the freedom to choose syntax styles for readability.

The parser plugin processes new instructions as shown in the following five steps and Figure 4.4. After assembling a new instruction, if any of operands refers to an unresolved symbol, the enhanced assemblers make relocation information for the unresolved symbol.

1. **Token decomposition:** The parser plugin breaks the input string into tokens, which are classified into either a symbol token, expression token, or a code token. In Figure 4.4, there are 6 tokens.

2. **Candidate selection:** The parser plugin chooses instruction candidates which have the same number of tokens as in the input instruction. In Figure 4.4, three instructions are selected as candidates.

3. **Token comparison:** For each of candidates, the parser plugin compares tokens at the corresponding position of the input instruction and a candidate, and find the instruction X which fits to the token pattern of the input instruction. In Figure 4.4, the instruction `add3i` has the same token pattern as of the input instruction.

4. **Operand calculation:** The tokens corresponding to operand variables like `%xxx` in the instruction X are operands. The parser plugin calculates operand values from those tokens of the instruction X. For example, if a token corresponding to an operand variable represents a register name, the assembler translates the name to an operand value.

5. **Instruction encoding:** The parser plugin encodes the input instruction by using the operation code of the instruction X and the operand values of the input instruction.

4.2.5 Machine description of new instructions

The tool generator generates machine description of the new instructions. The machine description is the instruction patterns of the new instructions, and is used in the processor-dependent components of the GCC. The instruction patterns are written in the GCC's intermediate representation called the register transfer language (RTL). Figure 4.5 shows an example of the generated machine description.

Normal instruction patterns defined using the statement of `define_insn` usually have behavior description specified in their contents. With the behavior description given in the pattern,

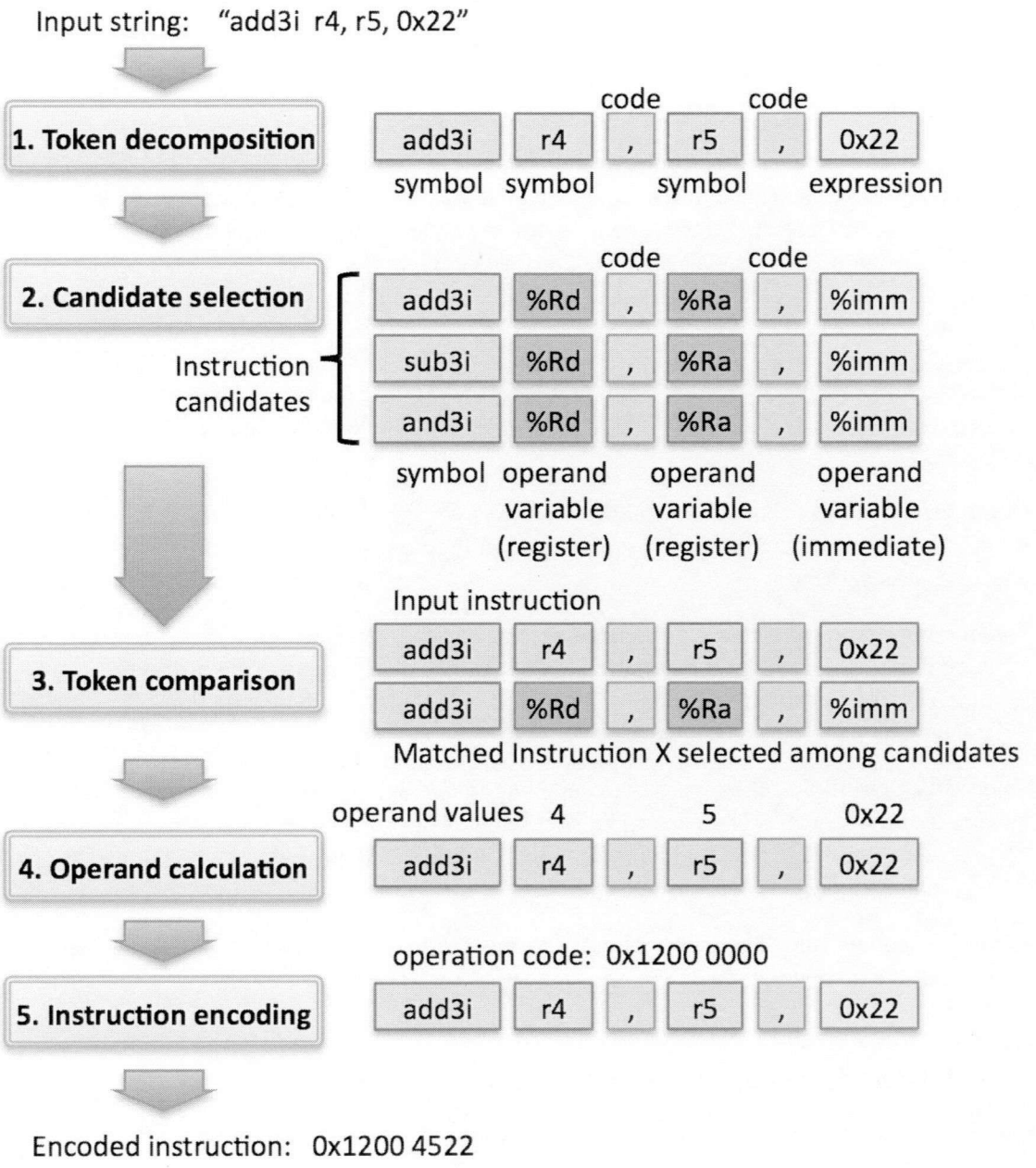


Figure 4.4: Assembling instructions on the plugins.

```
 1: ;; syntax:  myadd reg1, reg2, reg3
 2: (define_insn "builtin_cpu_MYADD"
 3: [
 4: (set
 5: (match_operand:SI 0 "cpu_gpr_regs_operand" "=r")
 6: (unspec:VOID [
 7:  (match_operand:SI 1 "cpu_gpr_regs_operand" "r")
 8:  (match_operand:SI 2 "cpu_gpr_regs_operand" "r")
 9: ]  UNSPEC_BUILTIN_MYADD))
10: ]
11: ""
12: "myadd %1, %2, %0"
13: [(set_attr "length" "4")]
14: )
15: ;; Define the default latency of 'myadd' as 1.
16: (define_insn_reservation    "MYADD"  1
17:   (eq_attr "insn_mnemonic"  "MYADD")
18:   "nothing"
19: )
```

Figure 4.5: A generated instruction pattern.

the GCC recognizes what kinds of operations the instruction performs. The generated instruction patterns, however, do not have well specified meaningful behavior, but just simple information that gives input and output operands. With the information on input and output operands given for machine description, the GCC can recognize which operand is used in which instruction and can also schedule instructions without meaningful behavior in machine description. Therefore, it was decided to omit meaningful behavior from generated instruction patterns.

The statement of `define_insn_reservation` at lines 16-19 simply defines the default latency for a new instruction. This statement allows the GCC to determine how many cycles are taken before the output operand of the new instruction becomes available. Instruction scheduling enabled by this statement gives a chance to improve the performance of the compiler-generated code. In particular, this is important for high-latency load instructions to read data from memory.

If a target processor has move instructions for new registers which do not exist in the base processors, the tool generator generates move instruction patterns from and to the new registers. The generated move instruction patterns have meaningful behavior description, so that the GCC can recognize that they perform move operations and the GCC can use them in code generation process.

4.2.6 Adding intrinsic functions on the GCC

In order to add intrinsic functions on the GCC, two macro functions `TARGET_INIT_BUILTINS` and `TARGET_EXPAND_BUILTIN` are used. `TARGET_INIT_BUILTINS` represents the name of

a function that performs initialization for intrinsic functions. For each of intrinsic functions, the initialization function makes registration of the information such as a return value type, arguments types, a function name, and an ID number. The tool generator generates statements for this registration of intrinsic functions. The generated statements are inserted into the function represented by the macro `TARGET_INIT_BUILTINS`.

`TARGET_EXPAND_BUILTIN` represents the name of a function that generates instruction patterns from given intrinsic functions. The function corresponding the macro `TARGET_EXPAND_BUILTIN` determines which instruction pattern should be used for a given intrinsic function and how arguments of the given intrinsic function should be translated into operands of the instruction pattern. The tool generator generates tables used for retrieving instruction patterns and their operand types from given intrinsic functions.

On the GCC, instead of intrinsic functions, there is another way to use new instructions added to the base processors, that is, inline assembly can also exploit the new instructions in C codes. Intrinsic functions, however, is better than inline assembly in terms of the following points.

- There is possibility that the GCC can schedule the instructions corresponding to intrinsics and optimize them if information on the instructions such as a latency and code size is given to the compiler. Inline assembly, however, has no means for better scheduling and optimization.
- The GCC can check types of output and input operands for intrinsic functions. Inline assembly, however, cannot do that.
- Although inline assembly is available only for its target processors, intrinsic functions could be emulated on non-target processors if emulation functions for the intrinsic functions were provided.

These reasons led to the choice of intrinsic functions rather than inline assembly.

Several methods [40–42] for automated instruction set extension have been reported so far, which generate compilers that can exploit the extended instructions without the need for modifying source codes of applications. Contrary to this, on the proposed method, the enhanced compiler does not automatically exploit the new instructions and programmers have to invoke intrinsic functions if they want to make the compilers to use the new instructions. However, changing algorithms or rewriting source codes for more speed is still important in many practical situations. Therefore, using intrinsic functions to accelerate target applications is a practical method.

4.3 Instruction set description using XML

This section explains the additional ISA specification in the proposed framework. The proposed tool generation flow begins with an XML document, which contains an additional ISA specification for the target processor. The XML provides a flexible and extensible framework for representing and structuring all kinds of data. In addition, XML is widely used in the World Wide Web community as a means of structured information exchange, and there are many software components and libraries that handle XML documents. In the case, of this framework an XML document is used to describe the specification of new registers and instructions to be added to the base processor.

The XML of the proposed method can describe not only ISAs of general RISC processors but also the following complex instructions:

1. instructions that have an operand with a restriction, e.g., the operand must be an even number register.
2. instructions in which an operand value is separated and placed into two different instruction fields. This tricky field arrangement might be used when there is not enough instruction field space for additional instructions.
3. instructions in which a value calculated from an operand is placed into an instruction field. This calculation might happen when the least significant n-bits of an immediate operand is trimmed off before the operand is placed into an instruction field.
4. instructions that use pair registers that consist of two contiguous registers.
5. instructions that have two or more output operands or have many input operands (The number of total operands must be less than or equal to ten).
6. instructions in which any combinations of the above ones are used.

Although these complex instructions do not appear in general RISC processors, in fact, they are used in the experiment based on the V850 processor described afterward. In addition, since the behavior of instructions is written in C language apart from XML documents, any kinds of operations of instructions can be described.

Here is a simple example in which the new instruction MYADD depicted in Figure 4.6 is added to a base processor. Figure 4.7 shows an XML document for the simple example. The XML document contains:

1. the base processor's nickname defined by the `<nickname>` tag,
2. the base registers on the base processor and new registers added to it defined by the `<register_bank>` tag,
3. new instructions defined by the `<insn>` tag,
4. GDB register numbers defined by the `<gdb>` tag, and
5. the file name containing the behavior of instructions defined by the `<behavior>` tag.

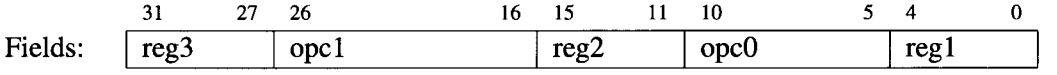
The behavior of the new instructions in the XML document is described in C language in a different file.

Since new definitions of instructions are important for generating the enhanced tools and they take up many lines in an XML document, the following sections will explain how instructions and registers are defined in an XML document.

4.3.1 Register definition

Register information is described using three tags: `<register_type>`, `<register_bank>`, and `<register_group>`. The `<register_type>` tag defines a register-type name and a

Syntax: `MYADD reg1, reg2, reg3`



Name	Type	Bits	Description
reg1	GPR	5	Register index of GPR registers
reg2	GPR	5	Register index of GPR registers
reg3	GPR	5	Register index of GPR registers
opc0	opcode	6	Bit pattern to distinguish it from other instructions
opc1	opcode	11	Bit pattern to distinguish it from other instructions

Figure 4.6: Instruction field structure of MYADD.

register length in bits. The `<register_bank>` tag defines a register bank, which consists of the same type of registers defined by the `<register_type>` tag.

The `<register_bank>` tag has several attributes, which are '`type`', '`size`', '`prefix`', '`base`', and '`letter`'. The '`type`' denotes a register type, which is one of the register types defined by the `<register_type>` tag. The '`size`' denotes the number of registers included in the register bank. The '`prefix`' denotes a prefix of register names. The prefix and a register index in the register bank become a register name. The '`base`' denotes whether the base processor has the register bank or not. The '`letter`', which is not used in Figure 4.7, denotes a letter for the GCC to represent a register class.

The `<register_group>` tag, which is not used in Figure 4.7, defines a register group that consists of several registers. The registers in a register group must be a register in any of the register banks and may belong to different register banks. The register banks and register groups defined by the tags `<register_bank>` and `<register_group>` are used as register operand types in `<insn>` tag.

4.3.2 Instruction definition

Each new instruction is defined using the `<insn>` tag. The `<insn>` tag has several child tags in its content. Here, important tags to define a new instruction are outlined. The `<mnemonic>` tag defines a mnemonic of the new instruction. Since the mnemonic is used as an ID of the new instruction, it must be different from other instructions' mnemonics. The `<syntax>` tag defines an instruction syntax, in which the operands of the instruction are denoted by names that begin from `%`, e.g. `%reg1` or `%reg2`. The instruction syntax may be formatted in any one of three styles: a style of mnemonic plus operands, a function style, or an algebraic style. The plugins are constructed to be able to accept these styles. The `<length>` tag defines the length of the new instruction in bits. If this tag is omitted, the default instruction length defined by the `<insn_length>` tag is used.

The `<field>` tag defines each of the instruction fields that are part of an instruction code word. If the new instruction's code word has many different fields, the designer writes `<field>` tags for all the fields. The `<field>` tags are supposed to be listed from the least significant bit (LSB)

```
 1: <Processor>
 2: <nickname>cpu</nickname>
 3: <register_type length="32">GPR_type</register_type>
 4: <register_bank type="GPR_type" size="32"
 5:                prefix="R" base="true">GPR</register_bank>
 6: <insn_length>32</insn_length>
 7: <insn>
 8:  <mnemonic>MYADD</mnemonic>
 9:  <syntax>MYADD %reg1, %reg2, %reg3</syntax>
10:  <field type="GPR"    length="5">reg1</field>
11:  <field type="opcode" value="0b11_1111" length="6">
12:    opc0</field>
13:  <field type="GPR"    length="5">reg2</field>
14:  <field type="opcode" value="0b111_1100_1000" length="11">
15:    opc1</field>
16:  <field type="GPR"    length="5">reg3</field>
17:  <input>
18:    <operand type="GPR" width="32">reg1</operand>
19:    <operand type="GPR" width="32">reg2</operand>
20:  </input>
21:  <output>
22:    <operand type="GPR" width="32">reg3</operand>
23:  </output>
24:  <description>
25:    This instruction calculates the sum of the contents
26:    of registers reg1 and reg2, and stores the result
27:    into register reg3.
28:  </description>
29: </insn>
30: <gdb>
31:  <regnum name="R0">0</regnum>
32:  <regnum name="R1">1</regnum>
33:  ....
34:  <regnum name="R31">31</regnum>
35: </gdb>
36: <behavior>cpu-isa.c</behavior>
37: </Processor>
```

Figure 4.7: Example of an XML document with additional ISA specification.

to the most significant bit (MSB). The <field> tag has several attributes, which are 'type', 'subtype', 'length', and 'value'. The 'type' denotes an instruction field type, i.e., what the instruction field represents. The field type may be either a register, an immediate value, or an operation code. If the field type is a register, its register type name is specified in 'subtype'. The 'length' denotes the length of the instruction field in bits. The 'value' denotes the numeric value of the instruction field for immediate values and operation codes. The numeric value can be represented as an expression, e.g., 0xF<<2, and the values of fields can be referred to in the expression using field names.

When a field's value is represented as an expression using the values of other fields, the tool generator needs to know how to obtain the values of instruction operands from the values of instruction fields. In this case, <disas> tag is used, which is not used in Figure 4.7. The <disas> tag denotes an instruction operand type and how to calculate its value from the instruction fields. The <disas> tag also has the same attributes that <field> tag has. The calculation expression is represented in the attribute 'value'. If an operand value can be obtained directly from the corresponding field value, you do not need to use <disas> tag.

If the new instruction has an immediate operand in its syntax and if the name of the immediate operand is defined by <field> tag or <disas> tag, the tool generator creates a new relocation type for the immediate operand. A new relocation type is also created if the attribute 'value' of the <field> tag includes a variable named 'cia' (current instruction address) or 'nia' (next instruction address).

The <input> and <output> tags define the input and output operands of the new instruction. Each of the input and output operands is defined by the <operand> and <memory> tags. The new instructions can have multiple input operands and multiple output operands. The tool generator uses the information given by these tags when generating the plugins of the compilers for the target processor. The prototypes of intrinsic functions and machine descriptions of the new instructions are defined according to the definition of the <input> and <output> tags. If a new instruction has an output operand, the intrinsic function corresponding to it returns the output operand. If a new instruction has multiple-output operands or no output operands, the intrinsic function corresponding to it becomes a void-type function, and the output operands are passed as arguments of the intrinsic function.

The <latency> tag, which is not used in Figure 4.7, defines the default latency of the new instruction. The default latency indicates how many cycles are consumed before the output operand of the new instruction becomes available. In the case of the default latency is larger than one it is used to generate fragments of machine descriptions for instruction scheduling.

4.3.3 Instruction behavior definition

Designers describe the behavior of a new instruction in C language in a different file other than in an XML document. There is an example of a behavior description in Figure 4.8, which is the behavior description for the new instruction defined in the XML document in Figure 4.7. The behavior of a new instruction is described in a single function. The function begins with 'behavior *(mnemonic)*' and has several unspecified arguments available in the function. The arguments are the values of instruction fields other than those of operation codes. Each of the arguments has the name defined by the <field> tag or <disas> tag.

```
 1: /* MYADD:  mnemonic defined by <mnemonic> tag */
 2: /* syntax: MYADD %reg1, %reg2, %reg3 */
 3: behavior (MYADD)
 4: {
 5:   /* GPR: register bank name. */
 6:   /* reg1, reg2, and reg3: register index */
 7:   int32_t val1 = REG_read32 (GPR, reg1);
 8:   int32_t val2 = REG_read32 (GPR, reg2);
 9:   int32_t val3 = val1 + val2;
10:   REG_write32 (GPR, reg3, val3);
11: }
```

Figure 4.8: Behavior description of instruction MYADD written in C language.

There are three non-operational-code fields in Figure 4.7: `reg1`, `reg2`, and `reg3`. They are all register operands and the same named arguments are available in the behavior function in Figure 4.8. Designers can access registers and memory in the behavior function through functions such as `REG_read32(REGTYPE,IDX)` and `MEM_read32(ADDR)`. The macro `NIA_SET(ADDR)` is used to modify the program counter and the macro `CIA` to get the content of the program counter.

4.4 Experiment

Here, the experiment on the tool generation using the proposed method is explained. The base processor used in this experiment is the V850 microcontroller. The V850 is a RISC processor optimized for embedded systems [43]. In the experiment, the proposed method generates a toolchain including a compiler for the target processor that consists of the V850 microcontroller and an SIMD extension.

Using the generated compiler and intrinsic functions for exploiting the SIMD extension, the code quality of the generated compiler is shown. Note that in this experiment the inline assembly cannot be an alternative means to exploit the SIMD extension because the SIMD extension includes additional registers which cannot be addressed on the base compiler without any modification. Adding the plugins to the base compiler allows the compiler to handle the SIMD extension.

Figure 4.9 and Table 4.1 show the target processor's block diagram and the summary of its SIMD extension. The SIMD extension includes 32 64-bit registers, an SIMD instruction set, and an SIMD functional unit which addresses 64-bit wide packed data. Available data types are 16 bits $\times$ 4, 32 bits $\times$ 2, and 64 bits $\times$ 1. The SIMD instruction set includes logical operations, data interleaving operations, data type conversions load/store operations, and packed arithmetic operations such as addition, subtraction, multiplication, and comparison.

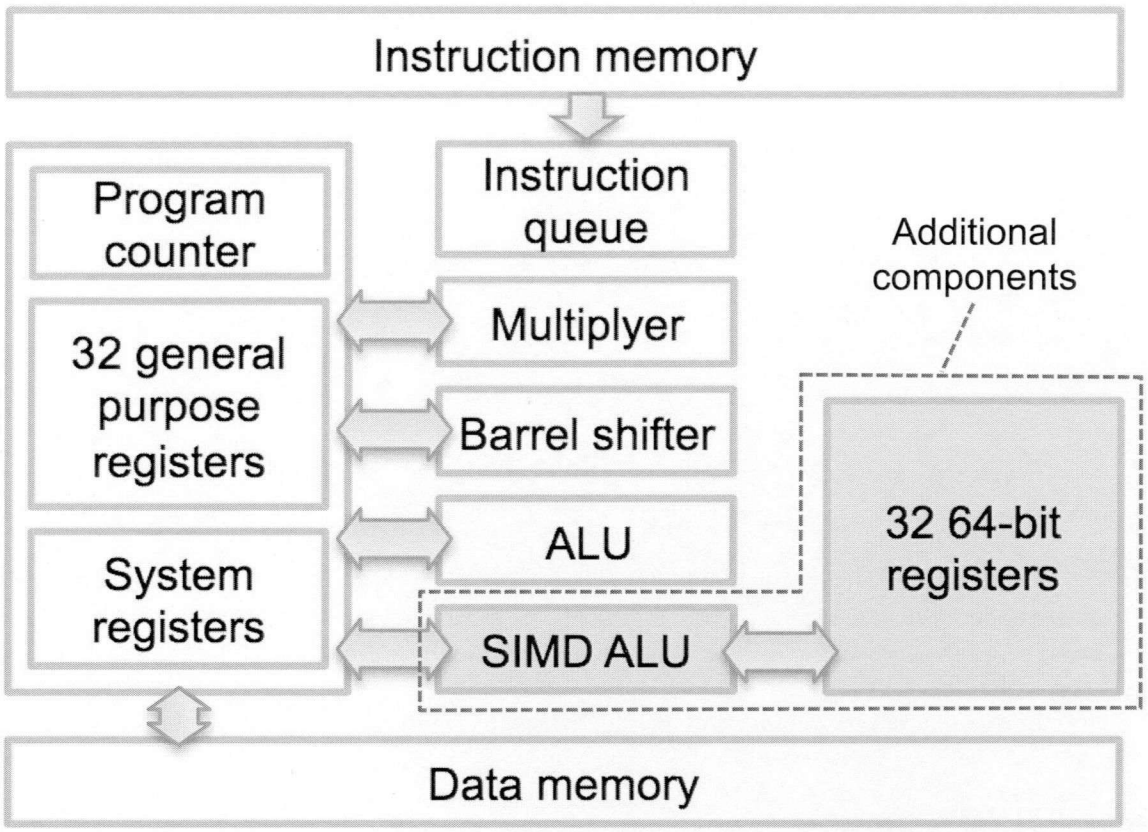


Figure 4.9: Block diagram of the V850 processor with SIMD extension.

4.4.1 Generated toolchain for SIMD extension

The specification of the SIMD extension was translated into an XML document and instruction behavior description, which were input files to the proposed tool generator. Then, the tool generator generated a toolchain with the plugins in it for the target processor. Table 4.2 shows the code amount of input files and the generated plugins. The number of SIMD instructions added to the V850 microcontroller is 179, and the input files to the tool generator have 10370 lines in total (XML: 5934, *.c: 4436). Although the code amount of the input files is not so small compared with the output files, which have 30098 lines in total, generating several tools such as a compiler, assembler, and simulator from simple specification documents is very beneficial.

4.4.2 Code generation using intrinsic functions

In order to investigate the compiler efficiency in terms of code generation using intrinsic functions, comparison between hand-optimized assembly codes and compiler-generated codes is discussed.

Table 4.1: Architecture summary of the V850 microcontroller with SIMD extension.

Base architecture	RISC processor for embedded systems. Harvard architecture. Single cycle instruction execution. Compact code size allowed by 2-byte instructions. 32 32-bit general purpose registers.
SIMD extension	32 64-bit registers Data types: 16 bits × 4, 32 bits × 2, 64 bits × 1. Packed arithmetic instructions. Load/store instructions. Data type conversion. Logical operation.

Table 4.2: The code amount of the generated plugins for the SIMD extension.

Base tools	GNU Binutils 2.17 GDB 6.6 GCC 3.4.6
Number of instructions	179
Number of lines of the input XML file	5934
Number of lines of the input behavior description	4436
Number of lines of the assembler plugin	8903
Number of lines of the simulator plugin	7699
Number of lines of the compiler code fragments	13496

For a number of basic signal processing functions such as filtering, sorting, and FFT, two kinds of programs were developed: (1) hand-optimized assembly codes using the SIMD extension, and (2) compiler-generated codes using intrinsic functions for the SIMD extension. Then the programs were profiled in terms of the number of executed instructions by using the generated simulator.

In order to build program (2), a new C program was developed using intrinsic functions from scratch for each of the signal processing functions. Since the programs of the signal processing functions written in normal C language have fewer lines than 100 lines, writing a new C program using intrinsic functions took a day or less per one signal processing function. However, for sorting and FFT which have more lines and require algorithm tuning suited to SIMD instructions, writing programs using intrinsic functions for them took a month per each. Table 4.4 shows lines of source codes: (a) original C code, (b) C code using intrinsic functions, (c) assembly code generated from (b), and (d) hand-optimized assembly code. Regarding all signal processing functions in Table 4.4, code (b) using intrinsic functions has more lines than code (a) written in normal C language.

As an example, an FIR filtering function written in normal C language is shown in Figure 4.12 (a), its variant using intrinsic functions is shown in Figure 4.12 (b), an assembly code generated from Figure 4.12 (b) is shown in Figure 4.12 (c), and a hand-optimized assembly code is shown in Figure 4.12 (d). In Figure 4.12 (b) and (c), intrinsic functions such as `_vxor()` and `_vld_dw_inc()` are translated into their corresponding assembly instructions such as `vxor` and `vld.dw`, respectively.

Figure 4.10 shows the increase in the number of executed instructions of the compiler-generated codes against the hand-optimized assembly codes. The number of executed instructions increases by only 7% on average when using intrinsic functions. Since compiler-generated codes without using intrinsic functions for the SIMD extension could increase to 900% on average against the hand-optimized assembly codes, only 7% increase in the executed instructions is acceptable. Furthermore, more than half the applications fall below the average of 107% in the ratio of the numbers of executed instructions. By using intrinsic functions, the compiler will replace variables with actual registers, ensuring better allocation, which is a bothersome work for programmers. In addition, the compiler-generated codes using intrinsic functions are nearly as good as the hand-optimized assembly codes. The proposed tool generation method made it possible to generate the compiler with such useful intrinsic functions.

The 7% increase could be caused by the following reasons:

Range checking before loops Compiler-generated codes have range checking before loops whether the condition to begin and continue the loops is met or not. Hand-optimized codes do not usually have such range checking because programmers know whether the range checking is unnecessary in their programs or not.

Redundant register transfers Compiler-generated codes have redundant register transfers on arguments and a return value of functions. Arguments given to functions are copied to local variables, and a local variable having a return value is also copied to a register at the end of functions. These register transfers sometimes may not be optimized and they remain as redundant move instructions.

A similar comparison done here was reported in the application note [44] for IDCT on Pentium4. The application note compared elapsed time for two types of IDCT implementation on

Pentium4 with the SSE2 instruction set: one is written in C language using intrinsic functions and the other written manually in assembly language. The increase of the elapsed time when using intrinsic functions was 9% over that of the hand-optimized assembly code. This is a practical example of how to improve IDCT on Pentium4 using intrinsic functions instead of assembly language. Therefore, 7% increase on average shown in Figure 4.10 is acceptable overhead and enough efficient against hand-optimized assembler.

Figure 4.11 shows the increase in the code size of the compiler-generated codes against the hand-optimized assembly codes. The code size decreases by 2% on average when using intrinsic functions while more than half the applications exceed 100% in the ratio of the code size shown. Although there exists a large variance of the ratio of the code size from 77% to 115% in Figure 4.11, it can be observed that the compiler-generated codes using intrinsic functions are nearly as good as the hand-optimized assembly codes in terms of the code size too.

4.5 Related work and discussion

Here, the proposed approach is compared with several related work for retargeting GNU Binutils and discuss how they differs. There have been a lot of works for generating tools from processor architecture description language. Table 4.3 shows a comparison among several conventional methods using the GNU toolchain and the proposed approach.

Tensilica developed a configurable processor core, called Xtensa, in the late 1990s and provided a tool generator for the Xtensa [45]. Although the target processor architecture of the tool generator is limited to the Xtensa, it can generate ports of both GNU Binutils and GNU Debugger, simulators, and compilers for customized Xtensa cores. Unfortunately, according to the reference manual [45], there is not any description on the capability of adding extra relocation types used for extra instructions.

CGEN [35], which was released as open-source software from Red Hat in 2000, is a tool to generate code fragments for assemblers, disassemblers, and simulators. The generated codes can be embedded into GNU Binutils. Only for the MeP processor, CGEN has a feature to add intrinsic functions to the GCC.

Abbaspour presented in 2002 a systematic approach to retarget GNU Binutils [36]. An experimental result to generate GNU Binutils was reported in [36] for the SPARC architecture. In those years, the development of the ArchC started, which is an open-source binary utility generator [38]. Baldassin reported in [38] that the ArchC can retarget GNU Binutils and generate simulators for several processor architectures including the i8051 processor, for which there was no reference ports in the original GNU Binutils.

On the other hand, the proposed method can generate a set of the GNU toolchain including a compiler for a target processor. Regarding the kinds of tools that can be generated, i.e., compiler, assembler, simulator, etc., the proposed method has the same capability as the tool generator for the Xtensa. However, there is a difference between the tool generator for the Xtensa and the proposed method, in which the main scope of the proposed method is to add new instructions to existing base processors but not to modify the ISA of a base processor. While the Xtensa's tool generator can be used only for the Xtensa, the proposed tool generator can be used for the existing processors

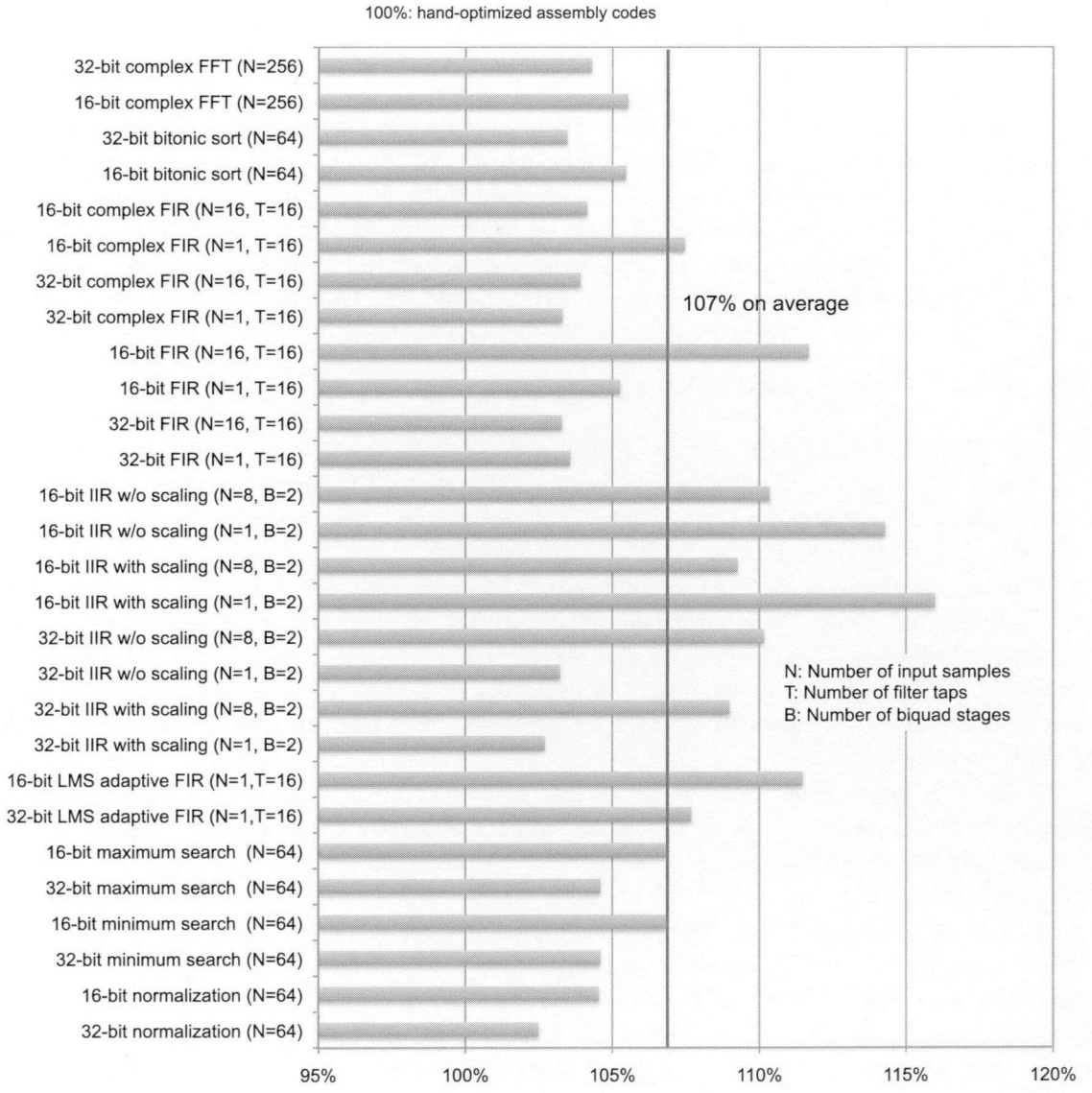


Figure 4.10: Increase in the number of executed instructions of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes.

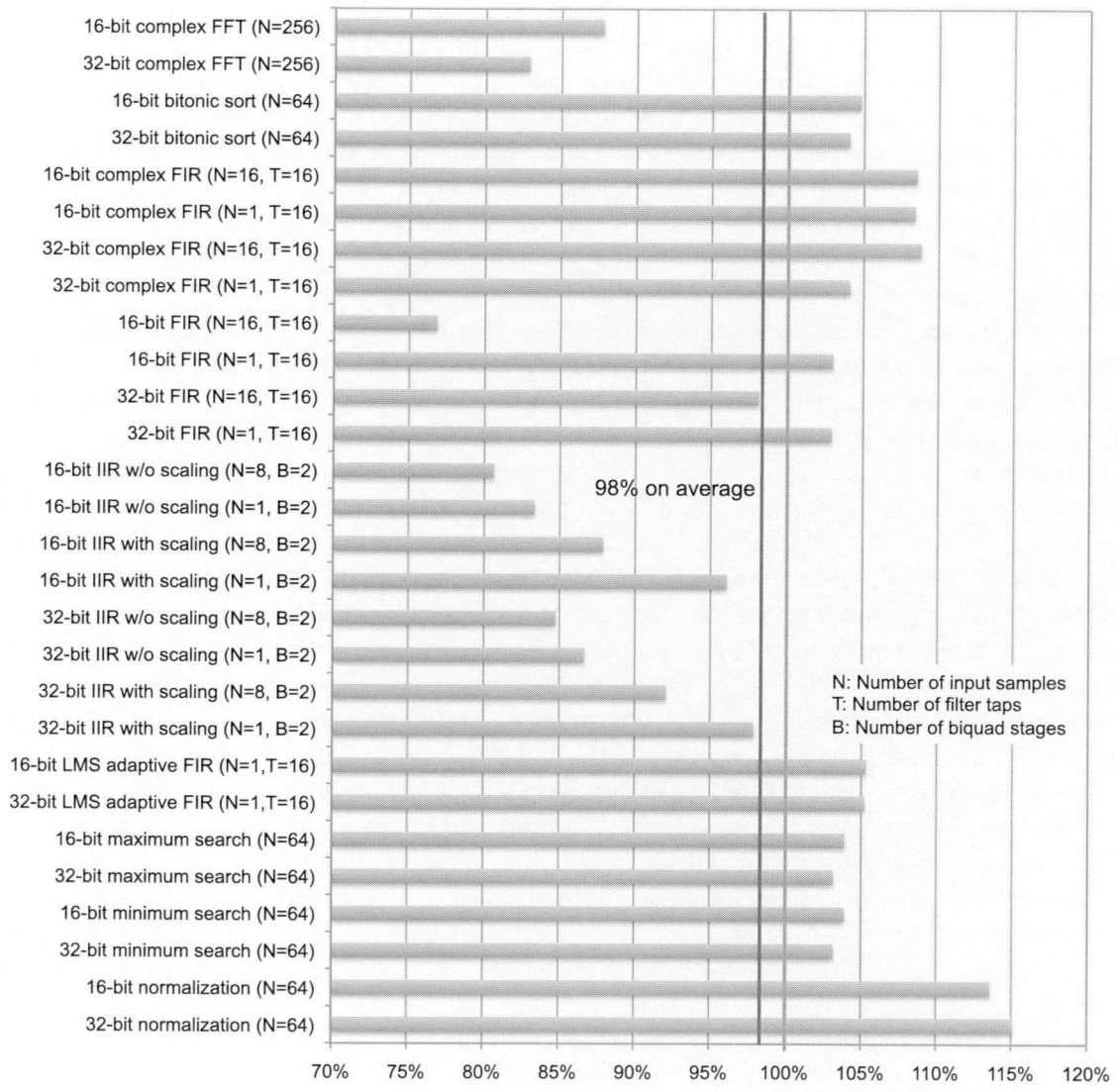


Figure 4.11: Increase in the code size of compiler-generated codes using intrinsic functions against that of hand-optimized assembly codes.

that have ports of the GNU toolchain. There is also a difference in instruction syntaxes. The conventional methods (a)-(d) in Table 4.3 has a rule that instructions be written in a syntax of single mnemonic plus multiple operands. The proposed tool generator, however, generates the assembler that can handle not only a mnemonic style but also a function style and an algebraic style.

In terms of the amount of handwork necessary for generating toolchain, the proposed method and conventional methods require similar amount of handwork such as describing instruction specification. Although XML documents for the proposed method are tend to have more lines than conventional methods, it does not have a big impact on the amount of handwork.

Regarding the time needed to generate toolchain, there is no big difference among the proposed method and conventional methods. The proposed method takes a couple of seconds to generate plugins, and takes several tens of minutes to build the toolchain involving the plugins. The time needed to build toolchain on the proposed method is the same as those of conventional methods such as ArchC and rbinutils.

On the proposed method, programmers need source modification using intrinsic functions in order to exploit new instructions and to make applications faster. This is a negative point compared with auto-customization frameworks of ISAs reported in references [40–42]. While rewriting source codes for more speed is still a practical method, exploiting new instructions without source modification on the proposed method is one of the future work.

The proposed approach is not only dedicated to the V850, but it can also be applied to other processors. In fact, the proposed method can generate toolchains for processors with a simple instruction extension based on the ARM and MIPS processors although these trials based on the ARM and MIPS processors are very simple and not mentioned in details in this chapter.

Other than the experiment shown in this chapter, two more experiments using the proposed approach are reported in [46,47]. The experiment reported in [46] shows that the proposed method can be used in the performance evaluation of processor architecture enhancement for variable length decoding. Another experiment reported in [47] shows that the proposed method is applicable to retargeting GNU Binutils and GNU GDB based on simple architecture specifications for newly developed processors ranging from a tiny 16-bit processor having 48 instructions to an in-house 32-bit processor having 461 instructions.

In addition, the proposed approach is applicable to toolchains other than the GNU toolchain since most of templates for generating plugins such as instruction parsing/encoding/decoding/disassembling plugins are independent from the GNU toolchain. Although templates and instruction patterns for adding intrinsic functions to compilers is dependent on the GCC, the underlying ideas of the proposed approach is able to be used in other toolchains.

Table 4.3: Comparison among tool generation methods using the GNU toolchain.
(a) The Xtensa tool generator by Tensilica [45]
(b) CGEN: Cpu generator by Red Hat [35]
(c) rbinutils by Abbaspour [36]
(d) ArchC by Baldassin [38]
(e) The proposed method

Target base-processor architectures	
(a)	Xtensa configurable cores
(b),(c)	RISC CPUs
(d)	RISC/CISC CPUs
(e)	Existing CPUs which have ports of the GNU toolchain
Architecture description language (ADL)	
(a)	TIE (Tensilica Instruction Extension) language
(b)	Lisp-like language
(c)	Simple language
(d)	Simple C-like language
(e)	XML-based language
How to generate GNU Binutils	
(a)	Base binutils + dynamic link libraries (DLLs)
(b)	Only opcode library
(c),(d)	Templates + code fragments
(e)	Base binutils + Templates + plugins + code fragments
How to define relocation types	
(a),(b)	N/A
(c)	Explicit definitions written in instruction set description
(d),(e)	Implicit definitions extracted from instruction set description
How to generate simulators	
(a)	Base simulator + DLLs
(b)	Only libraries to execute/decode instructions
(c)	N/A
(d)	Templates + code fragments
(e)	Base simulator + templates + plugins + code fragments
How to generate compilers	
(a)	Base compiler + DLLs + header files for intrinsics
(b),(c),(d)	N/A
(e)	Base compiler + templates + code fragments + header files for intrinsics and emulation libraries

4.6 Summary

A new method to generate software development tools has been proposed for instruction set extension of existing embedded processors. The proposed approach generates a toolchain (assembler, disassembler, linker, simulator, and compiler) for a target processor, which is based on an existing processor and which has additional instructions and registers, by adding software components as plugins to the base processor's toolchain to handle additional instructions and registers. This chapter demonstrated that the approach worked effectively through an experiment based on the V850 microcontroller. As shown in this chapter, by using intrinsic functions, the generated compiler could give as good performance as that of hand-optimized assembly codes.

Table 4.4: Lines of source codes: (a) original C code, (b) modified C code using intrinsic functions, (c) assembly code generated from (b), and (d) hand-optimized assembly code.

Function	(a)	(b)	(c)	(d)
32-bit normalization (N=64)	30	55	38	32
16-bit normalization (N=64)	30	57	41	35
32-bit minimum search (N=64)	20	66	41	34
16-bit minimum search (N=64)	20	79	49	41
32-bit maximum search (N=64)	20	66	41	34
16-bit maximum search (N=64)	20	79	49	41
32-bit LMS adaptive FIR (N=1,T=16)	25	104	71	63
16-bit LMS adaptive FIR (N=1,T=16)	18	102	70	62
32-bit IIR with scaling (N=1, B=2)	30	50	29	26
32-bit IIR with scaling (N=8, B=2)	33	56	38	30
32-bit IIR w/o scaling (N=1, B=2)	29	43	28	23
32-bit IIR w/o scaling (N=8, B=2)	33	48	34	28
16-bit IIR with scaling (N=1, B=2)	31	47	32	26
16-bit IIR with scaling (N=8, B=2)	41	59	39	31
16-bit IIR w/o scaling (N=1, B=2)	30	46	29	22
16-bit IIR w/o scaling (N=8, B=2)	42	62	35	28
32-bit FIR (N=1, T=16)	22	62	45	40
32-bit FIR (N=16, T=16)	33	99	71	61
16-bit FIR (N=1, T=16)	17	56	44	39
16-bit FIR (N=16, T=16)	33	80	60	62
32-bit complex FIR (N=1, T=16)	34	69	46	41
32-bit complex FIR (N=16, T=16)	41	82	56	45
16-bit complex FIR (N=1, T=16)	26	81	55	47
16-bit complex FIR (N=16, T=16)	33	77	55	41
32-bit bitonic sort (N=64)	127	224	215	197
16-bit bitonic sort (N=64)	127	188	156	141
32-bit complex FFT (N=256)	330	320	242	274
16-bit complex FFT (N=256)	320	338	274	292

N: Number of input samples, T: Number of filter taps, B: Number of biquad stages

```
 1: int32_t
 2: fir16_single (
 3:  int16_t *sig,
 4:  int16_t *coef,
 5:  int T,
 6:  int offset )
 7: {
 8: int i;
 9: int32_t y = 0;
10: for ( i=0 ; i<T ; i++ )
11: {
12:  int idx=i+offset;
13:  if (T<=idx) idx-=T;
14:  y+=coef[i]*sig[idx];
15: }
16: return y;
17: }
```

(a) C code written in normal C language

```
 1: int32_t
 2: fir16_single_simd_intrinsic (
 3:   int16_t *sig, int16_t *coef,
 4:   int T, int offset )
 5: {
 6: int i;
 7: int64_t w_3210, w_7654, w_ba98, w_fedc;
 8: int64_t x_3210, x_7654, x_ba98, x_fedc, acc;
 9: int32_t step_and_offset
10:   = (8<<16) | (2*(offset - (3 & offset)));
11: int32_t buff_size = 2*T - 1;
12: int64_t mod_param
13:   = _deposit_64(buff_size, step_and_offset);
14: int32_t align_byte = 2 * (3 & offset);
15: short   loop_cnt = T >> 4;
16: acc = _vxor(acc,acc);
17: /* load coefficients */
18: _vld_dw_inc(w_3210, coef);
19: _vld_dw_inc(w_7654, coef);
20: /* load input signals */
21: _vld_dw_mod(x_3210, mod_param, sig);
22: _vld_dw_mod(x_7654, mod_param, sig);
23: _vld_dw_mod(x_ba98, mod_param, sig);
24: _vld_dw_mod(x_fedc, mod_param, sig);
25: for ( i = 0 ; i < loop_cnt ; i++ )
26: { /* multiply and accumulate */
27:  int64_t xx;
28:  xx = _vconcat_b(align_byte, x_3210, x_7654);
29:  _vld_dw_inc(w_ba98, coef);
30:  acc = _vmsumad_h(acc, w_3210, xx);
31:  _vld_dw_mod(x_3210, mod_param, sig);
32:  xx = _vconcat_b(align_byte, x_7654, x_ba98);
33:  _vld_dw_inc(w_fedc, coef);
34:  acc = _vmsumad_h(acc, w_7654, xx);
35:  _vld_dw_mod(x_7654, mod_param, sig);
36:  xx = _vconcat_b(align_byte, x_ba98, x_fedc);
37:  _vld_dw_inc(w_3210, coef);
38:  acc = _vmsumad_h(acc, w_ba98, xx);
39:  _vld_dw_mod(x_ba98, mod_param, sig);
40:  xx = _vconcat_b(align_byte, x_fedc, x_3210);
41:  _vld_dw_inc(w_7654, coef);
42:  acc = _vmsumad_h(acc, w_fedc, xx);
43:  _vld_dw_mod(x_fedc, mod_param, sig);
44: }
45: return _extract_lo_64(acc);;
46: }
```

(b) C code using intrinsic functions

Figure 4.12: FIR filtering functions: C language.

```
 1: _fir16_single_simd_intrinsic:
 2:     mov -4,r10
 3:     and r9,r10
 4:     add r10,r10
 5:     mov r8,r11
 6:     add r8,r11
 7:     movhi hi0(524288),r0,r12
 8:     or r10,r12
 9:     addi -1,r11,r13
10:     andi 3,r9,r9
11:     add r9,r9
12:     sar 4,r8
13:     sxh r8
14:     vxor vr3, vr3, vr3
15:     vld.dw [r7]+, vr9
16:     vld.dw [r7]+, vr8
17:     vld.dw [r6]%, r12, vr7
18:     vld.dw [r6]%, r12, vr6
19:     vld.dw [r6]%, r12, vr5
20:     vld.dw [r6]%, r12, vr4
21:     cmp 0,r8
22:     ble .L7
23: .L8:
24:     vconcat.b r9, vr7, vr6, vr2
25:     vld.dw [r7]+, vr0
26:     vmsumad.h vr9, vr2, vr3
27:     vld.dw [r6]%, r12, vr7
28:     vconcat.b r9, vr6, vr5, vr2
29:     vld.dw [r7]+, vr1
30:     vmsumad.h vr8, vr2, vr3
31:     vld.dw [r6]%, r12, vr6
32:     vconcat.b r9, vr5, vr4, vr2
33:     vld.dw [r7]+, vr9
34:     vmsumad.h vr0, vr2, vr3
35:     vld.dw [r6]%, r12, vr5
36:     vconcat.b r9, vr4, vr7, vr2
37:     vld.dw [r7]+, vr8
38:     vmsumad.h vr1, vr2, vr3
39:     vld.dw [r6]%, r12, vr4
40:     loop r8, .L8
41: .L7:
42:     mov.dw vr3, r10
43:     jmp [r31]
```

(c) assembly code generated from (b)

```
 1: _asm_fir16_single_simd:
 2: mov     r0, r10
 3: andi    3, r9, r15
 4: mov     r15, r14
 5: shl     1, r14
 6: sub     r15, r9
 7: add     r9, r9
 8: mov     r8, r11
 9: sar     4, r11
10: movhi   8, r9, r12
11: mov     r8, r13
12: add     r8, r13
13: movea   -1, r13, r13
14: vld.dw    [r7]+, vr10
15: vld.dw    [r7]+, vr11
16: vxor      vr4, vr4, vr4
17: vld.dw    [r6]%, r12, vr6
18: vld.dw    [r6]%, r12, vr7
19: vld.dw    [r6]%, r12, vr8
20: vld.dw    [r6]%, r12, vr9
21: .L1:
22: vconcat.b r14, vr6, vr7, vr14
23: vld.dw    [r7]+, vr12
24: vmsumad.h vr14, vr10, vr4
25: vld.dw    [r6]%, r12, vr6
26: vconcat.b r14, vr7, vr8, vr14
27: vld.dw    [r7]+, vr13
28: vmsumad.h vr14, vr11, vr4
29: vld.dw    [r6]%, r12, vr7
30: vconcat.b r14, vr8, vr9, vr14
31: vld.dw    [r7]+, vr10
32: vmsumad.h vr14, vr12, vr4
33: vld.dw    [r6]%, r12, vr8
34: vconcat.b r14, vr9, vr6, vr14
35: vld.dw    [r7]+, vr11
36: vmsumad.h vr14, vr13, vr4
37: vld.dw    [r6]%, r12, vr9
38: loop      r11, .L1
39: .L2:
40: mov.w     0, vr4, r10
41: jmp       [lp]
```

(d) hand-optimized assembly code

Figure 4.13: FIR filtering functions: assembly language.

Chapter 5

Productivity improvement on parallelization of digital signal processing algorithm for multiple processors

This chapter addresses software parallelization based on schematic models of digital signal processing algorithms, and proposes a method to generate parallel C codes suited to pipeline processing from the models developed on the Simulink which is a model-based development tool. The Simulink are widely used in the field of control systems for ranging from algorithm development to code generation for embedded systems. Although there are several researches which focus on parallelization based on Simulink models, they exploit parallelism mainly within one step processing of the models, or among multiple-step processing by ignoring inter-step data dependencies. Here, one step processing means that a model processes an input signal and calculate an output signal. In order to exploit more parallelism among multiple-step processing while preserving the original semantics of the model, this chapter focuses on a pipeline processing based on a way of applying the theory of communicating sequential processes (CSP). Under the parallelization process, the proposed method eliminates loop structures in models and builds directed acyclic graphs (DAGs) suited to a pipeline processing. While data items are transferred through communication on the CSP, they are stored and shared in double buffers on the proposed method. On the experiment of applying the method for an audio processing model, the execution time of the parallelized code could be reduced successfully to 26.3% on a 4-core processor running at 400MHz with a symmetric multi-processing real-time operating system, compared with that of the sequential code.

5.1 Multicore processors and software parallelization

Multicore processors have been becoming popular to increase their performance in the field of PCs and embedded systems [48–50]. For example, communication infrastructures such as mobile base stations work on heterogeneous and/or homogeneous multicore processors to handle radio signals

and packets of many users [51]. Furthermore, multicore processors are also used for encoding, decoding, and image processing on high definition televisions [52–54], and are evaluated for car navigation systems [55, 56]. Behind this trend focused on multicore, there is a story that multicore is becoming one of key technologies to increase processors' performance since increasing processors' operating frequency is more and more difficult and requires much power consumption. This movement toward multicore is remarkable in the field of communication infrastructures which require higher processing performance than in other fields.

To exploit the inherent performance of multicore processors, it is very important to parallelize software working on them. However, parallelization makes it difficult to develop software since parallelization requires adequate workload balancing and access controlling of shared resources [57]. To ease parallelizing software, so far there have been many researches of parallel languages, frameworks, and compilers, etc. [6, 56, 58]. Recent examples are GPU-oriented frameworks such as CUDA [59] and OpenCL [60] to exploit data parallelism in which same calculations are performed for many pixels, and the Intel C/C++ compiler that has features such as automatic loop parallelization and automatic vectorization.

To take advantage of task parallelism, which is another aspect for parallelism, parallelization methods based on dataflow and pipeline processing have also been actively researched [61–63]. For example, StreamIt [61] is a research project on a source-to-source compiler for stream processing, which handles continuous data streams such as audio and video signals. Programmers describe pipelines in the language of the StreamIt for dataflow of stream processing, then the StreamIt compiler generates source codes that make every stage of the pipelines work in parallel. For another example, Molatomium [63] is a research project that also focuses on dataflow.

While these novel frameworks based on dataflow and pipeline processing are very useful for parallelizing software developed from scratch, they have a problem in terms of translation from conventional languages. To cure this problem, this chapter focus on the Simulink, a model development tool, which is widely used in a model-based development method spreading recently in the field of control systems. Models developed on the Simulink are widely used in the field of control systems for raging from algorithm development to code generation for embedded systems. Simulink models are just dataflow graphs themselves, and are suited to parallelization since they represent structural parallelism visually in them. If parallel software can be generated from existing models developed on the popular Simulink, language translation by hand should never be needed to make software parallel.

There are several activities by the MathWorks and others in terms of Simulink models and parallelization [64, 65]. The MathWorks, a developer of the Simulink, provides Parallel Computing Toolbox (PCT) as his own product before, which allows programmers to run large-scale models for simulation in parallel. On the latest version of the C code generation tool from the Simulink models, internal processing of individual blocks such as FFT and filters is parallelized. Although there have been several other researches and products for generating parallel code from Simulink models [66–68], programmers on these researches and products have to take charge of making tasks and allocating tasks to CPUs. One of conventional research extracts parallelism only from one-step processing of models [64]. Here, one step processing means that a model processes an input signal and calculate an output signal. Another research extracts parallelism among multiple-step processing by ignoring inter-step data dependencies, which may cause acceptable numerical errors [65]. The first one has an issue of less parallelism, and the second one has an issue of trade-off

between obtained parallelism and numerical error.

This chapter proposes a method to generate parallel C code from the Simulink models. The aim of the method is to extract parallelism from multiple-step processing without ignoring inter-step dependencies. Specifically, the method transforms a model having feedback loops to a directed acyclic graph (DAG) while preserving the original semantics of the model, and executes every node of the DAG concurrently on the basis of the theory of communicating sequential processes (CSP) [69, 70]. While the theory of CSP transfers data items through communication, which may cause data copies from buffer to buffer, the proposed method shares data items by using double buffers placed at shared memories. The contribution of this work is the CSP-based method using double buffering in combination with the loop structure decomposition for pipeline processing, and the evaluation of the proposed method. In the rest of this chapter, the proposed method is described, and then parallelization experiment is explained. Finally, this chapter shows that as a result of parallelizing an audio equalizer model on a 4-core processor running at 400MHz the proposed method reduces execution time down to 26.3% through parallelization.

5.2 Related work

There exists several approaches aiming at mapping an application onto processors including multiple cores in the most efficient way. The references [71–73] present a large overview of the different methodologies. Here several approaches are picked up.

Ptolemy [74] focuses on component-based heterogeneous modeling and allows to combine hierarchically different models of computations at high level of abstraction in a Simulink fashion. It uses tokens as the underlying communication mechanism. Controllers regulate how actors fire and how tokens are sent between each actors. This mechanism allows different models of computation to be combined within the Ptolemy framework.

PeaCE [75] specifies the system behavior with a heterogeneous composition of several models of computation. The PeaCE framework provides seamless co-design flow from functional simulation to system synthesis, utilizing the features of the formal models maximally during the whole design process. This framework is based on the Ptolemy project [76]. When dealing with C/C++ specifications, the PeaCE approach does not propose an automatic procedure to transform this specification into dataflow graphs. This step is manual and an example of this transformation on a MPEG-4 decoder is given in [77].

Daedalus [78] is a framework for system-level architecture exploration, high-level synthesis, programming, and prototyping of MPSoC architectures. This framework is based on the theory of Kahn Process Network (KPN) [79]. KPN is well suited for signal processing systems. The design flow of this framework is fully automated and comprises several tools. The KPNgen tool, which is one of tools in the framework, converts sequential applications written in C/C++ into KPNs. Based on the KPNs, sequential applications are mapped onto processor cores. Some modifications in the C/C++ specification as an input are sometimes needed in case the specification does not meet requirements of the KPNgen tool.

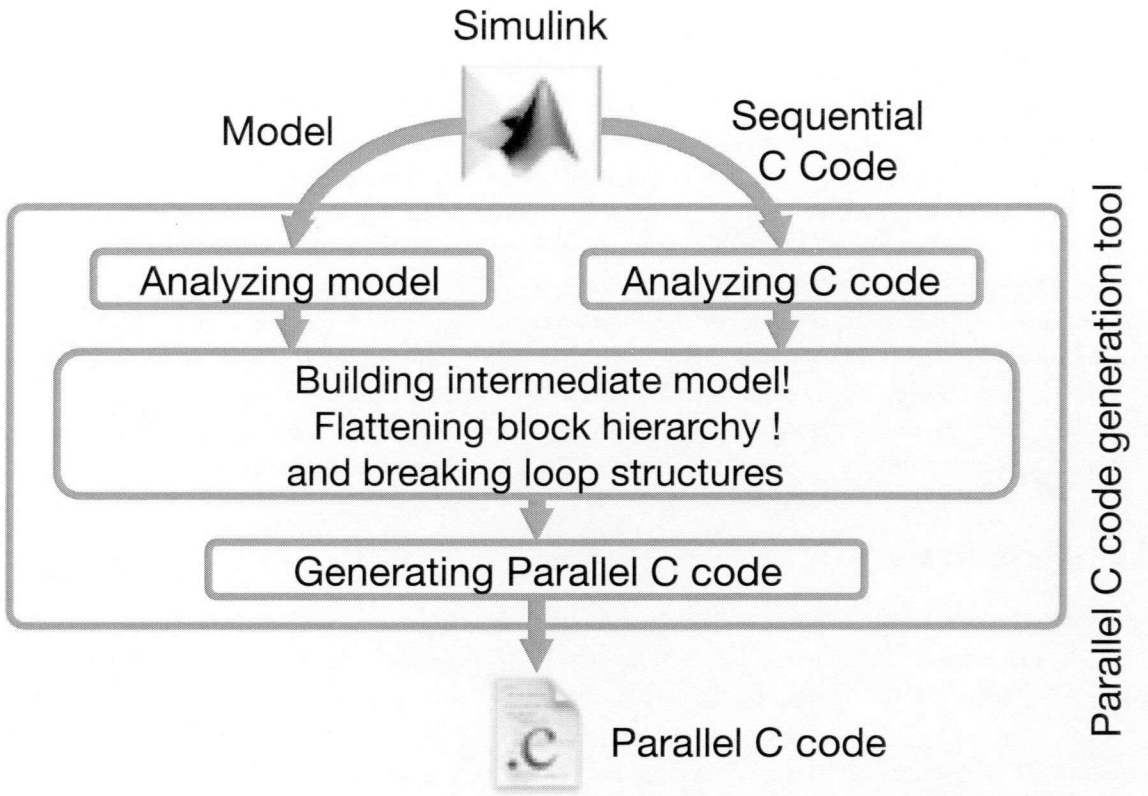


Figure 5.1: The process to generate parallel C code from a Simulink model.

5.3 Parallel C code generation from Simulink models

This chapter proposes a method to generate parallel C code from Simulink models on the basis of the theory of CSP. The Simulink is a model-based design tool, which uses a block diagram notation to represent mathematical operations of dynamic systems. Models designed by the Simulink includes blocks and lines, and blocks may include another blocks and lines hierarchically. The proposed method regards blocks in Simulink models as tasks and lines between blocks in Simulink models as communication channels. Figure 5.1 shows a process to generate parallel C code from Simulink models. Underling concepts of the proposed method are described below.

Mapping a block in a model to a task: The proposed method regards the processing of a block in a Simulink model as a task, and a dataflow graph represented with blocks and lines of a Simulink model as a data dependency graph of tasks, respectively. Then behavior of each task is retrieved from the sequential C code generated from a Simulink model by the Real-Time Workshop, which is a Simulink component.

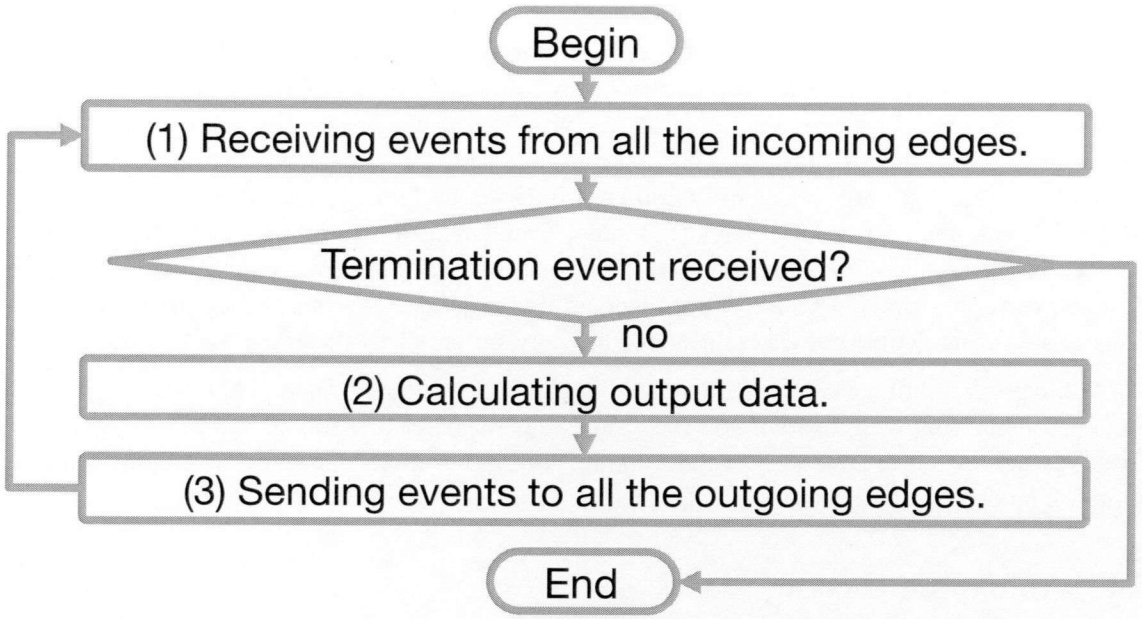


Figure 5.2: The flow chart of task execution.

Signaling completion of tasks with synchronized task communication: In the theory of CSP, processes running concurrently communicate via synchronized message passing. The proposed method make the tasks run concurrently while communicating based on the data dependency graph extracted from a Simulink model. The proposed method uses synchronized task communication as a means to notify completion of tasks to each other. An event of completion of task calculation is transmitted to other tasks via synchronized task communication, and the tasks that receive the event begin their own calculation. During synchronized task communication, tasks wait until the communication partners become ready. Figure 5.2 shows the task execution flow. Each task repeats the following steps:

(1) receiving events to know completion of ascending tasks that feed input data to this task,

(2) calculating output data, and

(3) sending events to notify that output data is ready to descending tasks that uses the output data.

Pipeline parallel processing using double buffers for task output data: On the proposed method, tasks obtain the calculation results of other tasks from shared memories but not from task communication. Although the theory of CSP transfers the calculation results of tasks through task communication, task communication is not adequate for large data transfer on multicore processors having shared memory since it is necessary to transfer data from the buffer storing the calculation results to the buffer for task communication. The data copies from buffer to buffer could be a problem when large data items are transferred through task communication. To avoid the data copies

from buffer to buffer, the proposed method transfers a buffer index via task communication, and shares data items among tasks via double buffers storing calculation results of tasks. The buffer index transferred from task to task represents which buffer to be used. When a large amount of data of a task is fed to multiple descending tasks, the proposed method write the output data to shared memory one time, while the original CSP requires data copy the number of descending tasks. Using double buffers to store task outputs and switching the buffers alternately allow data-producer tasks and data-consumer tasks to work in parallel simultaneously. In this way, costly data copies between buffers can become needless in the proposed method. Since the proposed method targets at multicore processors on PCs and embedded systems and they have shared memories, the cost of data transfer using double buffering on shared memories is smaller than using task communication.

Although Simulink can work with various models, the proposed method handles only models subjected to the following restrictions: fixed time step, discrete-time solver, and single signal rate. These restrictions make analyzing C code simple, and are acceptable ones for developing practical Simulink models.

5.4 Analyzing C code

The Simulink has a feature named as the Real-Time Workshop, which can generate sequential C code corresponding to given Simulink models. Here, the word of sequential means being not parallelized. The proposed method analyzes two files generated from the Real-Time Workshop: *model*.`c` and *model*.`h`. The file of *model*.`c` contains the following three functions:

model`_step()`
: executing a single step of the target model.

model`_initialize()`
: initializing the internal status of the target model.

model`_terminate()`
: terminating the processing of the target model.

In these functions, the function to be parallelized is *model*`_step()`. The proposed method breaks the content of the function of *model*`_step()` into code blocks separated by comment tags that represent which code block comes from which Simulink block. The comment tags are inserted by the Real-Time Workshop for code tracing between a Simulink model and C code, and make it possible to find one-to-one relationship between code blocks and model blocks. In the file of *model*.`c`, all variables and arrays to store output data of model blocks are put together into a single structure. Each of these variables and arrays are read also as input data. The aggregated structure makes it easy to duplicate the structure for double buffering. The header file of *model*.`h` contains input and output data structures of model blocks and a list of subsystem names included in the target model. A subsystem is a group of blocks, and is one level of hierarchy in models.

5.5 Analyzing model

The proposed method reads the model file *model*.`mdl`, analyzes it, and finds its hierarchical structure and connection of blocks. Simulink models have a hierarchy of blocks. A subsystem, which is

a group of blocks, forms one level of hierarchy and can contains other subsystems. In the files of Simulink models, information on block connection is recorded individually for every level of block hierarchy. In order to generate parallel C code, it is necessary to get block connection information between different levels of block hierarchy.

The proposed method finds a model block in a Simulink model corresponding to every code block in the function of *model*_`step()`, and binds them. Since every code block has an ID that represents the name of the corresponding model block, the ID is used during binding. Under the generation of sequential C code from Simulink models, code blocks for the model blocks that do not have any arithmetic operation such as constants, input ports, and output ports are merged into other code blocks in advance. Each of code blocks in sequential C code must correspond to a model block in a Simulink model, although the opposite relationship does not always hold.

5.6 Flattening block hierarchy

The proposed method builds an intermediate model, which becomes a task dependency graph to generate parallel C code. In the building process of an intermediate model, to take advantage of the inherent structural parallelism represented in Simulink models that have hierarchical structures, the proposed method flattens the hierarchical structure of model blocks. Target model blocks for this flattening are subsystem blocks that do not have any binding to code blocks extracted from the sequential C code. After the flattening, input and output port blocks included in such subsystem blocks are connected to external model blocks outside the subsystem blocks as shown in Figure 5.3.

5.7 Breaking loop structures

Several Simulink models have loop structures. Here, a loop structure means a feedback loop but neither a for-loop nor a while-loop. Loop structures prevent the proposed method from determining the execution order of model blocks in the intermediate models, and make parallelization difficult. To cure this problem, the proposed method breaks loop structures that exist in the intermediate models by dividing indirect-feedthrough blocks such as delay element blocks and integral blocks in loop structures while preserving the original semantics of Simulink models. Output data y_n and internal status in an indirect-feedthrough block are calculated as follows:

$$\text{Output:} \quad y_n = f(status_{n-1}) \tag{5.1}$$

$$\text{Update:} \quad status_n = g(x_n) \tag{5.2}$$

In a usual design guideline of Simulink models, there must be indirect-feedthrough blocks somewhere in usual loop structures of Simulink models. Taking advantage of the fact that indirect-feedthrough blocks can calculate output data y_n by using only internal status without using their input data x_n, as shown in Figure 5.4, an indirect-feedthrough block can be divided into two blocks: an output calculation block and a status update block. After this process of breaking loop structures, a DAG in terms of tasks and their data dependency is obtained from the modified intermediate model. The specific steps for dividing an indirect-feedthrough block are as follows.

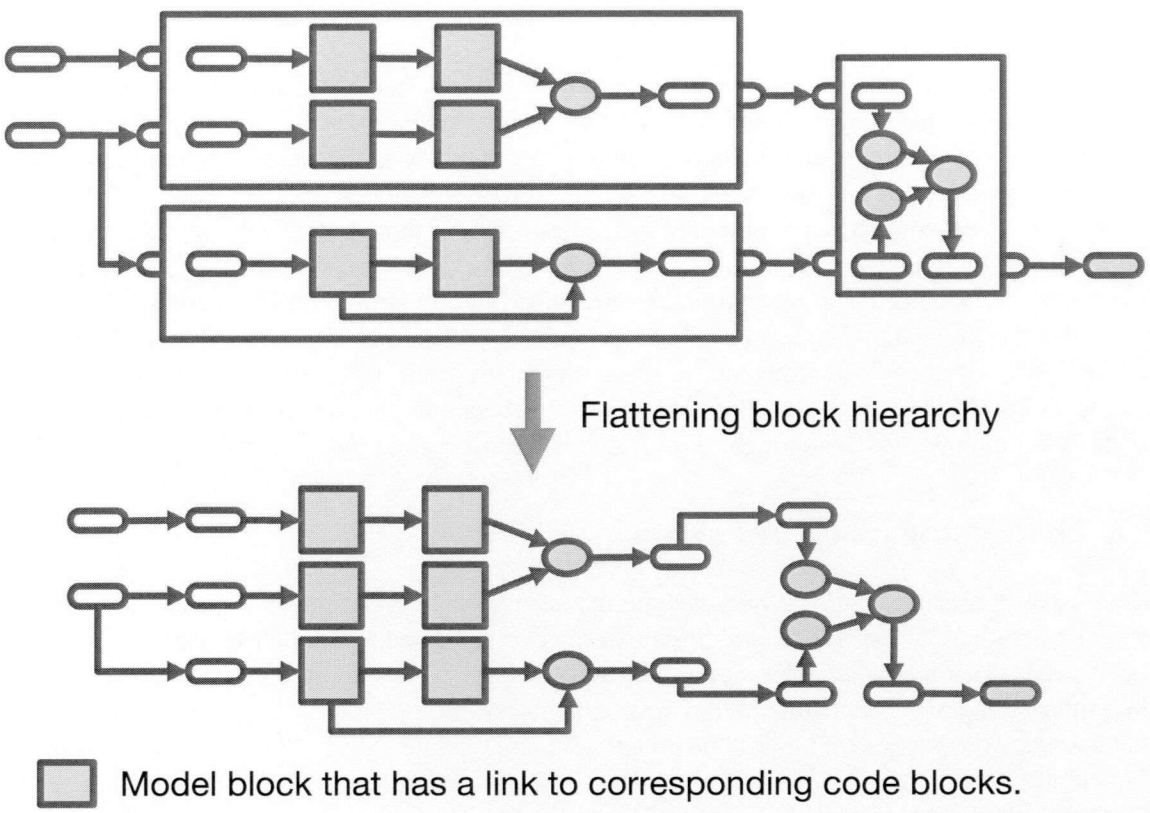


Figure 5.3: Flattening block hierarchy.

(1) Dividing a delay element block into two blocks: The proposed method finds indirect-feedthrough blocks, and divides such a delay element block into an output calculation block and a status update block. The output calculation block is made so as to have a binding with a code block for calculating output data, and the status update block is made so as to have a binding with a code block fro updating internal status. The status update block takes over the incoming edges of the indirect-feedthrough block, and the output calculation block takes over the outgoing edges of the indirect-feedthrough block.

(2) Adding an edge from the start block to an output calculation block: The proposed method creates the start block that starts the processing of the target model, adds an edge from the start block to the output calculation block. The edge added here means that the output calculation block does not have any data dependency from the other model blocks and it can calculate its output data using its own internal status.

(3) Adding an inter-step data dependency edge from a status update block to an output calculation block: Since the internal status updated at the current time step in the status update block will be used in the output calculation block at the future time step, there is a data dependency from

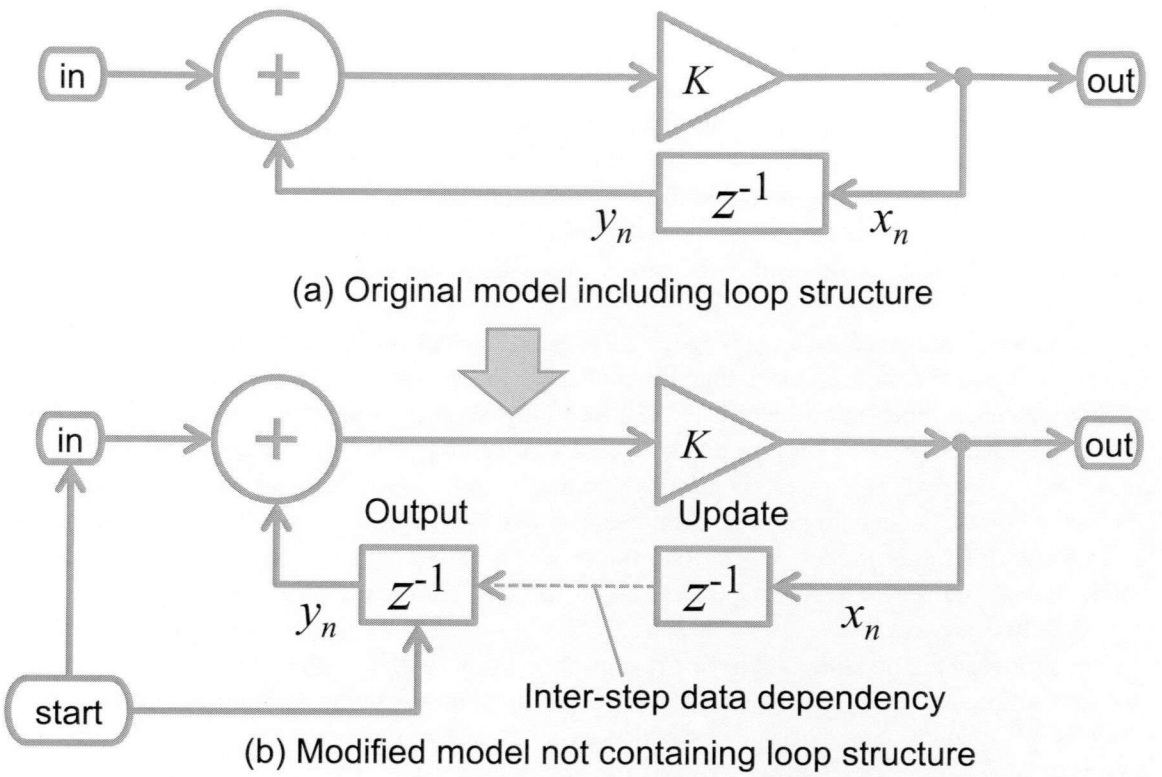


Figure 5.4: How to break a loop structure.

the status update block at time step n to the output calculation block at time step $n + 1$. This dependency is one between different time steps, and this chapter calls such a dependency as an inter-step data dependency. The proposed method adds an edge for this inter-step data dependency from the status update block to the output calculation block. The inter-step data dependency edge is used only for determining if a model block becomes ready for execution at run time but not for determining the execution order of model blocks before run time. The modified intermediate model in this way becomes a directed acyclic graph by ignoring edges of inter-step data dependency.

5.8 Generating parallel C code

After building an intermediate model for the target model, the proposed method generates parallel C code based on the intermediate model. Nodes and edges in the intermediate model are translated to tasks and task communications, respectively, and the generated parallel code includes the following three functions. These functions correspond to ones in the sequential C code generated from the target Simulink model.

model`_step_parallel()`
model`_initialize_parallel()`
model`_terminate_parallel()`

These functions are to be invoked from the task corresponding to the start block. The function of *model*`_initialize_parallel()` starts up all the tasks for the target model. Each task repeats the following steps: (1) receiving events, (2) calculating output data, and (3) sending events. In step (1), a task receives calculation completion events sent from ascending tasks that feed input data to this task. After receiving events from all the ascending tasks, the task moves on to step (2). In step (3), then, the task sends a calculation completion event to each of descending tasks that use the output data of the task. After sending events to all the descending tasks, the task goes back to step (1).

The function of *model*`_step_parallel()` sends a calculation completion event to the tasks corresponding to the model blocks that are connected to the start block, and then returns without waiting for the completion of the tasks. Tasks receiving the event sent from the start task calculate their output data, and send another event to their descending tasks. By invoking the function of *model*`_step_parallel()` any iterations you need, a plurality of iterations of the target model can be run simultaneously in parallel. At the end of iterations you execute, when you want to stop the operation of target model, you invoke the function of *model*`_terminate_parallel()`, which sends a terminate event to running tasks and returns. Tasks receiving a terminate event immediately come to a halt.

The proposed method does not have any rules in terms of (a) task assignment onto CPUs and (b) task scheduling, and those two features are up to the operating systems and/or other tools working with. In this chapter, (a) and (b) are provided by a symmetric multi-processing (SMP) operating system.

5.9 Experiment

To investigate how effective the proposed method works, in this section, experiment results are described for generating parallel C code from two Simulink models [80, 81]. The target environments to execute generated C code are a PC running the Windows operating system and an embedded system running a real time operating system (RTOS). The details of the environments and results of the experiment are shown in Table 5.1.

5.9.1 Audio equalizing

Audio equalizing in the model of [80] is an audio processing that reads audio signals from a file and modifies their waveform in both time and frequency domains. The audio signals are stereo and 16 bits/sample, and their sampling rate is 44.1kHz. Each of signals is represented as a 32-bit floating-point number. The audio equalizing model used in this experiment performs audio processing for a 1024-sample audio frame at a time. The number of blocks included in the model is 252, which includes the number of the subsystems as containers of blocks.

Figure 5.5 shows a task dependency graph extracted from the audio equalizing Simulink model. The number of tasks is 57, which includes the main task corresponding to the beginning block. Extracted tasks perform addition, subtraction, filtering, or FFT for 1024 audio signals. Since the PC used in this experiment has 4 CPUs, the execution time of parallelized programs could be ideally decreased down to 1/4=25%. For the experiment on the Windows PC, the execution time of the

Table 5.1: Experiment results of generating parallel C code from Simulink models.

	Model blocks	Tasks	Execution time ratio compared with sequential programs.		
			PC0	PC1	RTOS
Audio equalizing	252	57	85.8%	38.3%	26.3%
Lane detection	302	64	94.9%	44.3%	39%

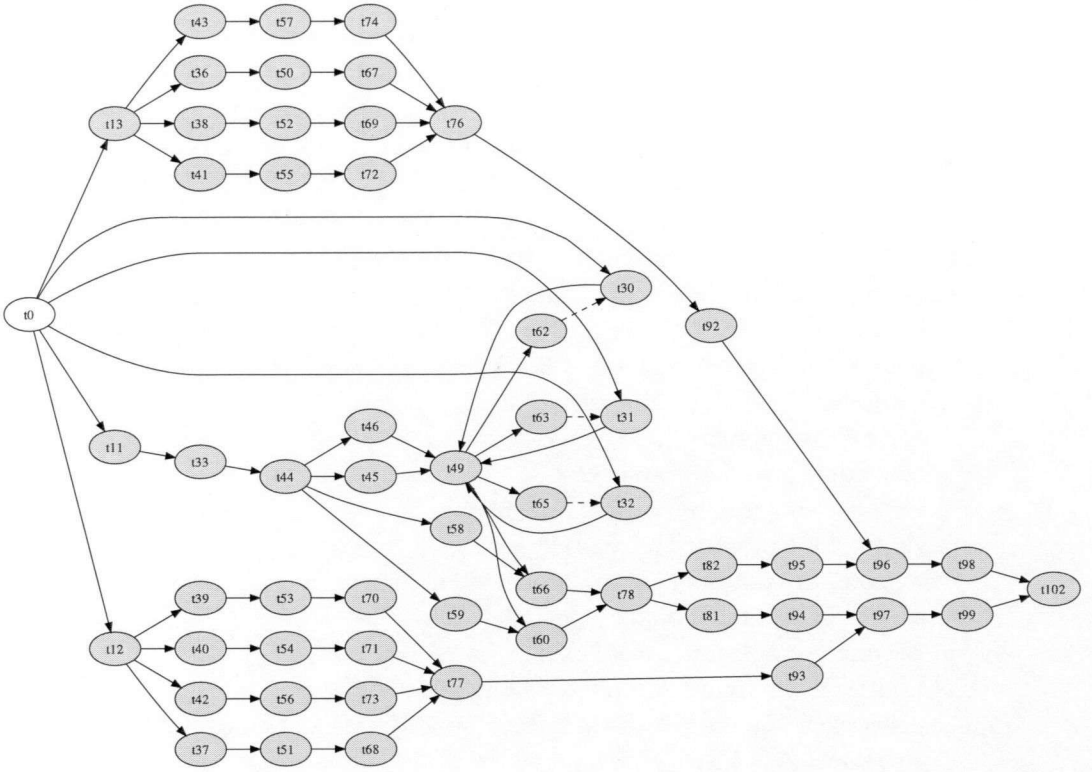
MATLAB/Simulink version: R2010b
Experiment environment: PC0 and **PC1**
OS: Windows Server 2008
processor: Xeon@1.83GHz (4 cores)
compiler: Intel C++ Compiler XE 12.1
compiler options: `/O2 /Qparallel`
task communication: Win32 API
PC0 does not use the proposed method.
PC1 use the proposed method.
∗ Scores of PC0 and PC1 are the execution time ratio against PC0 without option /Qparallel.
Experiment environment: RTOS
OS: eSOL eT-Kernel Multi-Core Edition (SMP) [82]
processor: NaviEngine ARM11 MPCore@400MHz (4 cores) [55]
compiler: ARM RealView Compiler 3.0
compiler options: `-g -O3`
task communication: message buffers (no queues, no time out)

parallelized program using the proposed method is reduced to 38.3% compared with a sequential program before parallelization while the execution time ratio of the program parallelized automatically by the Intel Compiler is 94.9%. For the experiment on the RTOS system, the execution time of the program parallelized by the proposed method is reduced to 26.3%, which is very close to the ideal case.

5.9.2 Lane detection

Lane detection in the model of [81] is an image processing that detects car lanes from roadway images captured by a camera in a car, tracks the detected lanes, and makes alert messages if the car is going to be out of the lanes. The size of the roadway images is 360 x 240 pixels. Hough transformation is used for detecting car lanes, and kalman filtering is used for tracking the detected lanes. Pixels in the image processing of this lane detection are represented as 32-bit floating-point numbers.

Figure 5.6 shows a task dependency graph extracted from the lane detection Simulink model. The number of blocks included in the lane detection model is 302, which includes the number of the



- Node tN represents a task.
- Node t0 is a main task, which corresponds to the beginning block and invokes such functions like *model*`_step_parallel()`.
- Edges between nodes represent data dependencies of tasks.
- Dashed edges between nodes represent inter-step data dependencies of tasks, which are the dependencies between different time steps.

Figure 5.5: Task dependency graph extracted from the audio equalizing model.

subsystems as containers of blocks. For the experiment on the Windows PC, the execution time of the parallelized program using the proposed method is reduced to 35% compared with a sequential program before parallelization, while the execution time ratio of the program parallelized automatically by the Intel Compiler is 85.8%. For the experiment on the RTOS system, the execution time of it has been reduced to 39%. On both of the PC and the embedded system that does not have any display device, any of result images were not displayed for the sake of profiling on different systems under the similar conditions.

5.10 Discussion

As shown in the experimental results in Table 5.1, the proposed method has worked effectively for the audio equalizing model, for which the execution time was reduced to 26.3% on the RTOS system. This result is very close to the ideal case for a 4-core processor. For the lane detection model, on the other hand, the execution time was reduced to 39% on the RTOS system. This section discusses this result.

One problem comes from a structure of task dependency graph. The task dependency graph of the audio equalizing model shown in Figure 5.5 has a simpler straight-forward structure than that of the lane detection model shown in Figure 5.6. This difference of task dependency graphs in terms of their structure results in the difference of performance in Table 5.1. In Figure 5.6, there are more long edges than in Figure 5.5. If there is a long edge between two nodes A and B and there is another path from node A to node B via other nodes, the nodes on the another path are difficult to be parallelized in a pipeline manor because node A cannot finish its task until node B starts its task. Remember each node works in the way shown in Figure 5.2.

Another problem is task granularity. Figures 5.7 and 5.8 show distribution of task execution time ratios for the audio equalizing and the lane detection, respectively. In these figures, a task execution time ratio over the maximum task execution time for the two models respectively is calculated for every task, and the number of tasks within every 1% segment of the ratio is plotted.

While the number of tasks that have lower task execution time ratio than 1% is 19 in the audio equalizing, the number of such tasks is 49 in the lane detection, as can be seen in Figures 5.7 and 5.8. The tasks that have lower task execution time ratio than 1% does not practically perform any calculation and does consume task communication overhead. This means that the parallelized lane detection has more workloads in terms of task communication than does the audio equalizing, and it leads to the lower execution time reduction of 39%.

5.11 Summary

This chapter proposes a method to generate parallel C code from models developed by the Simulink, which is a model development tool. The proposed method regards blocks in Simulink models as tasks and lines between blocks in Simulink models as communication channels respectively, and then generates parallel C code on the basis of the theory of communicating sequential processes (CSP). Under the process of parallelization, the proposed method breaks loop structures in Simulink models for parallelization while preserving the original semantics of the model. While the theory of CSP transfers data items through communication, the proposed method shares data items by using double buffers placed at shared memories. As a result of parallelizing an audio equalizer model on a four-core processor running at 400MHz the proposed method has reduced execution time down to 26.3% through parallelization.

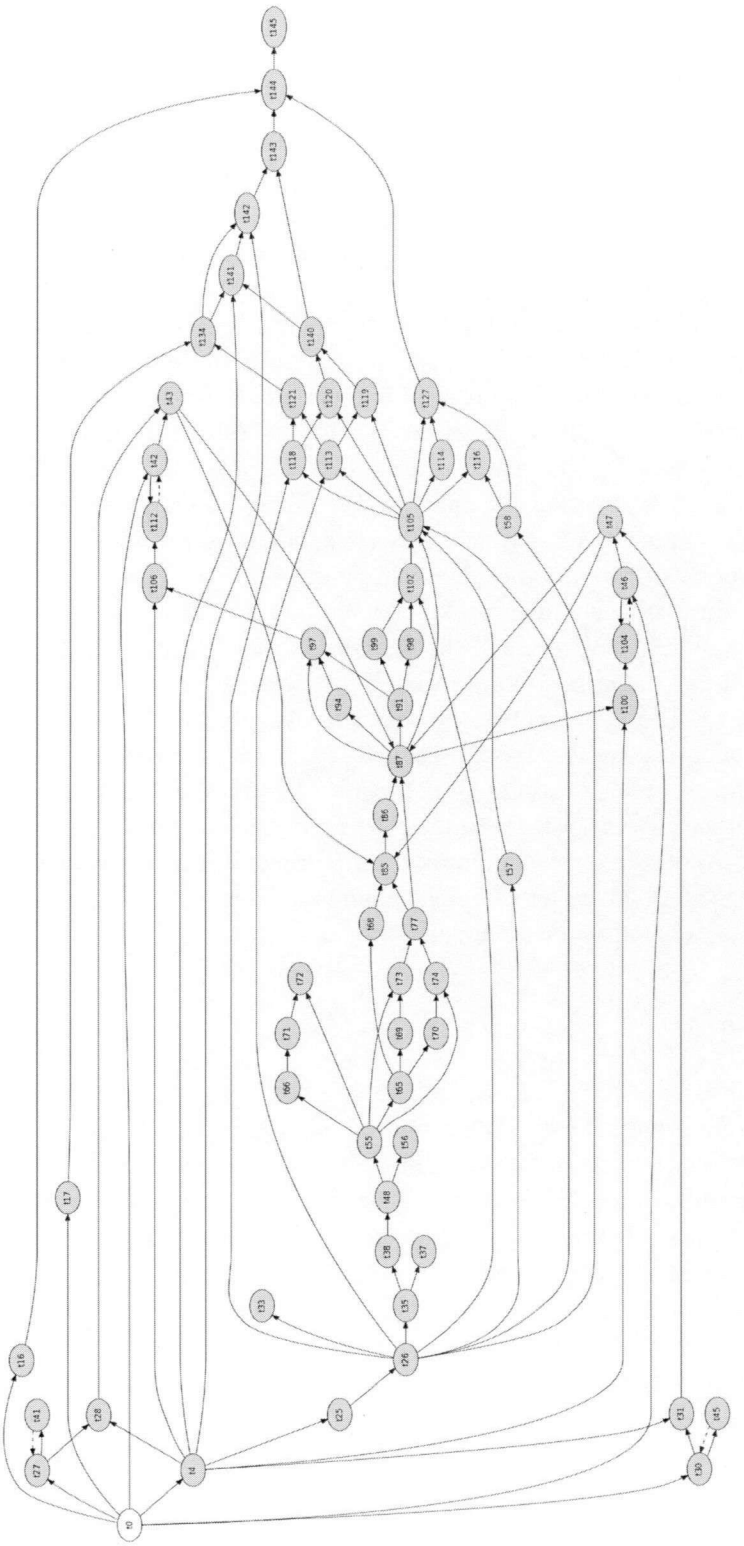
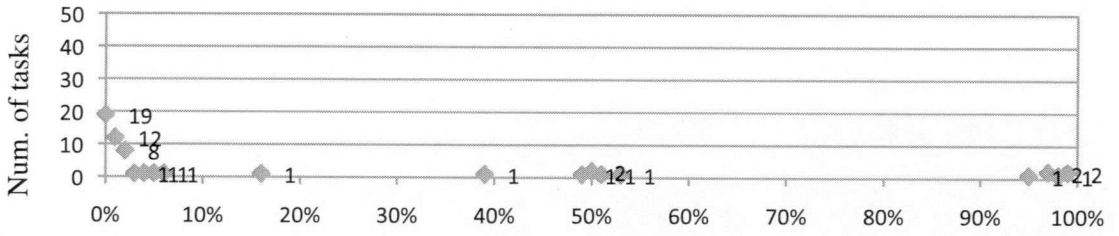
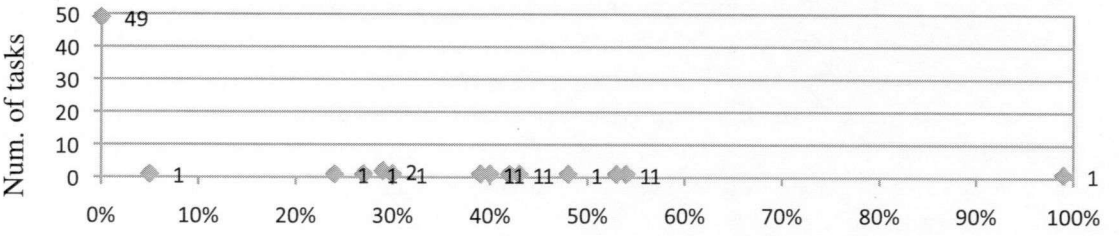


Figure 5.6: Task dependency graph extracted from the lane detection model.



Task execution time ratio over the maximum task execution time.

Figure 5.7: Histogram of task execution time ratio for the audio equalizing.



Task execution time ratio over the maximum task execution time.

Figure 5.8: Histogram of task execution time ratio for the lane detection.

Chapter 6

Conclusion

This thesis studied performance evaluation and design productivity improvement for digital signal processing systems. To deal with this enormous challenge, this thesis focused on the following major issues:

1. software development tool available at an early design stage for performance evaluation,
2. overall performance evaluation on real workloads before LSI fabrication, and
3. software parallelization of calculation-requiring digital signal processing applications to ease performance evaluation.

With regard to performance evaluation during actual processor design, this thesis first overviewed a design experience of a digital signal processor core in chapter 2. Through this design experience, the above major issues were derived.

The first contribution of this thesis is early-stage performance evaluation of a digital signal processing system composed of a processor core and many peripherals before LSI fabrication. To handle heavy real workloads of audio and video signals for performance evaluation, this method makes use of an FPGA and a test chip of a digital signal processor core as an intermediate approach between software simulation (versatile but slow running speed) and full hardware prototyping (fast running speed but difficult to modification). The developed method made it possible to emulate the target LSI composed of the processor core shown in the design experience and many peripherals running at a scaled-down operating frequency 1/3 and to evaluate audio and video processing on the LSI for actual situations.

The second contribution is demonstration of software development tool generation applicable for the use of performance evaluation. In the experimental result of the proposed method for software development tool generation in the case of adding SIMD instructions to the embedded microprocessor V850, by using intrinsic functions, the generated compiler yielded good code with only 7% increase in the number of instructions against the hand-optimized assembly codes. This means that the compiler generated from the proposed method gave a sufficient performance such that be able to be used for performance evaluation. This makes it possible to provide software development tool at an early design stage of processors for digital signal processing systems, and compiler availability improves productivity of performance evaluation of processors.

The third contribution is a solution to software parallelization. To increase the productivity in terms of parallelization of digital signal processing algorithm, this thesis proposed a method to generate parallel C codes suited to pipeline processing from models that represents behavior of digital signal processing applications as block diagrams. On the experiment of applying the method for an audio processing model, the execution time of the parallelized code could be reduced successfully to 26.3% on a 4-core processor running at 400MHz with a symmetric multi-processing real-time operating system, compared with that of the sequential code. Through this experiment this thesis demonstrated that parallelization based on models is promising to improve productivity of software parallelization for digital signal processing systems.

Future work

In the technology of software development tool generation, the optimized compiler generation from processor architecture descriptions is still challenging topic. To create machine-dependent part of compilers, in addition to deep knowledge of a target processor architecture, designers need extensive knowledge of specific rules on the internal structure of compilers. If many tasks involving compiler generation can be automated, which will significantly reduce development time required for a compiler based on the GCC, and you will be able to focus more time on architecture optimization of the target processor and compiler optimization.

In terms of generation of debuggers from processor architecture descriptions, supporting calling convention for target processors on debuggers is a task to be addressed. Convenient step execution commands such as 'step over', 'step into', and 'run untile return' will become available if debuggers understand the calling convention from a function to another on target processors. Although it is not difficult to implement the support for simple calling convention on debuggers, creating a scheme to accept various conceivable types of calling convention for debuggers is difficult in reality. If such a scheme is established indeed, designers will be able to generate more useful debuggers with more features for step execution.

Model-based parallelization still has several issues on both sides of software and hardware, which include, for example, model partitioning, task assignment to multiple processors, scheduling of multiple tasks, efficient method for communication and synchronization, finding bottlenecks of parallelized software, and so on. There are a number of previous studies on parallel programming based on dataflow so far. Dataflow programming, which is focused on and is used in particular fields, are in general not widely spread. One of the reasons is that there is no de facto open-source tool for dataflow programming. Model-based development is a kind of dataflow programming, and situation surrounding model-based development is being used in specific fields, same as that of dataflow programming. However, the larger the size of software becomes and the more the number of processors in a chip increases, situation could change. Models and dataflow have structural parallelism represented visually in them, and they are clearly suitable for intuitive parallel programming compared to programming based on just source code. If models are utilized in early design stages and in abstract level design and executable programs generated from the models come into practical use, the productivity of parallel software development of complex systems will be improved definitely.

Bibliography

[1] Keshab K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley and Sons, 1998.

[2] Keshab K. Parhi and Takao Nishitami, eds., *Digital Signal Processing for Multimedia Systems*, CRC Press, 1999.

[3] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed., Morgan Kaufmann, 2011.

[4] Uwe Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed., Springer, 2007.

[5] Dake Liu, *Embedded DSP Processor Design Embedded DSP Processor Design: Application Specific Instruction Set Processors*, Elsevier, 2008.

[6] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, March 2008.

[7] S. Bauer, J. Kneip, T. Mlasko, B. Schmale, J. Vollmer, A. Hutter, and M. Berekovic, "*The MPEG-4 multimedia coding standard: algorithms, architectures and applications*," Journal of VLSI Signal Processing, Vol. 23, pp. 7–26, October 1999.

[8] Madhukar Budagavi, Wendi Rabiner Heinzelman, Jennifer Webb, and Raj Talluri, "*Wireless MPEG-4 video communication on DSP chips*," IEEE Signal Processing Magazine, Vol. 17, No. 1, pp. 36–53, January 2000.

[9] Ichiro Kuroda and Takao Nishitani, "*Multimedia processors*," Proceedings of the IEEE, Vol. 86, No. 6, pp. 1203–1221, June 1998.

[10] Paolo Faraboschi, Giuseppe Desoli, and Joseph A. Fisher, "*The latest word in digital and media processing*," IEEE Signal Processing Magazine, Vol. 15, No. 2, pp. 59–85, March 1998.

[11] Jennifer Eyre and Jeff Bier, "*The evolution of DSP processors*," IEEE Signal Processing Magazine, pp. 43–51, March 2000.

[12] Takahiro Kumura, Daiji Ishii, Masao Ikekawa, Ichiro Kuroda, and Makoto Yoshida, "*A low-power programmable DSP core architecture for 3G mobile terminals*," Proceedings of International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vol. 2, No. ITT-4-3, pp. 1017–1020, May 2001.

[13] Tom R. Halfhill, "*StarCore reveals its first DSP*," Microprocessor Report, Vol. 13, No. 6, pp. 13–16, May 1999.

[14] Texas Instruments Inc., *TMS320C55x Technical Overview*. Literature Number SPRU393, February 2000.

[15] Ravi K. Kolagotla, Jose Fridman, Marc M. Hoffman, William C. Anderson, Bradley C. Aldrich, David B. Witt, Michael S. Allen, Randy R. Dunton, and Lawrence A. Booth Jr., "*A 333-MHz dual-MAC DSP architecture for next-generation wireless applications*," Proceedings of International Conference on the Acoustics, Speech, and Signal Processing, Vol. 2, pp. 1013–1016, 2001.

[16] Makoto Yoshida, Hiroyasu Ohtomo, and Ichiro Kuroda, "*A New generation 16-bit general purpose programmable DSP and its video rate application*," VLSI Signal Processing VI, pp. 93–101, 1993.

[17] Jose Fridman, "*Sub-word parallelism in digital signal processing*," IEEE Signal Processing Magazine, Vol. 17, No. 2, pp. 27–35, March 2000.

[18] Texas Instruments Inc., *TMS320C55x DSP Programmer's Guide*. Literature Number SPRU376A, July 2001.

[19] Simon Haykin, *Adaptive Filter Theory*, 3rd ed., Prentice Hall, 1996.

[20] Guozhu Long, Fuyun Ling, and John G. Proakis, "*The LMS algorithm with delayed coefficient adaptation*," IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. 37, No. 9, pp. 1397–1405, September 1989.

[21] Andrew J. Viterbi, *CDMA Principles of Spread Spectrum Communication*, Addison-Wesley Publishing Company, 1995.

[22] Yukihiro Naito and Ichiro Kuroda, "*H.263 mobile video CODEC based on a low power consumption digital signal processor*," Proceedings of International Conference on the Acoustics, Speech, and Signal Processing, Vol. 5, pp. 3041–3044, 1998.

[23] Atsushi Hatabu, Takashi Miyazaki, and Ichiro Kuroda, "*QVGA/CIF resolution MPEG-4 video CODEC based on a low-power and general-purpose DSP*," Proceedings of IEEE Workshop on Signal Processing Systems (SIPS), pp. 15–20, 2002.

[24] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, and Takeshi Yoshimura, "*A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication*," Proceedings of Design Automation Conference (DAC), pp. 299–304, 2004.

[25] Takahiro Kumura, Masao Ikekawa, Makoto Yoshida, and Ichiro Kuroda, "*VLIW DSP for mobile applications*," IEEE Signal Processing Magazine, Vol. 19, No. 4, pp. 10–21, July 2002.

[26] ARM Holdings, *AMBA Specification*, 2.0 ed., 1999.

[27] ISO/IEC, "*Information technology - coding of audio-visual objects - part 2: visual*," ISO/IEC 14496-2, 1999.

[28] Jamil Chaoui, "*OMAP^{TM}: enabling multimedia applications in third generation (3G) wireless terminals*," Dedicated Systems Magazine 2001 Q2, pp. 34–39, 2001.

[29] A. Fauth, J. Van Praet, and M. Freericks, "*Describing instruction set processors using nML*," Proceedings of the European Design and Test Conference, pp. 503–507, March 1995.

[30] Paul C. Clements, "*A survey of architecture description languages*," Proceedings of International Workshop on Software Specification and Design, pp. 16, March 1996.

[31] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr, "*LISA - machine description language and generic machine model for HW/SW co-design*," Proceedings of the IEEE Workshop on VLSI Signal Processing, pp. 127–136, 1996.

[32] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau, "*EXPRESSION: a language for architecture exploration through compiler/simulator retargetability*," Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp. 485–490, March 1999.

[33] Manuel Hohenauer, Hanno Scharwaechter, Kingshuk Karuri, Oliver Wahlen, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Hans van Someren, "*A methodology and tool suite for C compiler generation from ADL processor models*," Proceedings of the Conference on Design, Automation and Test in Europe (DATE), Vol. 2, pp. 1276–1281, March 2004.

[34] Shinsuke KOBAYASHI, Yoshinori TAKEUCHI, Akira KITAJIMA, and Masaharu IMAI, "*Compiler generation in PEAS-III: an ASIP development system*," Proceedings of International Workshop on Software and Compilers for Embedded Processors (SCOPES), March 2001.

[35] Redhat, "*CGEN: the cpu tools generator.*" `http://sources.redhat.com/cgen/`.

[36] Maghsoud Abbaspour and Jianwen Zhu, "*Retargetable binary utilities*," Proceedings of Design Automation Conference (DAC), pp. 331–336, June 2002.

[37] Prabhat Mishra, Aviral Shrivastava, and Nikil Dutt, "*Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs*," ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 11, No. 3, pp. 626–658, July 2006.

[38] Alexandro Baldassin, Paulo Centoducatte, and Sandro Rigo, "*An open-source binary utility generator*," ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 13, No. 2, pp. 1–17, April 2008.

[39] Ricardo E. Gonzalez, "*Xtensa: a configurable and extensible processor*," IEEE Micro, Vol. 20, No. 2, pp. 60–70, March-April 2000.

[40] David Goodwin and Darin Petkov, "*Automatic generation of application specific processors*," Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pp. 137–147, 2003.

[41] Fei Sun, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha, "*Custom-instruction synthesis for extensible-processor platforms*," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 2, pp. 216–228, February 2004.

[42] Yuji Nagamatsu, Nagisa Ishiura, and Nobuyuki Hikichi, "*Retargeting GCC and GNU toolchain for extended instruction set*," Technical report of IEICE, No. VLD2005-103, pp. 37–41, January 2006.

[43] NEC Electronics Corporation, *User's Manual: V850 Family for Architecture*, March 2001.

[44] Intel Corporation, *Using Streaming SIMD Extensions 2 (SSE2) to Implement an Inverse Discrete Cosine Transform*, 2.0 ed., September 2000.

[45] Tensilica, Inc., *Tensilica Instruction Extension (TIE) Language Reference Manual*, November 2006.

[46] Yuji Kunitake, Takahiro Kumura, and Hiroto Yasuura, "*A case study on instruction set extension for variable length decoding on a custom processor*," IPSJ Technical Report ARC, Vol. 2010-ARC-187, No. 21, pp. 1–6, January 2010 (in Japanese).

[47] Takahiro Kumura, Soichiro Taga, Nagisa Ishiura, Yoshinori Takeuchi, and Masaharu Imai, "*Automatic generation of GNU binutils and GDB for ASIP cores based on plug-in method*," Proceedings of Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), March 2012, to appear.

[48] Lina J. Karam, Ismail AlKamal, Alan Gatherer, Gene A. Frantz, David V. Anderson, and Brian L. Evans, "*Trends in multicore DSP platforms*," IEEE Signal Processing Magazine, Vol. 26, No. 6, pp. 38–49, November 2009.

[49] Yukihiro Takeuchi, Yohei Nakata, Hiroshi Kawaguchi, and Masahiko Yoshimoto, "*Scalable parallel processing for H.264 encoding application to multi/many-core processor*," Proceedings of International Conference on Intelligent Control and Information Processing (ICICIP), pp. 163–170, August 2010.

[50] Ngai-Man Cheung, Xiaopeng Fan, Oscar C. Au, and Man-Cheung Kung, "*Video coding on multicore graphics processors*," IEEE Signal Processing Magazine, Vol. 27, No. 2, pp. 79–89, March 2010.

[51] Doug Pulley, "*Multi-core DSP for base stations: large and small*," Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC), pp. 389–391, March 2008.

[52] Mitsuhiro Matsunaga, Eiji Tsuboi, Satoru Shimojima, Masaki Nakamizo, and Hiroyuki Gojima, "*EMMA3, an LSI for HD DVD player/recorder systems*," NEC Technical Journal, Vol. 2, No. 4, December 2007.

[53] Richard Selvaggi and Larry Pearlstein, "*Broadcom mediaDSP: a platform for building programmable multicore video processors*," IEEE Micro, Vol. 29, No. 2, pp. 30–45, March/April 2009.

[54] Masanori Mori, Yoshihiro Nishida, and Kenichi Doniwa, "*CELL platform expanding limits of TV performance*," TOSHIBA REVIEW, Vol. 65, No. 4, 2010 (in Japanese).

[55] Masayasu Yoshida, Takeshi Sugihara, Toshiaki Takahashi, Yasuhiko Koumoto, and Toshinori Ishihara, "*"NaviEngine 1," system LSI for SMP-based car navigation systems*," NEC Technical Journal, Vol. 2, No. 4, 2007.

[56] Takamichi Miyamoto, Saori Asaka, Hiroki Mikami, Masayoshi Mase, Yasutaka Wada, Hirofumi Nakano, Keiji Kimura, and Hironori Kasahara, "*Parallelization with automatic parallelizing compiler generating consumer electronics multicore API*," Proceedings of International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 600–607, December 2008.

[57] Edward A. Lee, "*The problem with threads*," IEEE Computer, Vol. 39, No. 5, pp. 33–42, May 2006.

[58] Hahn Kim and Robert Bond, "*Multicore software technologies*," IEEE Signal Processing Magazine, Vol. 26, No. 6, pp. 80–89, November 2009.

[59] Jason Sanders and Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, July 2010.

[60] John E. Stone, David Gohara, and Guochun Shi, "*OpenCL: a parallel programming standard for heterogeneous computing systems*," Computing in Science and Engineering, Vol. 12, No. 3, pp. 66–73, May/June 2010.

[61] Michael I. Gordon, William Thies, and Saman Amarasinghe, "*Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*," Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 151–162, 2006.

[62] William Thies, Michal Karczmarek, and Saman P. Amarasinghe, "*StreamIt: a language for streaming applications*," Proceedings of International Conference on Compiler Construction, pp. 179–196, 2002.

[63] Motohiro Takayama, Ryuji Sakai, Nobuhiro Kato, and Tomofumi Shimada, "*Molatomium: parallel programming model in practice*," USENIX Workshop on Hot Topics in Parallelism, June 2010.

[64] Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu, "*Automatic parallelization of Simulink applications*," Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 151–159, 2010.

[65] Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu, "*Skewed pipelining for parallel Simulink simulations*," Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 891–896, March 2010.

[66] Sang il Han, Xavier Guerin, Soo-Ik Chae, and Ahmed. A. Jerraya, "*Buffer memory optimization for video codec application modeled in Simulink*," Proceedings of Design Automation Conference (DAC), pp. 689–694, July 2006.

[67] Lisane Brisolara, Sang il Han, Xavier Guerin, Luigi Carro, Ricardo Reis, Soo-Ik Chae, and Ahmed Jerraya, "*Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC*," Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 81–89, 2007.

[68] dSPACE, "*Real-time interface for multiprocessor systems (RTI-MP).*" `http://www.dspace.jp/en/pub/home/products/sw/impsw/rtimpblo.cfm`.

[69] Stephen D. Brookes, Charles Antony Richard Hoare, and A. W. Roscoe, "*A theory of communicating sequential processes*," Journal of ACM, Vol. 31, No. 3, pp. 560–599, July 1984.

[70] Charles Antony Richard Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[71] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli, *Readings in Hardware/Software Co-design*, Ch. Design of embedded systems: formal models, validation, and synthesis, pp. 86–107, Kluwer Academic Publishers, 2002.

[72] Axel Jantsch and Ingo Sander, "*Models of computation for embedded system design*," IEE Proceedings of Computers and Digital Techniques, Vol. 152, No. 2, pp. 114–129, Mar 2005.

[73] Alberto Sangiovanni-Vincentelli and Marco Di Natale, "*Embedded system design for automotive applications*," Computer, Vol. 40, pp. 42–51, October 2007.

[74] James Lyle Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, 1981.

[75] Soonhoi Ha, Sungchan Kim, Choonseung Lee, Youngmin Yi, Seongnam Kwon, and Young-Pyo Joo, "*PeaCE: a hardware-software codesign environment for multimedia embedded systems*," ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 12, No. 3, pp. 24:1–24:25, May 2008.

[76] University of California Berkeley, "*Ptolemy project*," `http://ptolemy.eecs.berkeley.edu/`, 2011.

[77] Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, and Soonhoi Ha, "*Conversion of reference C code to dataflow model H.264 encoder case study*," Proceedings of Asia and South Pacific Conference on Design Automation (ASPDAC), pp. 152–157, March 2006.

[78] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy Pimentel, Simon Polstra, R. Bose, Claudiu Zissulescu, and Ed Deprettere, "*Daedalus: toward composable multimedia MP-SoC design*," Proceedings of Design Automation Conference (DAC), pp. 574–579, 2008.

[79] G. Kahn, "*The semantics of a simple language for parallel programming*," Proceedings of the IFIP Congress, pp. 471–475, 1974.

[80] Younes Seyedi, "*Professional Simulink audio equalizer.*" `http://www.mathworks.com/matlabcentral/fileexchange/`.

[81] The Mathworks, "*Lane departure warning system.*" a sample model distributed with Video and Image Processing Toolbox for MATLAB/Simulink.

[82] Masaki Gondo, "*Blending asymmetric and symmetric multiprocessing with a single OS on ARM11 MPCore*," eSOL Co., Ltd. white paper, Information Quarterly, Vol. 6, No. 2, 2007.