Low power processor architecture and multicore approach for embedded systems

メタデータ	言語: eng 出版者: 公開日: 2017-10-05 キーワード (Ja): キーワード (En): 作成者: メールアドレス: 所属:
URL	http://hdl.handle.net/2297/43866

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 International License.



博 士 論 文

Low power processor architecture and multicore approach for embedded systems

組込み用途向け低消費電力プロセッサ・アーキテクチャとマルチコア研究

金沢大学自然科学研究科

電子情報科学専攻

学籍番号（1323112001）

氏 名 大谷　寿賀子

主任指導教員名 新居　浩二

提出年月 ‘15/10/28

Contents

Contents 1

List of Figures 6

List of Tables 9

Acknowledgements 11

Chapter 1 13

Introduction 13

Chapter 2 17

Applications and System Trends 17

2.1 Four Key Technologies that support IoT 17

2.2 Research Goals 19

Chapter 3 21

Low-Power MCU Processor Architecture 21

3.1 Microcontroller Basic Strcture 21

3.2 Basic Design Approach for Energy Saving 22

3.3 Introduction to Low-Power Architecture 24

3.4 Core features to boost performance 27

3.4.1 RX Architecture Overview 27

3.4.2 RXv2 Pipeline Design 27

3.4.3 Pipeline integrated FPU 28

3.4.4 DSP with wide accumulators 29

3.5 Energy saving architecture 31
3.5.1 Embedded memory system architecture 31
3.5.2 Improving instruction fetch effectiveness 31
3.5.3 AFU: Advanced Fetch Unit 32
3.5.4 Processor performance and power consumption 34
3.5.5 Core features to make code compact 37
3.6 RX instruction set architecture 37
3.6.1 Overview of Instruction set 37
3.6.2 Optimized op codes leads to superior code density 38
3.6.3 Data Transfer instruction 40
3.6.4 1byte conditional branch instruction 44
3.6.5 Compare instruction 45
3.6.6 3-operand instruction 47
3.6.7 Registers 48
3.6.8 Code size evaluation 51
3.7 Related Works 52
3.8 Summary 54
3.9 List of RX instruction Set 56
3.9.1 Arithmetic and logical instructions 56
3.9.2 Floating-point operation instructions 57
3.9.3 Data Transfer instructions 58
3.9.4 Branch Instructions 59
3.9.5 Bit manipulation instructions 60

3.9.6 String manipulation instructions 60
3.9.7 System control instructions 61
3.9.8 DSP function instructions 62
Chapter 4 63
PEACH: A Multicore Communication SoC with PCI Express I/F 63
4.1 Introduction 63
4.2 PEACH Architecture 65
4.2.1 PEACH overview 65
4.2.2 Chip Architecture 67
4.2.3 PCI Express interface with up-configuration function 69
4.2.4 PCI Express up-configuration function 71
4.2.5 Intelligent Interrupt Controller 73
4.3 Network Managing 74
4.3.1 Data Flow Control 74
4.3.2 PEARL network route construction 78
4.3.3 Network system power management 78
4.4 Evaluation System 78
4.4.1 PEARL system board 78
4.4.2 Switching time of PCI Express up-configuration function 81
4.5 Related Works 81
4.6 Summary 82
Chapter 5 83

4

A Heterogeneous Multicore SoC for Secure Multimedia Applications....83

5.1 Introduction....83

5.2 Secure Media System....84

5.2.1 Concept of the secure media system....84

5.2.2 SoC Overview....87

5.2.3 Physical Integration of the SoC and the SiP....89

5.2.4 Protection by Software....90

5.3 Multicore Hypervisor....91

5.3.1 Micro Clustering Model....91

5.3.2 Functions of the multicore hypervisor....92

5.3.3 Startup sequence....93

5.3.4 Inter-OS Communication....94

5.3.5 Interrupt handling....94

5.3.6 Hypervisor Operating System....95

5.4 System Software....96

5.4.1 Software Architecture....96

5.4.2 Secure media block software....97

5.4.3 Task mapping....98

5.5 System Evaluation....100

5.5.1 Evaluation system....100

5.5.2 Evaluation results....101

5.6 Related Works....103

5.7 Summary....104

Chapter 6 105

Conclusions 105

6.1 Future work 105

References 107

Publications 112

List of Figures

Figure 1.1 Thesis outline 14

Figure 2.1 Four key technologies that support IoT 18

Figure 3.1 MCU (Microcontroller) Basic Structure 21

Figure 3.2 Intermittent operations for reduction in power consumption 22

Figure 3.3 Power break down of a microcontroller 23

Figure 3.4 RXv2 CPU block diagram. 25

Figure 3.5 Overview of RXv2 CPU core 25

Figure 3.6 RX core road map 26

Figure 3.7 Benchmark Comparison 26

Figure 3.8 RXv2 pipeline structure 28

Figure 3.9 The Coprocessor-type FPU and the pipeline integrated-type FPU (proposed) 28

Figure 3.10 RX DSP functionality 30

Figure 3.11 MCU: High-Capacity Internal Flash 32

Figure 3.12 Fetch Unit for Microcontroller with Advanced Fetch Unit 33

Figure 3.13 Embedded Flash processing performance 33

Figure 3.14 Benchmark Results of DSP Algorithm programs such as FFT, IIR filter and Matrix under zero-wait flash memory access. 35

Figure 3.15 Performance Comparison of RXv2 with RXv1 and a RISC processor 36

Figure 3.16 RX instruction set architecture 38

Figure 3.17 Analysis of Instruction Frequency 39

Figure 3.18 Byte assignment of RX Instruction format 39

Figure 3.19 Analysis of general-purpose register configuration 49

Figure 3.20 Register Set of the CPU....50

Figure 3.21 Code size analysis of the RX and a RISC-based MCU: Static Code Size (a) and Dynamic Code (b)....51

Figure 3.22 The test chip of the microcontroller with the RXv2 processor....54

Figure 4.1 The communication link, PEARL. 64

Figure 4.2 Neighbor communication on PEARL. 66

Figure 4.3 PEACH block diagram. 67

Figure 4.4 PEACH micrograph. 68

Figure 4.5 PCI Express up-configuration function by software control. (a) Maximum data transfer rate (b) Low power consumption. 71

Figure 4.6 Power Consumption of PCI Express PHY (W) at each requested transfer volume. . 72

Figure 4.7 Block Diagram of Intelligent ICU 74

Figure 4.8 Efficient packet processing and fault handling in PEACH. 75

Figure 4.9 Two data transmission flows: Processor mode using interrupt services 76

Figure 4.10 The intelligent ICU's fast automatic data-transfer function improves transfer latency. 77

Figure 4.11 Prototype of PEARL network system. 80

Figure 5.1 Implemented secure media system board....84

Figure 5.2 Concept of the secure media system. 86

Figure 5.3 Block diagram of the SoC 86

Figure 5.4 Micrograph of SoC and SiP....89

Figure 5.5 Protection by software....90

Figure 5.6 Multicore hypervisor and micro clustering model. 92

Figure 5.7. Startup sequence. 93

Figure 5.8. Interrupt operation. 95

Figure 5.9. Software layer configuration 97

Figure 5.10. Task structure in the secure media block 98

Figure 5.11. Task mapping in the secure media block 99

Figure 5.12. Data-flow of decoding MP4 container file 99

Figure 5.13. Implemented evaluation system 100

Figure 5.14. Block diagram of the evaluation system 101

Figure 5.15. Comparison of workload of CPU#1. 102

Figure 5.16. Workload balance. 103

List of Tables

Table 3.1 Design Highlights of a Low Power CPU 23

Table 4.1 PEACH Chip Features 70

Table 4.2 Comparison of Power Efficiency 70

Table 4.3 Power Consumption of PCI Express PHY (Normalized) 72

Table 4.4 PCI Express up-configuration function-switching time. 81

Table 5.1 Functional features of the SoC 88

Table 5.2 Physical features of SoC and sip 88

Table 5.3 Combinations of Supported OSs 96

Table 5.4 Workload of CPU#1 in Secure Media Block 102

Acknowledgements

Research in microprocessor architecture requires a team effort. During my research, I was fortunate to work with great people who influenced the direction and the quality of my work.

First, I would like to thank Koji Nii, my thesis advisor, for his overall guidance and support.

I am especially grateful to Yoshio Matsuda, my thesis advisor, and Toru Shimizu, my former thesis advisor now at Keio University, for providing me with the opportunity to study as a doctoral student and for their encouragement.

I am especially grateful to Hiroyuki Kondo, a chief processor architect at Renesas Electronics Corporation, for mentoring me. He helped me develop a passion for research and encouraged me to pursue developing microprocessor architecture. Our numerous discussions had a strong influence on my research.

I also want to thank following people: Kunle Olukotun at Stanford University. His guidance motivated me to cultivate multicore processor architecture. Kazutami Arimoto, at Okayama Prefecture University, Taisuke Boku at Tsukuba University, Toshihiro Hanawa at Tsukuba University and Christoforos Kozyrakis at Stanford University. They gave me valuable advice for my researches and papers. Kazuya Ishida, Isao Kotera and Naoshi Ishikawa, my colleagues at Renesas Electronics Corporation with whom I have been working together for more than ten years. Without their persistence and selflessness, our progress would not have been possible.

No acknowledgments page would be complete without thanking Ellen Higuchi for wrestling with my technical papers and giving plausible English expressions.

Finally, I want to thank my parents, Shigeki and Sueko Otani, my brother Hiroyuki Otani, for their love and unwavering support.

The PEACH project was supported by a JST/CREST program entitled “Computation Platform for Power-aware and Reliable Embedded Parallel Processing Systems”.

The TRON-SMP project was partially funded by the New Energy and Industrial Technology Development Organization (NEDO), via Grant #0628002.

Chapter 1

Introduction

"IoT" or "Internet of things" formerly known as "ubiquitous computing" has been absolutely essential to our society and its infrastructures. Devices are linked to networks from anywhere in the world and will be mutually controlled while information is being exchanged. A microcontroller is one of the important elements of IoT. The microcontroller designers are strongly urged to achieve both high performance computation and low power consumption, which is a hybrid technology with powerfulness of computing and friendliness to the environment. Furthermore, while network services are gaining popularity, dependability and security of network are more important. A key solution to meet these demands is a compact and low power processor core and multicore technology.

This thesis focuses on the development of efficient microcontroller architecture for IoT. The basis for the argument is the key of a low power processor architecture is how effective handle on chip memories. Furthermore, collaboration of software and hardware on multicore architecture can provide dependable and secure networks.

Thesis Contributions

The main contributions of this dissertation are:

- An RX processor core which is suitable for IoT. The RX processor Instruction set architecture (ISA) and its microarchitecture can achieve lower power consumption and boost performance.
- An eight-core communication SoC with PCI Express interface. The multicore SoC can realize a high-performance, power-aware, highly dependable network.
- A secure multimedia system that uses heterogeneous multicore SoC and software virtualization.

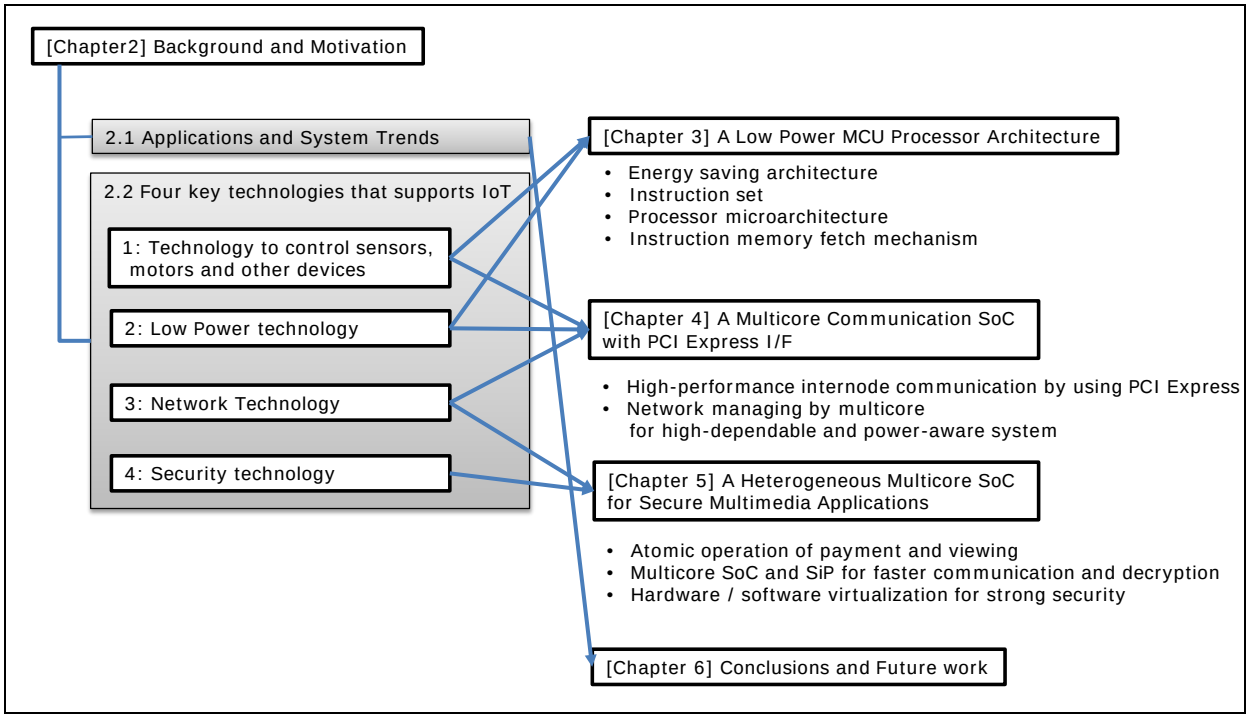


Figure 1.1 Thesis outline

The outline of the remainder of this thesis is as follows (Figure 1.1).

Chapter 2 provides the background and motivation for this work. It discusses the characteristics and requirements of IoT by presenting four key IoT technologies.

Chapter 3 introduces RX processor core with a low-power processor architecture. The RX processor instruction set architecture (ISA) and its microarchitecture can achieve lower power consumption and boost performance. RXv2 reaches 4.5 Coremark per MHz and the RXv2 processor delivers approximately more than 2.2 – 5.7x the power efficiency of the previous work. The RXv2 processor delivers 1.9 – 3.7x the cycle performance of previous work in digital signal applications. This chapter is from [S. Otani and H. Kondo, “RX v2: Renesas’s New-Generation MCU Processor,” IEICE Transactions, Vol. E98-C, No. 7, pp. 544-549, Jul. 2015, (copyright ©2015 IEICE).]

Chapter 4 presents an eight-core communication SoC with PCI Express interface. PEACH with four PCI Express ports realizes high-performance communication of 4 x 20Gbps and power efficiency of 0.04W/Gbps. The power efficiency of InfiniBand 4X (Commodity network devices) is 0.083W/Gbps. Thus, PEACH provides 51.5% better power efficiency than InfiniBand 4X. We also evaluate the PEARL network system and demonstrate its fault-tolerant ability. This chapter is from [S. Otani, H. Kondo, I. Nonomura, T. Hanawa, S. Miura and T. Boku, “Peach: A Multicore Communication System

on Chip with PCI Express," IEEE Micro, vol. 31, no. 6, pp. 39-50, Nov.-Dec. 2011, copyright ©2011 IEEE).]

Chapter 5 demonstrates a secure multimedia system by using a heterogeneous multicore SoC with SiP and software virtualization. The multicore hypervisor virtualizes hardware resources and prohibits operating systems and applications from accessing hardware resources directly. This chapter is from [H. Kondo, O. Yamamoto, S. Otani, N. Sugai, and T. Shimizu, "Software architecture of a secure multimedia system using a multicore SoC and software virtualization," in IEEE Int. Conf. Consumer Electronics, Dig. Tech. Papers, pp. 1-2, Jan. 2009, (copyright ©2009 IEEE)]

Finally, Chapter 6 concludes the thesis and suggests directions for future work.

Chapter 2

Applications and System Trends

The IoT, or Internet of Things, has become popular. Giving intelligence to devices and connecting them together creates new value.

With the diffusion of IoT, devices operate independently and work autonomously. If IoT is employed, devices can be linked via networks, working autonomously to provide a pleasant environment for people working in the office, in the city, at home and in the factory.

IoT is experiencing rapid evolution. In 2020, the year of the Tokyo Olympics, 50 billion devices will be connected to a network. A trillion sensors will be connected to a network [bryzek14]. An era is about to begin in which everything is linked to huge networks.

2.1 Four Key Technologies that support IoT

There are four key technologies that support IoT, 1) network technology to link one device to another, 2) technology to control sensors, motors and other devices, 3) low power consumption technology to raise energy efficiency and 4) security technology (Figure 2.1).

The shift of centralized control and operating systems will accelerate toward distributed systems, and network servers are no exception. Highly dependable network technology is vital to connect downsized servers in various locations.

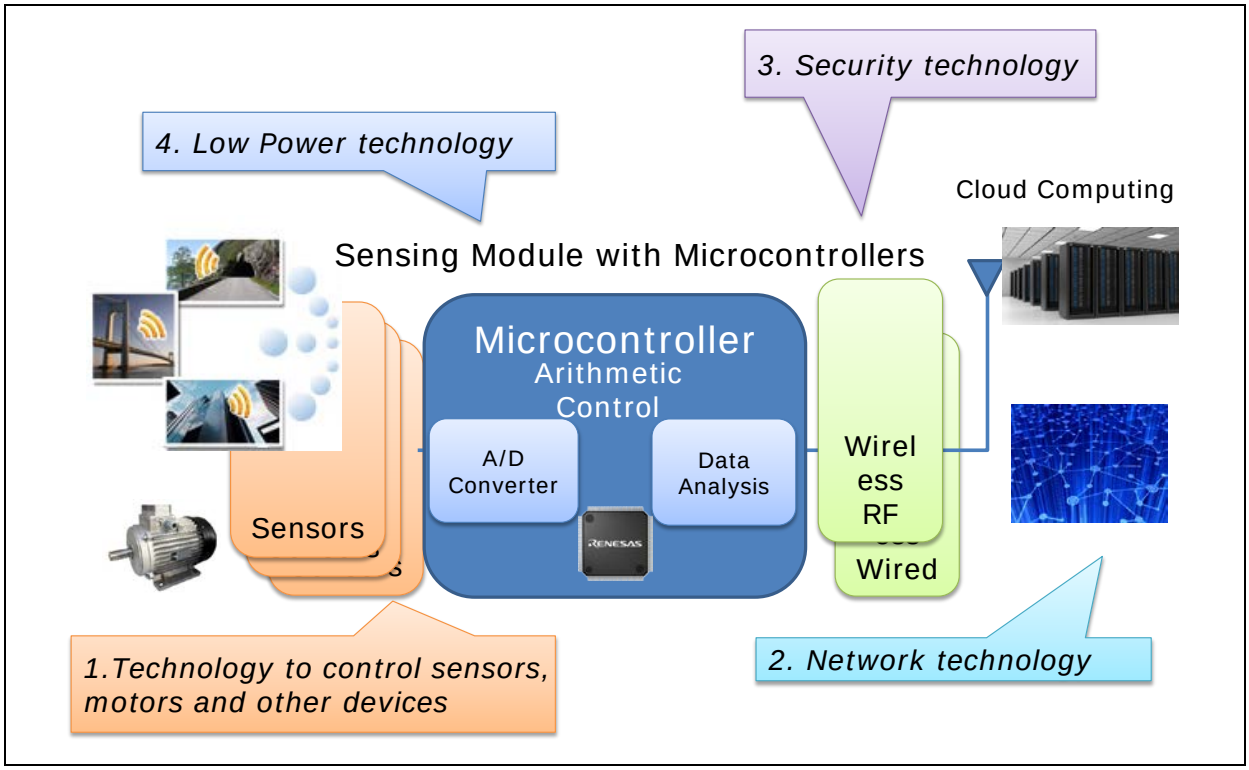


Figure 2.1 Four key technologies that support IoT

The technology to control sensors and security technology to ensure the solid protection of information are particularly important. For example, recent advances in infrastructure technology include construction monitoring, which has been installed in bridges, tunnels, and roads. The number of installations of network cameras to monitor the environment will be five times larger than in 2006. These monitors can be controlled over the network. But if the systems are hacked, severe incidents and panic ensue. Security technologies can protect society against these risks. IoT is offering comfort and convenience, but with security concerns.

With an increase in the number of devices on networks, power consumption becomes a major issue. Sensing modules must always be active to collect information and be long-lived in infrastructures.

Centralized control for energy saving via networks is evolving. One of effective way to reduce energy saving is to adopt inverter technologies. The inverter adoption ratio is not high in developing countries. Even in air conditioners which use the largest amount of power, only 50% utilize inverter technology in the world. A 10% increase in world inverter adoption would reduce the number of thermal power plants by 430. There are two reasons that inverter technology has not spread: to avoid

difficulty of system design and to meet lower cost requirement by using sensor-less motors. MCU can solve these problem

In IoT applications, it is vital to consider how to link applications and microcontrollers and how to communicate for people with electronics devices.

2.2 Research Goals

Given the applications and systems requirements, we consider four key technologies for an efficient microcontroller architecture for IoT systems:

- Network technology
- Security technology
- Technology to control sensors, motors and other devices
- Low-power technology

The above features of the architecture and microarchitecture techniques are presented in the following chapters.

Chapter 3

Low-Power MCU Processor Architecture

3.1 Microcontroller Basic Strcture

MCUs (microcontrollers), which control electric devices, consist of CPUs, memories and peripheral interfaces. Figure 3.1 shows the basic structure of MCUs. The CPU reads instructions, decodes and executes arithmetic operations and read/write data. Memories store program code and data. Peripheral interfaces connect the CPU and I/O devices. There are two types of memories. Flash memory is a ROM (Read Only Memory) which mainly stores instructions and retains data even if power is turned off. SRAM is a RAM (Random Access Memory) which stores data. This working memory loses data if power is turned off. The feature that most distinguishes MCU from MPU (microprocessor units) is integration of the memory system. This feature contributes to low power consumption by eliminating wiring between external memories and a chip.

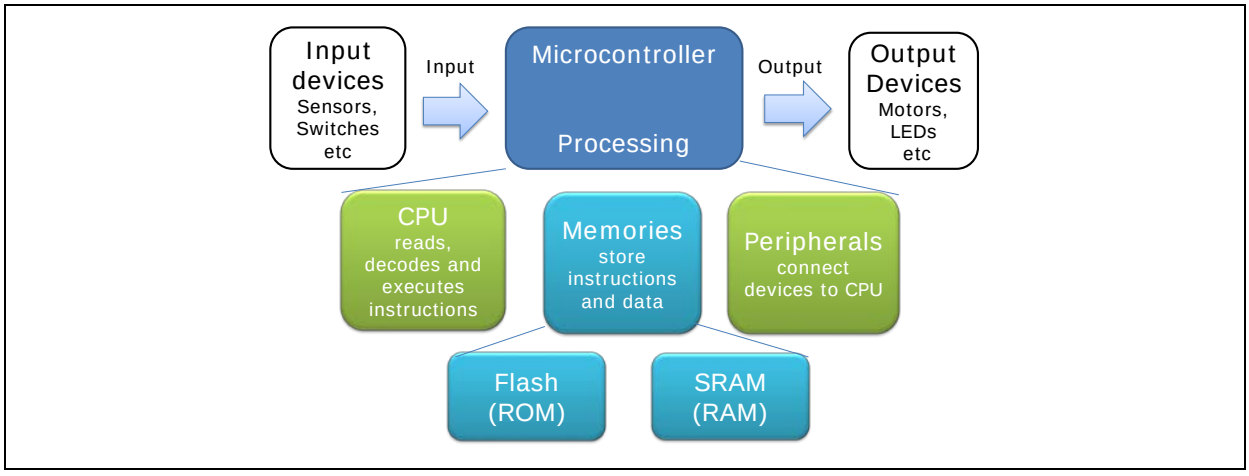


Figure 3.1 MCU (Microcontroller) Basic Structure

3.2 Basic Design Approach for Energy Saving

The basic strategy of reducing power consumption is to lower the operating current and shorten the operating time. Figure 3.2 shows the difference in power consumption of a low-power microcontroller with another microcontroller. The blue bar represents an energy-saving microcontroller with lower operating current and higher performance. The low-power microcontroller completed the same task in much less time, which also enables it to stay in low-power sleep mode longer. This intermittent operations strategy of low-power microcontrollers enables batteries to last a long time.

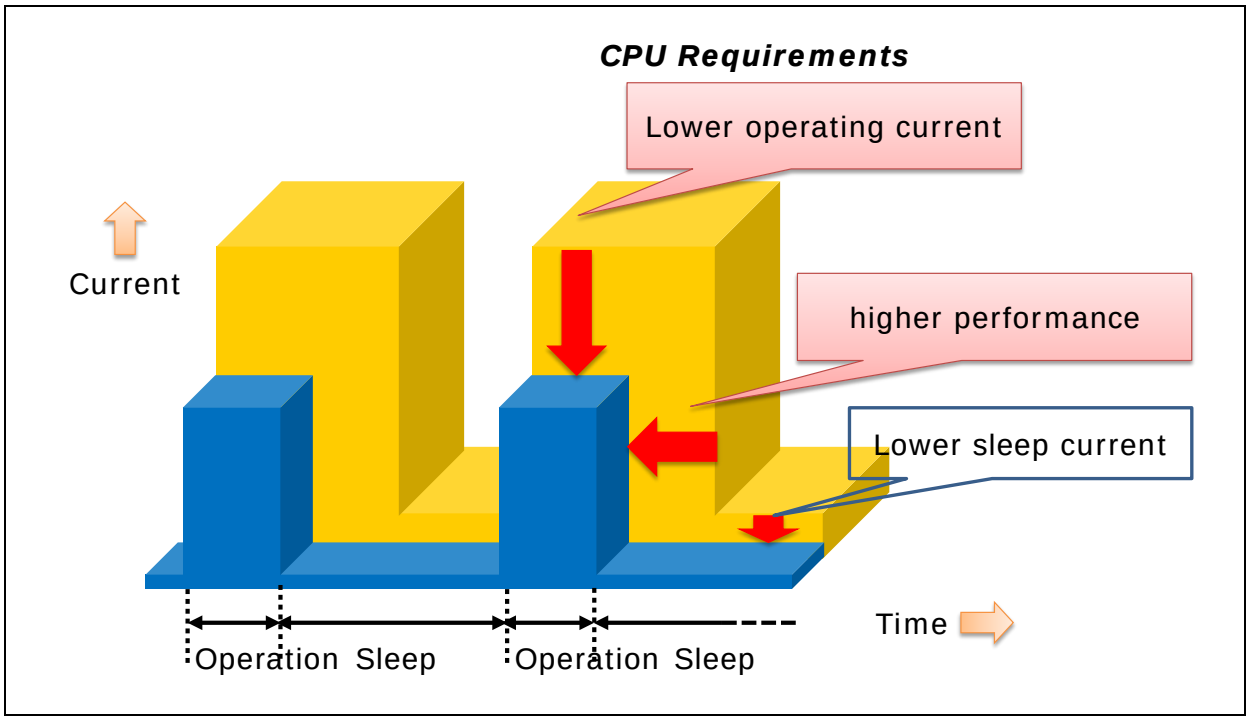


Figure 3.2 Intermittent operations for reduction in power consumption

Design highlights of a low-power processor architecture are shown in Table 3.1. Three rows are CPU design highlights; instruction set architecture, processor microarchitecture and memory access mechanism. The check marks indicate the particular design meets the particular requirement.

All three items are vital to achieve high performance. Instruction set architecture and memory access mechanisms contribute to low operating current.

	Requirement	High Performance	Low Operating Current
Design Highlights	Instruction Set	✔	✔
	Micro-Architecture (Hardware Structure)	✔	
	Memory Access Mechanism	✔	✔

Table 3.1 Design Highlights of a Low Power CPU

MCUs (microcontroller units) with on-chip memory systems substantially reduce energy consumption compared to MPUs (microprocessor units) with off-chip memory systems because of the wiring capacity between external memories and the chip. However, the low-power requirement of the embedded applications is more and more strict. The power breakdown of a microcontroller is shown in Figure 3.3. A substantial portion of chip power comes from internal Flash memory. Therefore, reducing Flash memory directly affects the reduction of power consumption of the whole microcontroller. Considering microcontroller structure, the greater part of the Flash accesses comes from instruction fetches.

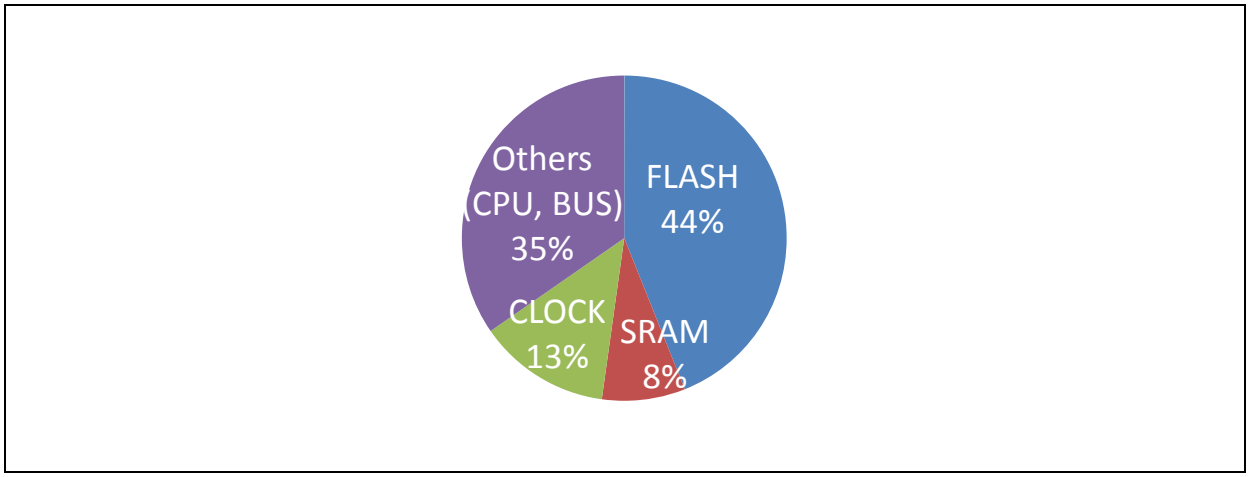


Figure 3.3 Power break down of a microcontroller

3.3 Introduction to Low-Power Architecture

Application fields of microcontrollers have spread to building automation, medical devices, motor control, e-metering, and home appliances. The demand for such highly intelligent systems has increased. To meet the demand, the scale and complexity of software has begun to rise. The rapid growth of memory capacity and the advance of microcontroller functions have led to the higher frequency and higher processing performance of embedded processors. Furthermore, many embedded systems still have high cost, power consumption, and space constraints. In order to meet users' demands for these requirements, new RX processor core (RXv2) architecture has been developed. [otani13].

It is vital for MCUs to handle floating point computation requirements to meet the recent demand for industrial applications. However, the cost of adding an FPU unit to existing MCUs would have been extremely high. The RX includes a compact single precision FPU as a part of the MCU's basic configuration [linley10], [mips13].

The FPU/DSP functions of the new RXv2 have been enhanced. The RXv2 processor block diagram is shown Figure 3.4. The core has integer, divide, multiply-accumulate and floating point units with sixteen 32bit general purpose registers. Key differences from the previous processor, RXv1, are an improved dual-issue pipeline structure, DSP extensions and a pipelined FPU. The overview of RXv2 specification shows in Figure 3.5.

The RXv2 processor core also incorporates AFU to reduce pipeline branch penalties and Flash memory accesses. The improved power efficiency of the RXv2 architecture with our benchmark evaluation will be discussed in Section 3.5.

Program code is, of course, often the largest consumer of memory in control-intensive applications, affecting both system cost and size. Also, instruction fetches are responsible for a significant portion of power and memory bandwidth. Therefore, both static and dynamic code size are key factors in embedded systems. RX family instruction set architecture uses variable-length instructions to minimize the static and dynamic code size.

These features have the benefit of boosting performance and making code compact. Figure 3.6 shows RXv2 CPU core roadmap. RX has two generations, RXv1 and RXv2. Figure 3.7 shows performance comparison to other embedded processors. RX reaches 4.5 Coremark/MHz on an integer benchmark for embedded systems.

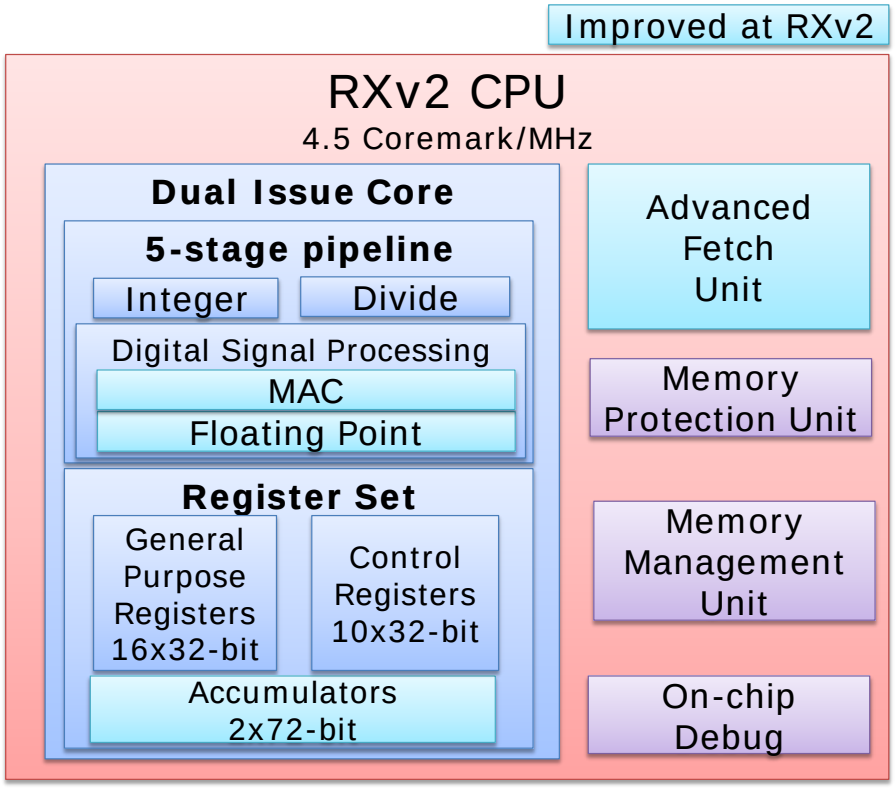


Figure 3.4 RXv2 CPU block diagram.

Item	Specification
Architecture	32bit CISC
General purpose registers	32bit x 16ch
Instructions	109 instructions Superset of RXv1 (19 new instructions)
Pipeline	5 stage, Dual Issue
DSP function	1-cycle MAC instruction (32bit x 32bit +72bit) Two Accumulators
FPU (Single Precision)	IEEE754 compliant data type and exceptions Pipeline processing
Target operating Freq.	Up to 240MHz
Memory Protection Unit	Supported
Performance (Coremark)	4.5 Coremark/MHz

Figure 3.5 Overview of RXv2 CPU core

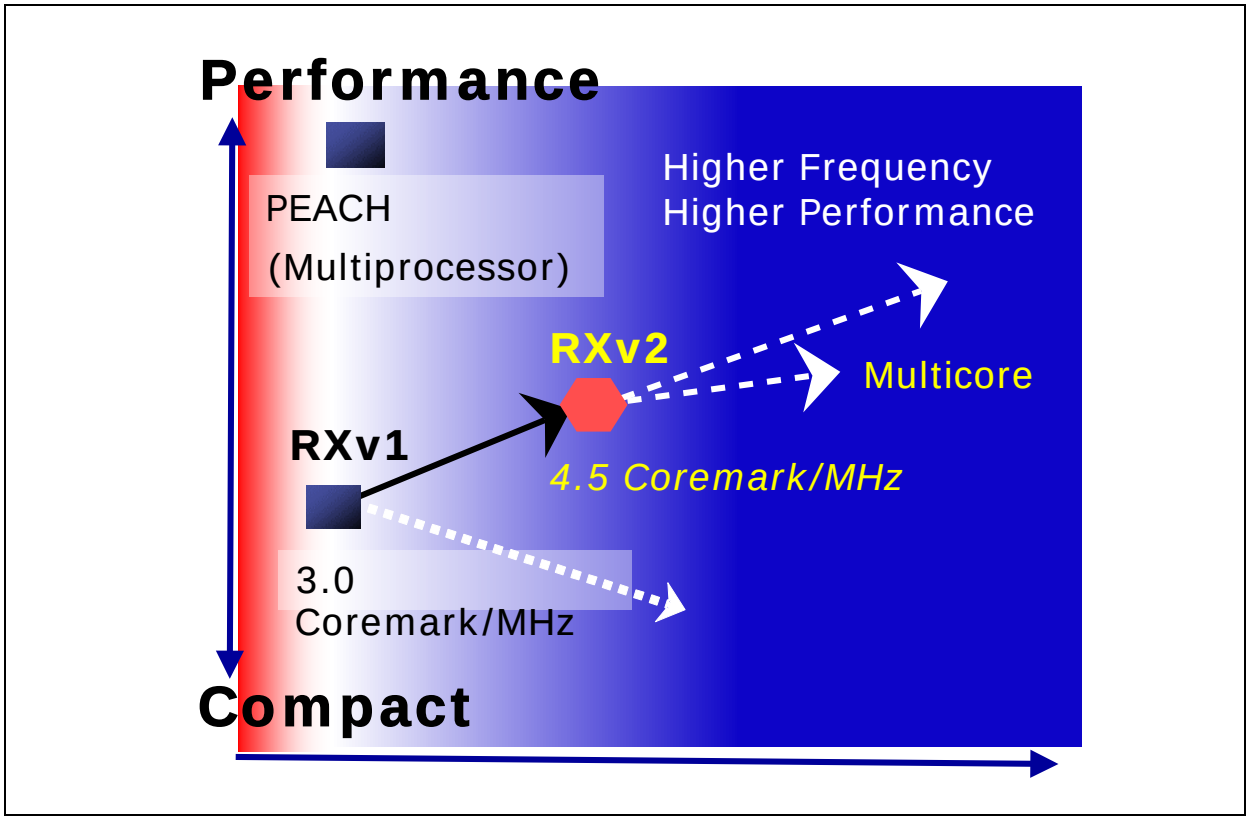


Figure 3.6 RX core road map

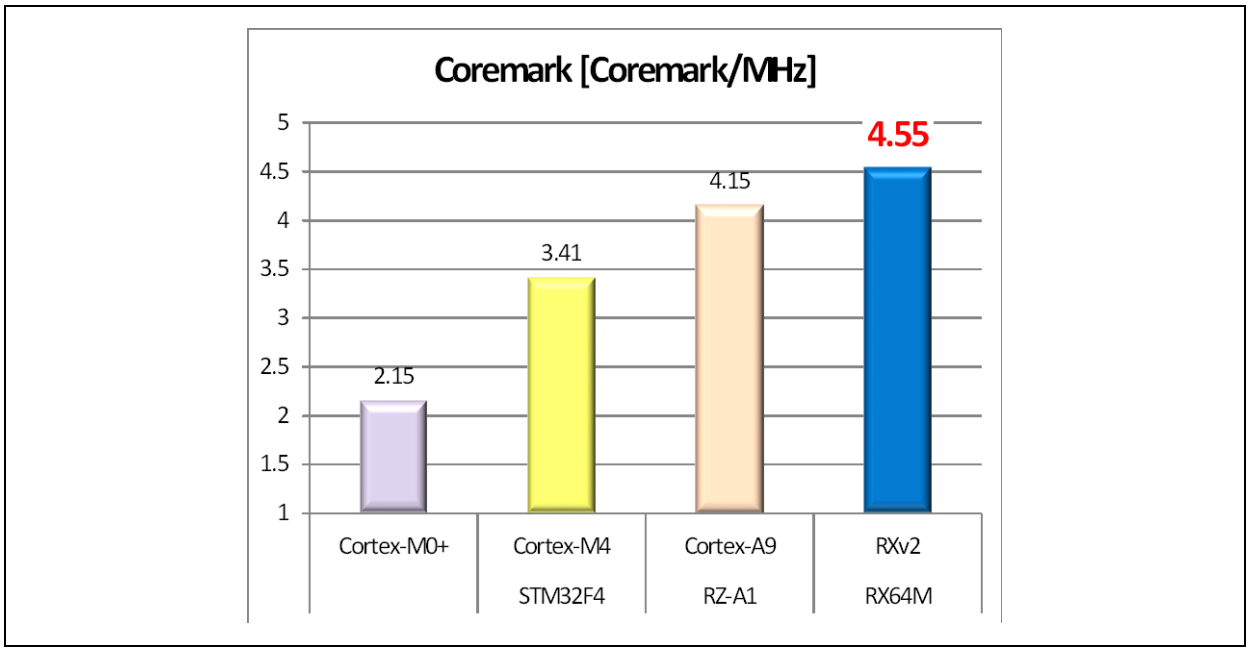


Figure 3.7 Benchmark Comparison

3.4 Core features to boost performance

3.4.1 RX Architecture Overview

In the past, modern MCUs have added DSP capabilities to create Digital Signal Controllers. Each of these MCUs has limited DSP performance and limited applications. Floating-point math has become essential in various applications such as motor control, factory automation and industrial office automation. However these applications require floating-point math to realize real-time operations. Adding a DSP/FPU is a logical step to offload compute-intensive work from MCUs. The RX CPU core has been a pioneer in the convergence of MCU and DSP/FPU in the 100MHz midrange market. Both an integrated floating-point unit (FPU) and digital signal processing (DSP) hardware enable the RX to have superior math capabilities.

3.4.2 RXv2 Pipeline Design

The first generation of RX CPU (RXv1) makes use of a single-issue, five-stage pipeline structure. RXv2 also has the same five-stage pipeline, but a dual-issue core can increase the throughput of IPC (instructions per cycle) [mips13], [burgess94], [sugure04]. Merely expanding the instruction set architecture (ISA) is not enough to boost the performance of digital signal applications. High data supply capability is crucial. Figure 3.8 shows RXv2 pipeline structure. The RXv2 executes FPU/DSP instructions and memory accesses simultaneously for high data supply. RXv2 supports a dual-issue integer, float and load/store pipeline. Additionally, the RXv2 can execute various pairs of instructions simultaneously, so instructions per cycle (IPC) are dramatically improved from the RX.

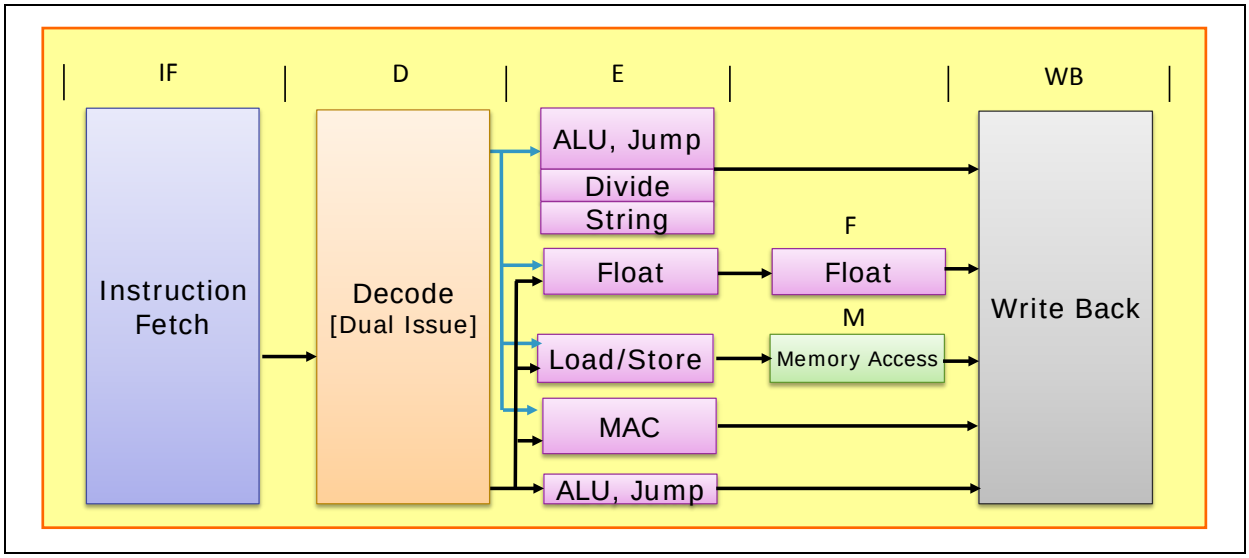


Figure 3.8 RXv2 pipeline structure

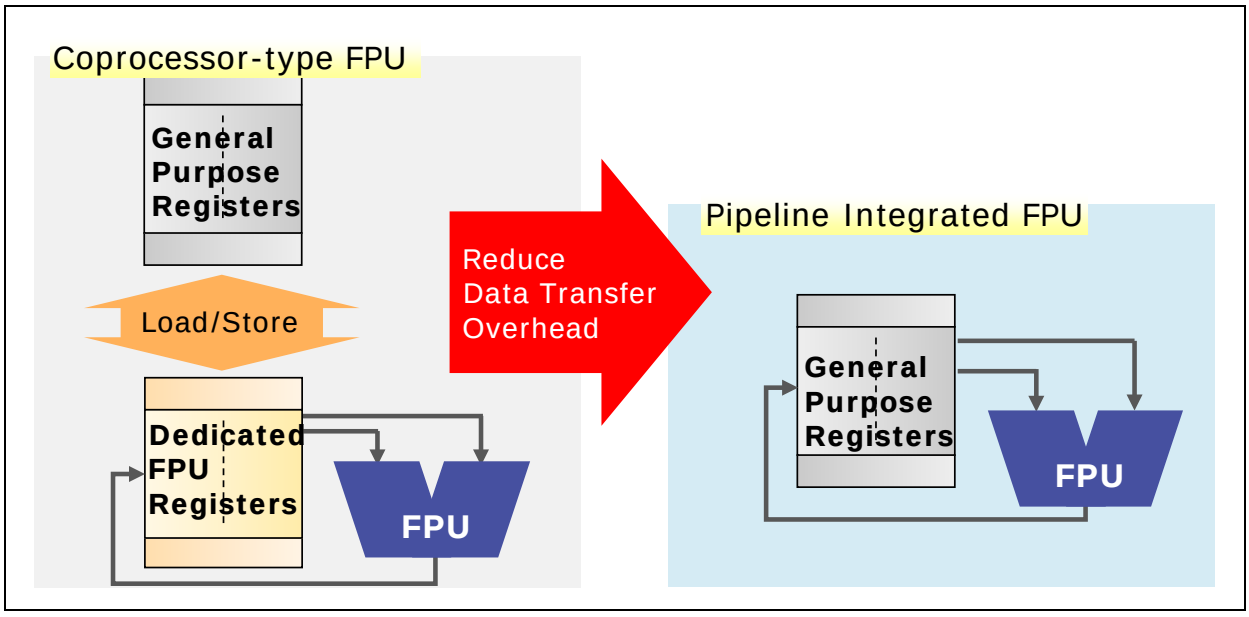


Figure 3.9 The Coprocessor-type FPU and the pipeline integrated-type FPU (proposed)

3.4.3 Pipeline integrated FPU

The most distinguishing feature of RX processors is a pipeline integrated FPU. Most MCUs have a coprocessor-type FPU, which adds inefficient FPU-dedicated registers to load and store results of operations. The pipeline integrated-type FPU used in RX processors can access general purpose

registers, which reduces data transfer overhead between the FPU registers and general purpose registers (Figure 3.9). This design can also reduce the area of the CPU core by sharing general purpose registers.

The RXv2 FPU has new instructions (SQRT, Float/Integer conversion). Furthermore, the RXv2 FPU instructions employ a three-operand format of FPU instructions to further reduce intermediate variable and waste of register assign.

The new FPU unit adopts pipeline processing to boost throughput and shortens the latency of FPU executions (FADD/FSUB 4cycles -> 2cycles, FMUL 3cycles -> 2cycles). The RXv2 processor performs most operations in one to three cycles and in single-cycle throughput. Adding the three-operand format and speeding up multiply-accumulate operations boost fast Fourier transform (FFT) and Infinite impulse response (IIR) filter performance.

FPU instructions are widely used in various applications and algorithms to achieve a high degree of numeric stability and dynamic range. We expect this upward trend of FPU use in embedded systems and even move into lower-range architectures.

3.4.4 DSP with wide accumulators

One strength of RX DSP architecture is the use of wide accumulators which allows DSP function operations to store their results in a much larger space separated from general purpose registers (Figure 3.10). The MACLO MACHI instructions multiply the 16 bits of a register by the 16 bits of another register, and add the result to the value in the accumulator. At the end of the series of multiply-accumulate operations, the RACW (Round the accumulator word) instruction rounds and saturates the value of the accumulator into 16bit. The packed 16bit data format of the DSP function operation reduces the number of data memory accesses, which improves digital signal processing performance and decreases power consumption derived from memory accesses.

RXv2 increases the number of accumulators from one to two. The accumulators have been widened from 48 bits to 72 bits. Using two accumulators boosts the performance of fixed-point DSP algorithms. For example, FIR has parallelism in that each computation result of two series of operations (coefficient * data) is stored in each accumulator, which reduces the number of data transfers from memory. The RXv2 DSP function instructions can handle 32bit and 16bit fixed point multiply and multiply-accumulate operations in a single cycle.

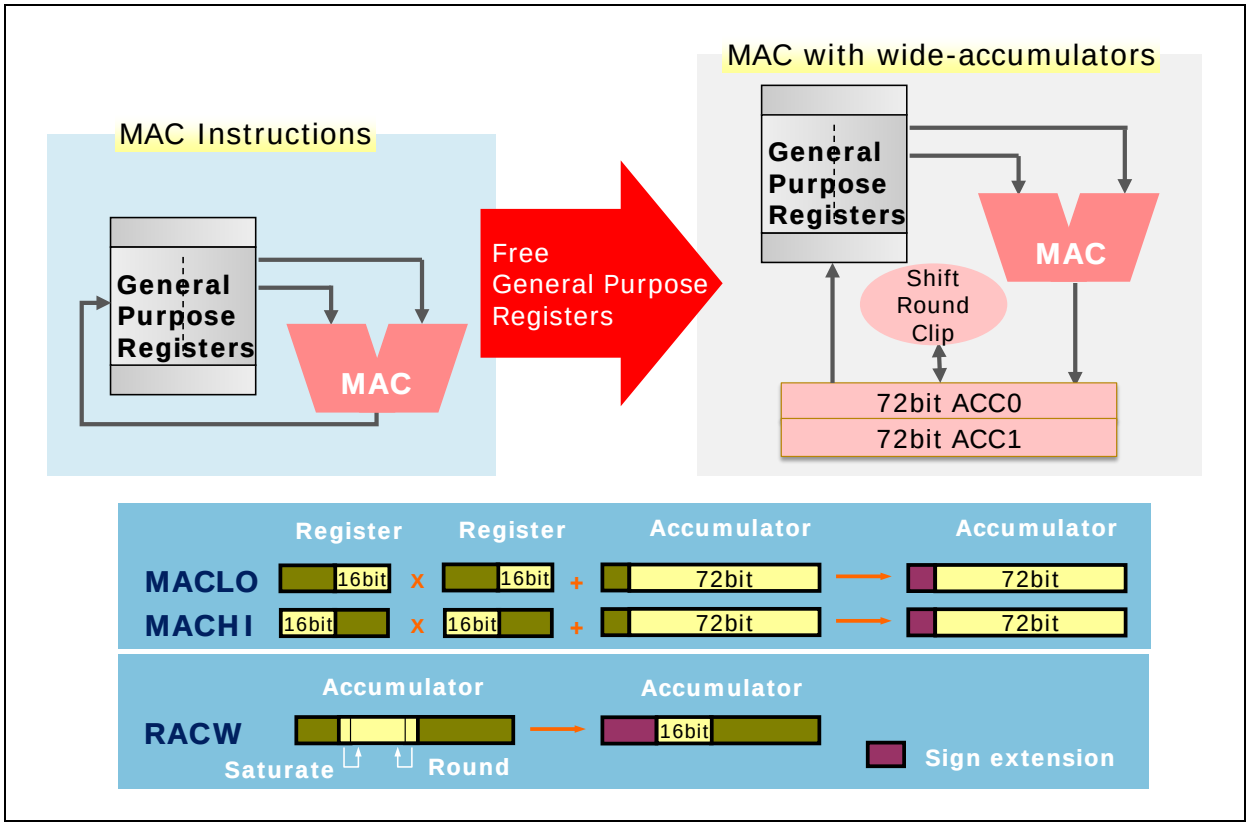


Figure 3.10 RX DSP functionality

In direct contrast to the pipeline-integrated FPU approach, the RX DSP function adopts dedicated accumulators, not general purpose registers to hold calculation results. In DSP algorithms, at the end of the series of data load and multiply accumulate operations, results are referenced. This DSP architecture is assembler-friendly and DSP library programmers can easily tune their programs because of the dedicated accumulators. Register resource shortage is a severe problem for typical MCUs because they have only sixteen general registers. Dedicate accumulators used in RX also solve this problem.

For example, when we execute 32bx32b->64b, four 32-bit registers (two source registers, two destination registers) must be used. To free general registers for other computation, the RXv2 has an EMULA, EMACA, EMSBA (32bx32b->ACC, ACC±32bx32b=ACC) instruction that stores 64-bit results in the accumulators. These instructions uses only two general registers instead of four registers.

As we mentioned before, the dual-issue pipeline exploits parallelisms in DSP operations and memory accesses, which can make full use of DSP computation ability by feeding enough data from memories. Of course some applications such as VoIP will require a dedicated DSP chip. But many sensor, speech and audio applications can be implemented by MCUs with RXv2.

3.5 Energy saving architecture

3.5.1 Embedded memory system architecture

As described in Section 3.2, Flash memory consumes a substantial portion of power in the microcontroller. Program code is located in flash memory, so the key strategy for low operating current is to reduce instruction memory accesses.

To reduce instruction memory accesses, a cache system is inevitable in today's embedded microcontrollers. Even though a top-priority issue is energy saving, it is absurd to sacrifice no-wait internal Flash memory performance by using a cache system. Reducing the power consumption of internal memories can be achieved by replacing a portion of large memories with large power consumption with memories with smaller power consumption.

3.5.2 Improving instruction fetch effectiveness

The importance of the memory hierarchy has increased with advances in the performance of processors. An embedded microcontroller has high-capacity embedded Flash memory, which is equal to the performance in 100% hit cache (Figure 3.11). However, when slower Flash is used, wait-states are required because the CPU operates faster than the native speed of the Flash memory, causing the CPU to stall, which degrades overall performance.

A typical approach is to add an instruction cache between the CPU decoder and the Flash memory. There are two reasons to add an instruction cache. First, we need to mitigate a processor-memory speed gap to feed the CPU enough instructions. Second, we also need to reduce flash memory accesses to lower the power consumption. A large fraction of the total power budget of the microcontroller is the energy consumption in the path from the FLASH memory to the CPU. Therefore, decreasing the number of flash memory accesses is crucial in reducing power consumption. The two pillars of RXv2's low power consumption are to adopt AFU and variable length ISA. First, variable length ISA delivers small dynamic code size (described in detail in Core features to make code compact), which can reduce instruction memory bandwidth. Next, if the requested instruction is contained in AFU, this request can be handled by simply reading AFU.

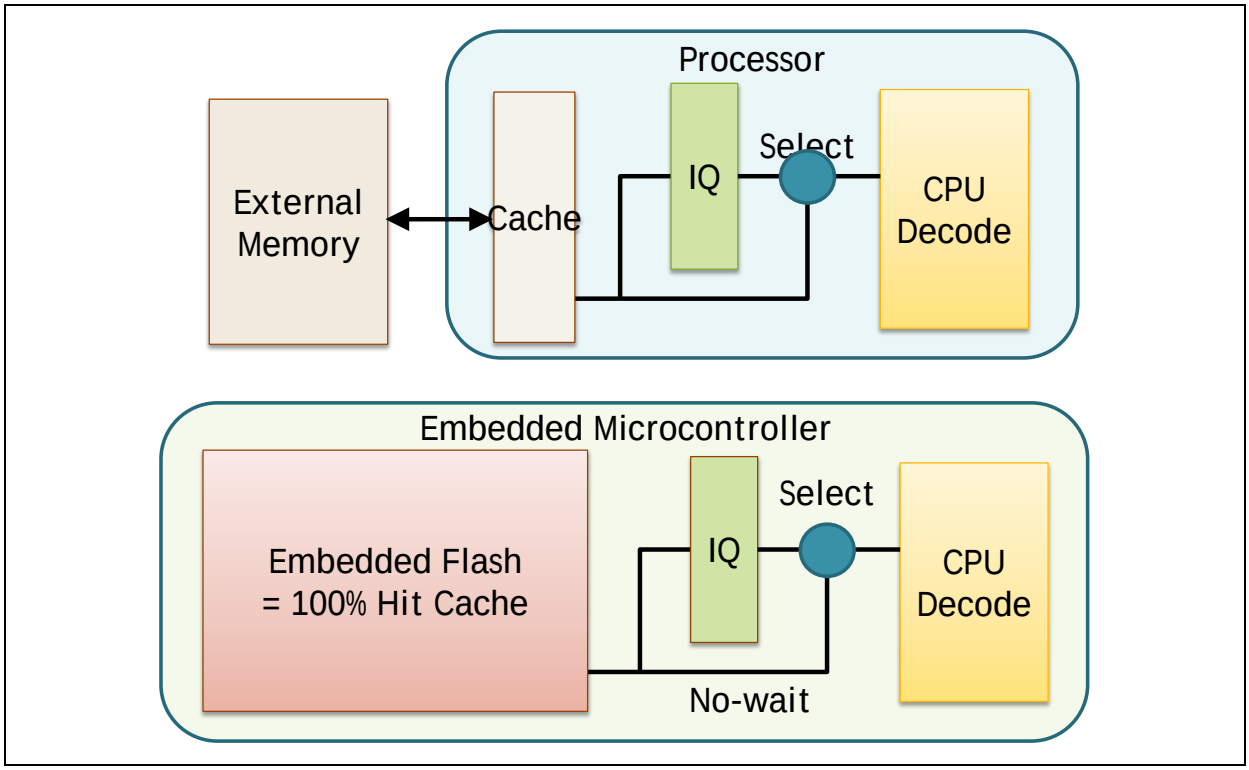


Figure 3.11 MCU: High-Capacity Internal Flash

3.5.3 AFU: Advanced Fetch Unit

AFU was added between the CPU decoder and the Flash memory (Figure 3.12). A new branch target cache [bray91] in AFU collaborates with instruction queue (IQ).

Several performance-cost trade-offs were considered in order to determine AFU structure. The RX utilizes our company's industry-leading 40nm flash technology which enables 120MHz operation with zero-wait states (Figure 3.13). Fetch latency from the Flash memory to CPU decoder directly is one cycle. Therefore, RXv2 can avoid instruction pre-fetch performance degradation. This small processor-memory speed gap allows us to concentrate on mitigating the branch penalty to improve performance. RXv2 benefits from adopting a branch target cache, which has a comparatively smaller area than that of a typical cache systems.

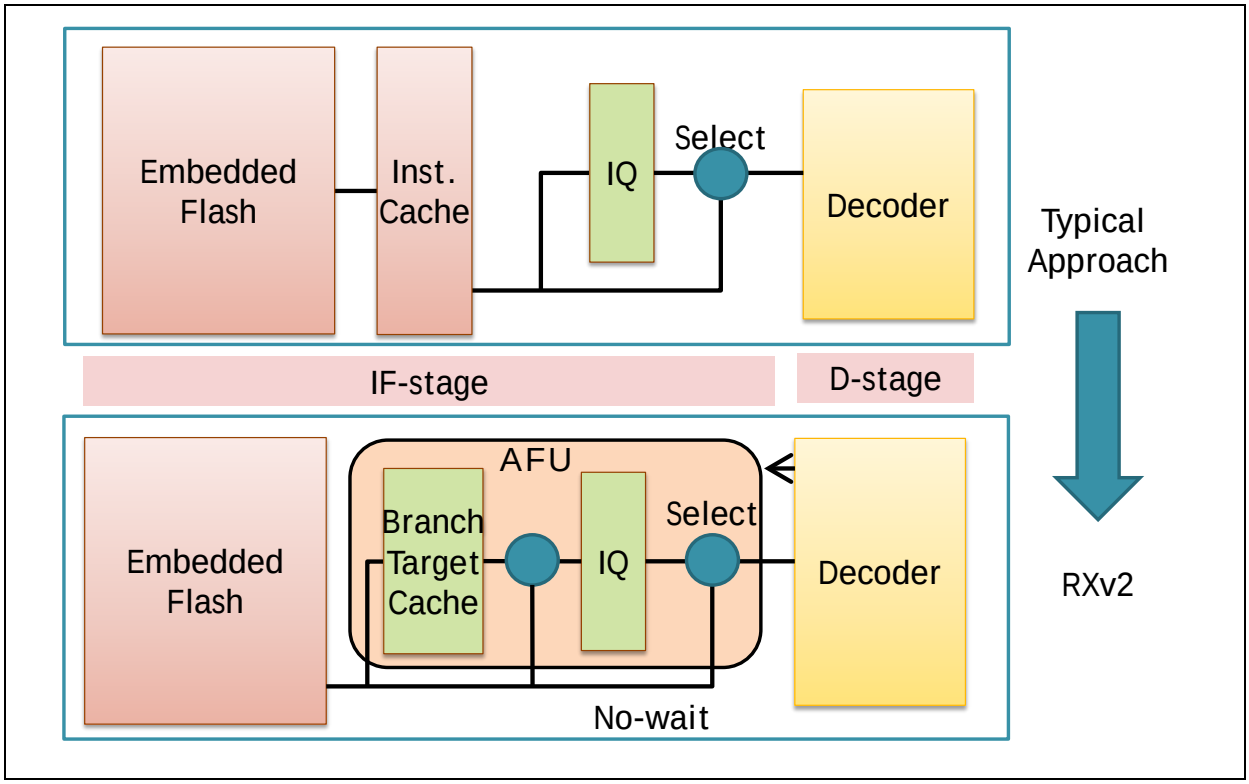


Figure 3.12 Fetch Unit for Microcontroller with Advanced Fetch Unit

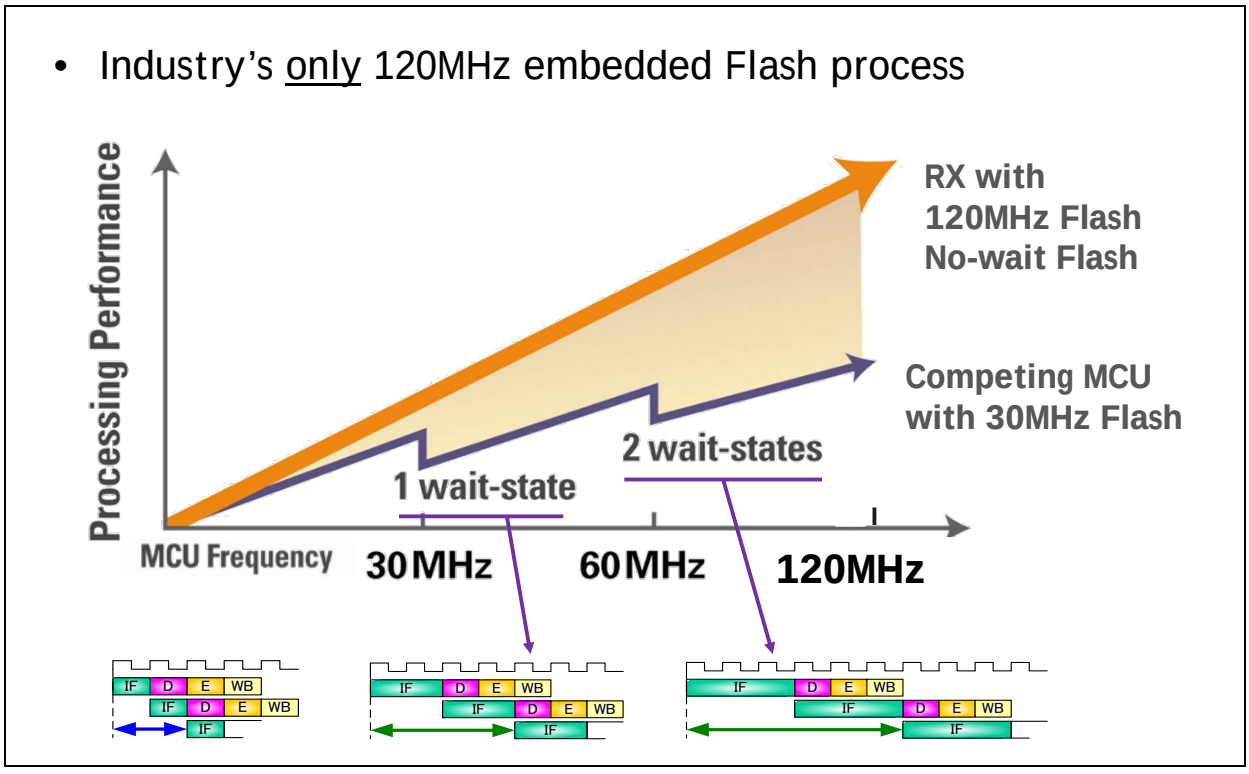


Figure 3.13 Embedded Flash processing performance

AFU consists of an IQ and a small fully-associative branch target cache with LRU replace algorithm. AFU has the following functions:

- storing branch target code (branch target cache）

- detering unused prefetching (instruction queue reuse in small loops, prefetch stop when JUMP instruction is detected.)

-replacing a cache line under dynamic priority control (8-entry LRU, Adaptive lock etc.)

AFU and zero-wait embedded Flash can reduce power consumption and improve performance because AFU reduces memory accesses and zero-wait Flash memory does not deliver cache miss penalty. AFU makes instruction buffering decisions on the fly based on an analysis of program flow. When a short loop code is detected, AFU can reuse fetched instructions in IQ and the branch target cache. IQ is locked to protect codes in the loop. This short-loop buffering reduces both branch penalties and eliminates flash memory accesses at a lower cost than that of a typical approach such as a loop-cache which stores the whole loop code.

Another efficient utilization of fetched data from memory is “fast short forward branch”. The CPU core sends the distance to the branch target. If IQ finds the target code in IQ, CPU fetches codes from IQ without a pipeline flash and memory accesses. This technique improves if-then-else control flow in cycle performance and power dissipation.

AFU of RXv2 improves its processing performance by 6% in Coremark [halfhill09] with zero-wait Flash memory.

3.5.4 Processor performance and power consumption

Differences in performance appear when benchmarking DSP programs that include numeric operation function such as filter programs. Figure 3.14 illustrates that the DSP of RXv2 has contributed mainly to performance improvements compared to RXv1 when executing FFT, IIR and Matrix under 16bit fixed point, 32bit fixed point and float conditions. The RXv2 processor delivers 1.9 – 3.7 the cycle performance of the RXv1. As a result, the RXv2 provides 1.5 – 3.4 the cycle performance of a RISC-

based processor. RXv2 achieves performance as high as commonly used DSP and improves far more as a DSP microcontroller.

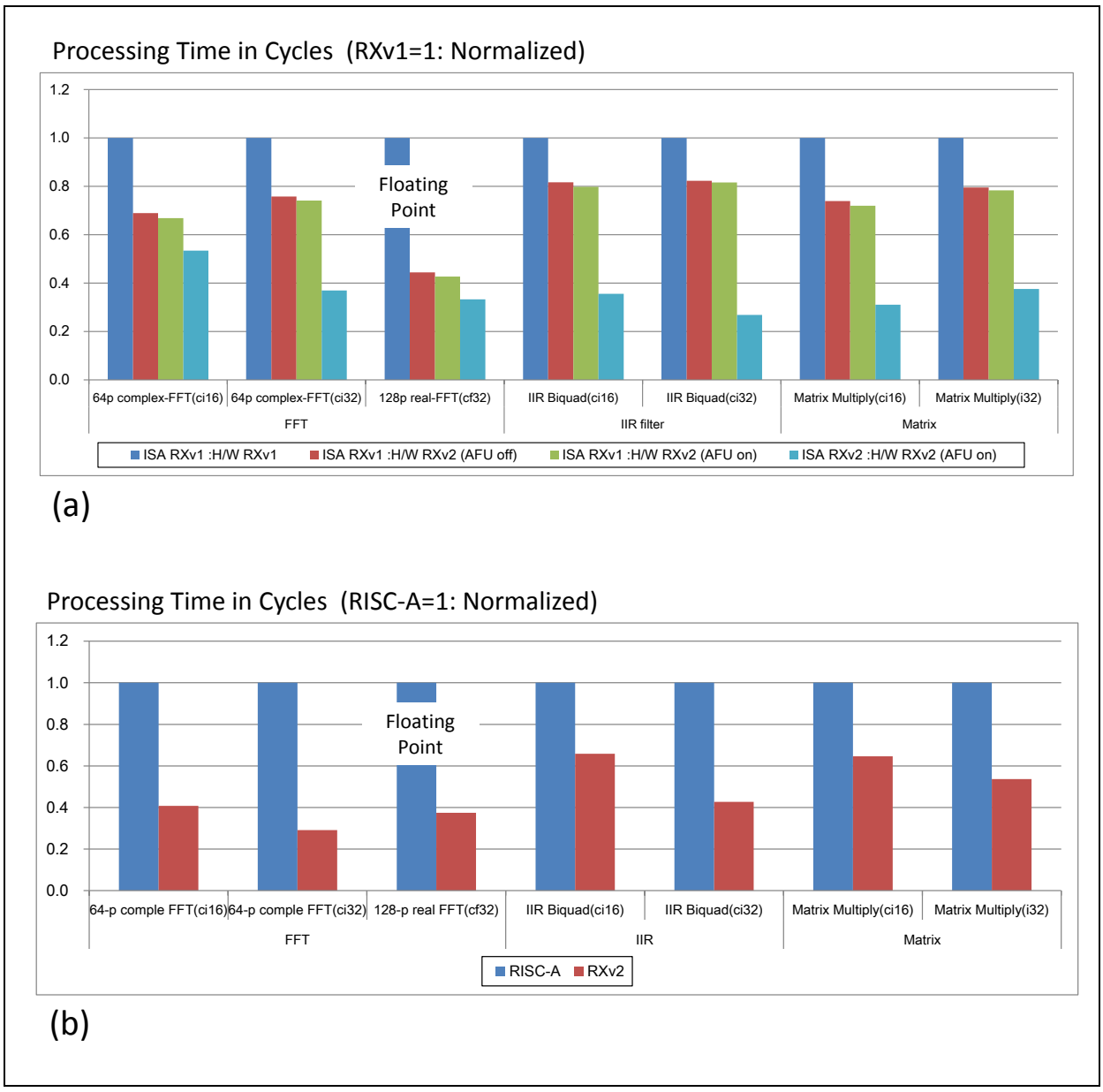


Figure 3.14 Benchmark Results of DSP Algorithm programs such as FFT, IIR filter and Matrix under zero-wait flash memory access.

Breakdown of the RXv2 performance enhancement from the RXv1 (a), Performance comparison to a RISC-based processor (Cortex-M4) (b).

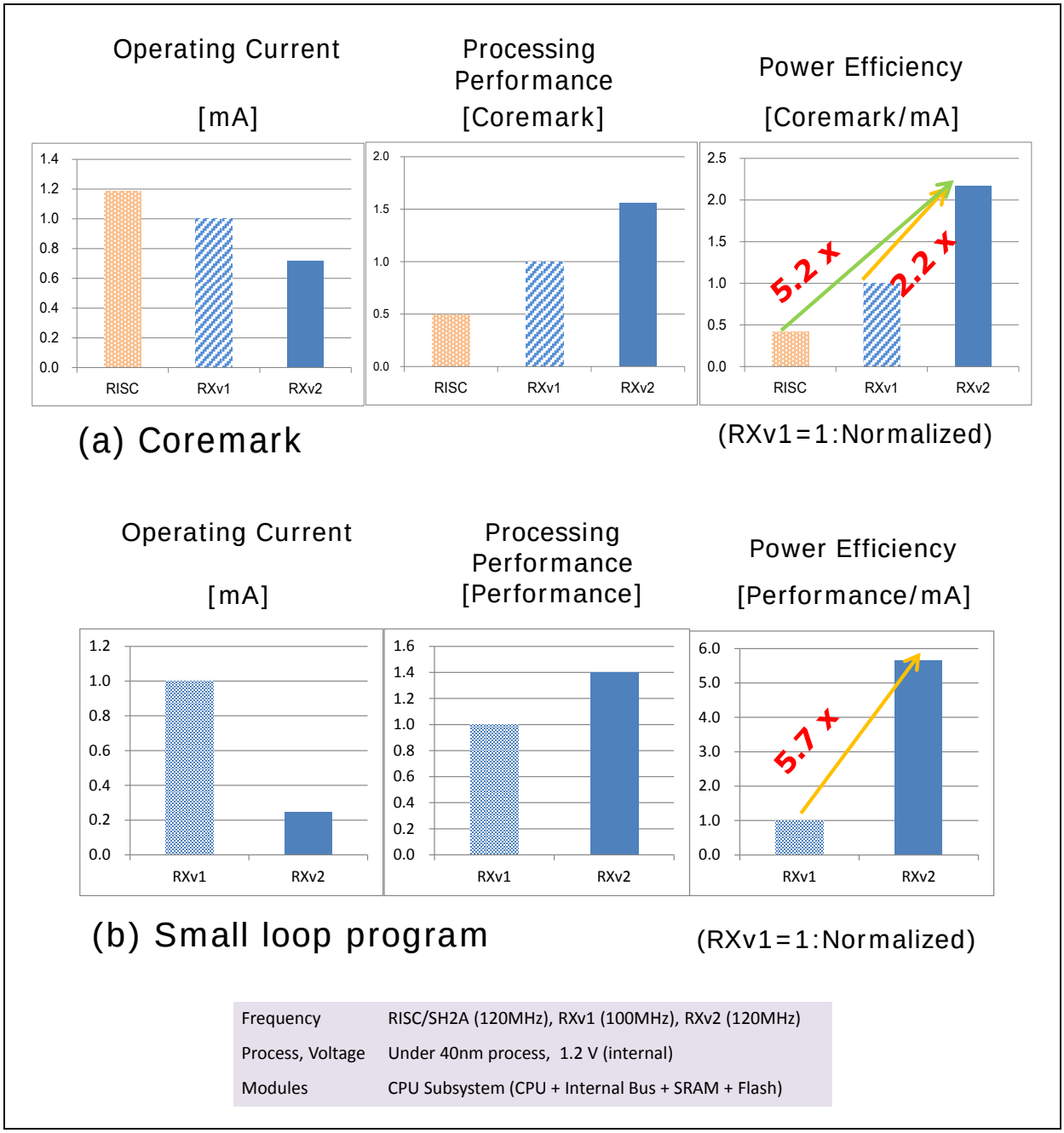


Figure 3.15 Performance Comparison of RXv2 with RXv1 and a RISC processor

We evaluated the performance and power dissipation of the RXv2 device (120MHz) in a simulation with gate-level power analysis using actual loading. RXv2 reaches 4.5 Coremark per MHz. RXv2 achieved a 50% - 150% improvement in various performance categories compared to existing products. The result is performance that outperforms the competing RISC microcontrollers. Figure 3.15 illustrates the performance advantage of the RXv2 device compared to the RX device. The RXv2 processor

delivers approximately more than 2.2 – 5.7x the power efficiency of the RXv1 in executing Coremark and a small loop program (a power evaluation program). Figure 3.15(a) also illustrates that the RXv2 processor achieves 5.2x the power efficiency of a RISC processor (SH-2A), which shows that the performance of RXv2 is sufficient to fulfill the performance requirement for current and future embedded systems. The decrease of the number of Flash memory accesses by AFU is a dominant determiner of reducing power consumption in benchmarks. AFU reduces the number of Flash memory accesses by 25%.

3.5.5 Core features to make code compact

Small memory size is inevitable in embedded applications because of their severe cost constraints, especially in MCUs with on-chip memories [bunda93]. Furthermore, program compression has a benefit for energy saving by reducing the number of bit fetched from memories. Several RISC architecture machines offered a mix of 16bit and 32bit instructions to compensate for the disadvantage of the code density. Despite the effort to mitigate this penalty, RISC MCUs still have basically inferior code density because of the lesser work accomplished per instruction [mips13], [sugure04], [xarm10].

3.6 RX instruction set architecture

3.6.1 Overview of Instruction set

RX has a compact architecture with 109 carefully-selected instructions, which is equal to the number of instructions in a RISC-based architecture (Figure 3.16). The RX instruction set consists of eight types of instructions: arithmetic/logic instructions, floating-point operation instructions, data transfer instructions, branch instructions, bit manipulation instructions, string manipulation instructions, system control instructions and DSP function instructions [renesas13].

To achieve a high performance, high code density and low power system, the RX instruction set architecture uses a variable-length instruction format (1byte – 8byte). The RX instructions are variable in length at the byte level with the exact instruction length dependent on the data size and addressing mode used, which increases instruction code density and reduces the amount of data fetched from memory per operation.

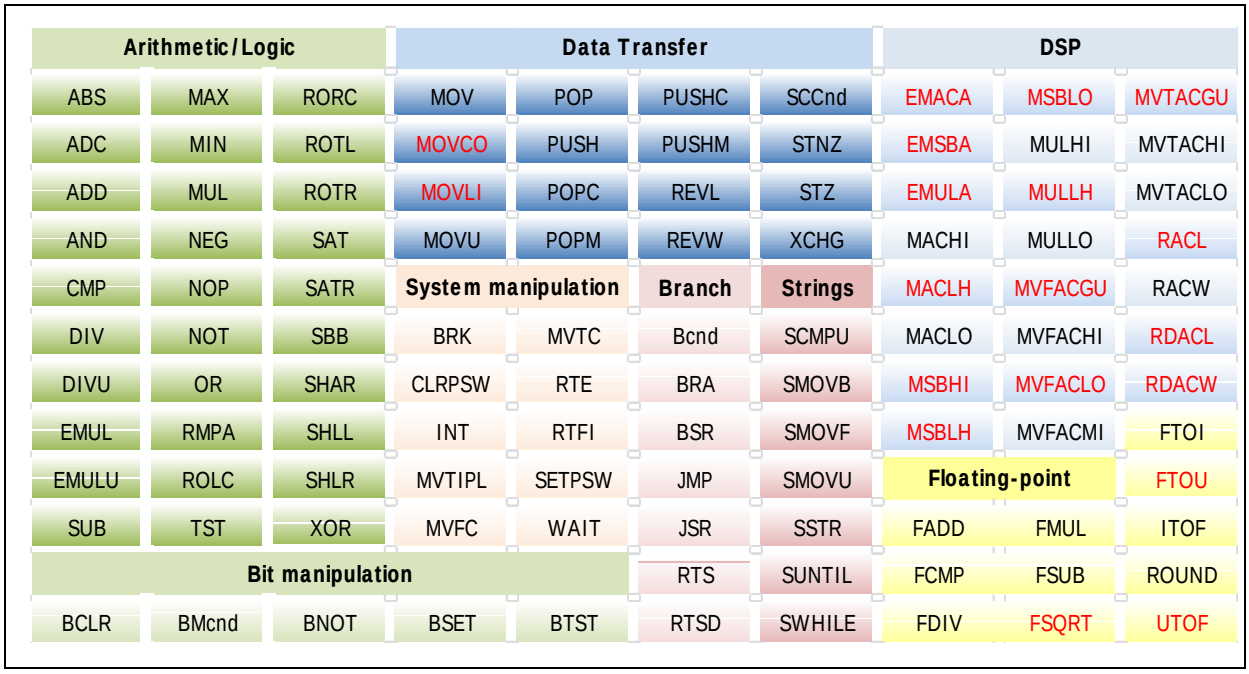


Figure 3.16 RX instruction set architecture

3.6.2 Optimized op codes leads to superior code density

CISC architecture inherently has the advantage in terms of the work accomplished per instruction and that always translates to a code-density win [hennessy06]. RX architecture stretches that advantage via a flexible instruction set architecture that can encode some instructions in as little as one byte.

At the other end of the spectrum, instructions can takes as many as eight bytes when the instruction needs to specify a large address range or large data values that are unable to handle 32bit instructions.

We did a further analysis of real application code to discern the most frequently used instructions and further reduce code size (Figure 3.17). We determined the most frequently used instructions are assigned to shorter instruction codes, from one-byte to four-byte instructions. We also added addressing modes and included a three-operand instruction format to optimize code density.

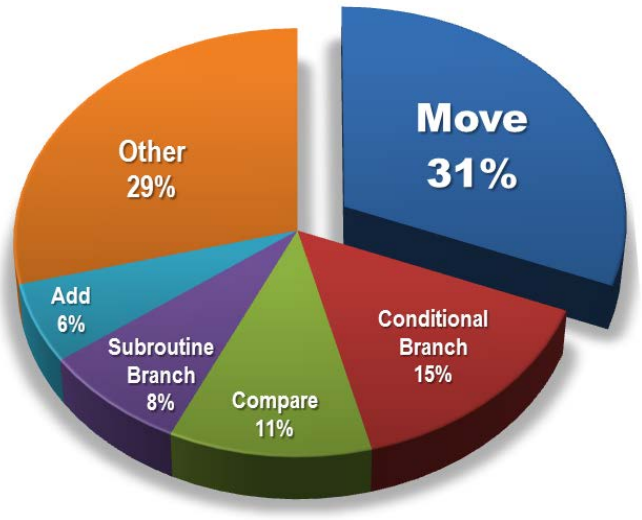


Figure 3.17 Analysis of Instruction Frequency

Assign short code to frequently-used instruction

1byte length Instruction (Frequently-used "Conditional Branch")

Relative condition branch:BEQ,BNE
Unconditional relative branch:BRA

2byte-length Instructions (Frequently-used "Data Transfer and Comparison instructions")

Data Transfer	:MOV	(register to register, memory to memory, Load, Store)
Comparison	:CMP	(register to register, register to immediate)
Addition	:ADD	(register + register, register + immediate)
Subroutine branch	:BSR	
Multiplication	:MUL	(register x register)

3byte-length Instructions (Frequently-used "Arithmetic and logical instructions")

Division	:DIV	(register / register)
Multiply-accumulate	:EMAC	(register x register)
Floating-point addition	:FADD	(register + register)
Floating-point multiplication	:FMUL	(register x register)

Figure 3.18 Byte assignment of RX Instruction format

Move (MOV) instructions were found to be the most frequently used instructions, accounting for more than 30% of all operations. Conditional branch instructions were the next most frequent, followed by Compare instructions, Subroutine Branch instructions and Add (ADD). Move instructions therefore received the most enhancements in terms of additional addressing modes, and the ability to automatically increment and decrement values stored in registers. The next most frequent instructions were also shortened. Add instructions were both shortened and enhanced with a three-operand format. Figure 3.18 illustrates RX instruction code that assigns shorter code to frequently used instructions.

The instruction set is decidedly CISC in nature and is a primary factor in the code density and performance advantage. The following is a detailed look at some instructions to illustrate the benefits of the architecture.

3.6.3 Data Transfer instruction

MOV instruction illustrates RX variable length instruction set advantages. Unlike in the RISC case where instructions are generally fixed in length, the CISC counterpart is variable in length and far more flexible. The MOV instruction is the most frequently used instruction in real application code, therefore its binary code assignment instruction is important for execution performance and code size compression.

The RX MOV instruction can handle the following six types of data transfer from source to destination.

src	dest	Function
Immediate value	Register	Transfers the immediate value to the register. When the immediate value is specified in less than 32 bits, it is transferred to the register after being zero-extended if specified as #UIMM and sign-extended if specified as #SIMM.
Immediate value	Memory location	Transfers the immediate value to the memory location in the specified size. When the immediate value is specified with a width in bits smaller than the specified size, it is transferred to the memory location after being zero-extended if specified as #UIMM and sign-extended if specified as #SIMM.
Register	Register	Transfers the data in the source register (src) to the destination register (dest). When the size specifier is .B, the data is transferred to the register (dest) after the byte of data in the LSB of the register (src) has been sign-extended to form a longword of data. When the size specifier is .W, the data is transferred to the register (dest) after the word of data from the LSB end of the register (src) has bee sign-extended to form a longword of data.
Register	Memory location	Transfers the data in the register to the memory location. When the size specifier is .B, the byte of data in the LSB of the register is transferred. When the size specifier is .W, the word of data from the LSB end of the register is transferred.
Memory location	Register	Transfers the data at the memory location to the register. When the size specifier is .B or .W, the data at the memory location are sign-extended to form a longword, which is transferred to the register.
Memory location	Memory location	Transfers the data with the specified size at the source memory location (src) to the specified size at the destination memory location (dest).

Instruction Format

Syntax	Size	Processing Size	Operand src	Operand dest	Code Size (Byte)
MOV.size src, dest	Store (short format)				
	B/W/L	size	Rs (Rs = R0 to R7)	dsp:5[Rd]*1 (Rd = R0 to R7)	2
	Load (short format)				
	B/W/L	L	dsp:5[Rs]*1 (Rs = R0 to R7)	Rd (Rd = R0 to R7)	2
	Set immediate value to register (short format)				
	L	L	#UIMM:4	Rd	2

Syntax	Size	Processing Size	Operand src	Operand dest	Code Size (Byte)
MOV.size src, dest	Set immediate value to memory location (short format)				
	B	B	#IMM:8	dsp:5[Rd]*1 (Rd = R0 to R7)	3
	W/L	size	#UIMM:8	dsp:5[Rd]*1 (Rd = R0 to R7)	3
	Set immediate value to register				
	L	L	#UIMM:8*2	Rd	3
	L	L	#SIMM:8*2	Rd	3
	L	L	#SIMM:16	Rd	4
	L	L	#SIMM:24	Rd	5
	L	L	#IMM:32	Rd	6
	Data transfer between registers (sign extension)				
	B/W	L	Rs	Rd	2
	Data transfer between registers (no sign extension)				
	L	L	Rs	Rd	2
	Set immediate value to memory location				
	B	B	#IMM:8	[Rd]	3
	B	B	#IMM:8	dsp:8[Rd]*1	4
	B	B	#IMM:8	dsp:16[Rd]*1	5
	W	W	#SIMM:8	[Rd]	3
	W	W	#SIMM:8	dsp:8[Rd]*1	4
	W	W	#SIMM:8	dsp:16[Rd]*1	5
	W	W	#IMM:16	[Rd]	4
	W	W	#IMM:16	dsp:8[Rd]*1	5
	W	W	#IMM:16	dsp:16[Rd]*1	6
	L	L	#SIMM:8	[Rd]	3
	L	L	#SIMM:8	dsp:8[Rd]*1	4
	L	L	#SIMM:8	dsp:16 [Rd]*1	5
	L	L	#SIMM:16	[Rd]	4
	L	L	#SIMM:16	dsp:8[Rd]*1	5
	L	L	#SIMM:16	dsp:16 [Rd]*1	6
	L	L	#SIMM:24	[Rd]	5
	L	L	#SIMM:24	dsp:8[Rd]*1	6
	L	L	#SIMM:24	dsp:16 [Rd]*1	7
	L	L	#IMM:32	[Rd]	6
	L	L	#IMM:32	dsp:8[Rd]*1	7
	L	L	#IMM:32	dsp:16 [Rd]*1	8
	Load				
	B/W/L	L	[Rs]	Rd	2
	B/W/L	L	dsp:8[Rs]*1	Rd	3
	B/W/L	L	dsp:16[Rs]*1	Rd	4
	B/W/L	L	[Ri, Rb]	Rd	3
	Store				
	B/W/L	size	Rs	[Rd]	2
	B/W/L	size	Rs	dsp:8[Rd]*1	3
	B/W/L	size	Rs	dsp:16[Rd]*1	4
	B/W/L	size	Rs	[Ri, Rb]	3

Syntax	Size	Processing Size	Operand src	Operand dest	Code Size (Byte)
MOV.size src, dest	Data transfer between memory locations				
	B/W/L	size	[Rs]	[Rd]	2
	B/W/L	size	[Rs]	dsp:8[Rd]*1	3
	B/W/L	size	[Rs]	dsp:16[Rd]*1	4
	B/W/L	size	dsp:8[Rs]*1	[Rd]	3
	B/W/L	size	dsp:8[Rs]*1	dsp:8[Rd]*1	4
	B/W/L	size	dsp:8[Rs]*1	dsp:16[Rd]*1	5
	B/W/L	size	dsp:16[Rs]*1	[Rd]	4
	B/W/L	size	dsp:16[Rs]*1	dsp:8[Rd]*1	5
	B/W/L	size	dsp:16[Rs]*1	dsp:16[Rd]*1	6
	Store with post-increment*3				
	B/W/L	size	Rs	[Rd+]	3
	Store with pre-decrement*3				
	B/W/L	size	Rs	[–Rd]	3
	Load with post-increment*4				
	B/W/L	L	[Rs+]	Rd	3
	Load with pre-decrement*4				
	B/W/L	L	[–Rs]	Rd	3

RX Family RXv2 Instruction Set Architecture User's Manual: Software
(copyright ©2013 Renesas Electronics Corporation)

The RX MOV instruction supports various operand formats and addressing modes. Therefore, programmers and compilers can reduce code size effectively by using appropriate instruction format for the most commonly used instance of MOV.

1. Wide variety of immediate field of MOV instructions.

Immediate value can be selected directly from 8bits, 16bits, 24bits and 32bits with signed and unsigned types. This feature provides the benefit of eliminating operations that set an immediate value to a register.

2. Short-format MOV instructions.

There are a number of short-format MOV instructions that are used most frequently and those instructions are 2 or 3 bytes in length. Consider a typical example of this instruction:

MOV.L Rs, dsp:5[Rd]

This instruction transfers a 32bit value from a source register (Rs) to a memory location that is defined by the location stored in a destination register (Rd) added to a 5bit displacement value. The effective address of the operand is the least significant 32 bits of the sum of the displacement (dsp:5) value, after zero extension to 32 bits and multiplication by 4, and the value in the specified register.

The MOV instruction and the L designation for a long word requires 5 bits in the instruction code. By applying limitation on use of general-purpose registers into half of full 16 registers (R0 – R7), both of the register designations are compacted into 3 bits. Therefore, this MOV instruction is encoded only in two bytes. For modern compilers or hand-coded assembly language, this limitation is minor. In comparison, consider the same instruction with full access to 16 registers and with the range afforded by a 16bit displacement value. That instruction would double in size to 4 bytes. When clever encoding can reduce a powerful instruction from 4 bytes to 2 bytes, the inherent advantage of a CISC instruction set is greatly enhanced.

3.6.4 1byte conditional branch instruction

The conditional branch instruction plays a big part in code density; therefore, it is always encoded in the minimum possible length.

Function

- This instruction makes the flow of relative branch to the location indicated by src when the condition specified by *Cnd* is true; if the condition is false, branching does not proceed.
- The following table lists the types of B*Cnd*.

B*Cnd*		Condition	Expression	B*Cnd*		Condition	Expression
BGEU, BC	C == 1	Equal to or greater than/ C flag is 1	≤	BLTU, BNC	C == 0	Less than/ C flag is 0	>
BEQ, BZ	Z == 1	Equal to/Z flag is 1	=	BNE, BNZ	Z == 0	Not equal to/Z flag is 0	≠
BGTU	(C & ~Z) == 1	Greater than	<	BLEU	(C & ~Z) == 0	Equal to or less than	≥
BPZ	S == 0	Positive or zero	0 ≤	BN	S == 1	Negative	0 >
BGE	(S ^ O) == 0	Equal to or greater than as signed integer	≤	BLE	((S ^ O) | Z) == 1	Equal to or less than as signed integer	≥
BGT	((S ^ O) | Z) == 0	Greater than as signed integer	<	BLT	(S ^ O) == 1	Less than as signed integer	>
BO	O == 1	O flag is 1		BNO	O == 0	O flag is 0	

Instruction Format

Syntax	Length	Operand		Code Size (Byte)
		src	Range of pcdsp	
(1) BEQ.S src	S	pcdsp:3	3 ≤ pcdsp ≤ 10	1
(2) BNE.S src	S	pcdsp:3	3 ≤ pcdsp ≤ 10	1
(3) B*Cnd*.B src	B	pcdsp:8	–128 ≤ pcdsp ≤ 127	2
(4) BEQ.W src	W	pcdsp:16	–32768 ≤ pcdsp ≤ 32767	3
(5) BNE.W src	W	pcdsp:16	–32768 ≤ pcdsp ≤ 32767	3

(copyright ©2013 Renesas Electronics Corporation)

The conditional branch instructions can comprise 15% of the instructions in a typical program – second in frequency only to the MOV instruction. There are conditional branches based on greater than or less than operators, and based on positive, zero, or negative values to offer flexibility to the programmer.

The RX instruction set encodes such instructions in as compact a length as a single byte on Conditional Branch instructions including BEQ (branch if equal), BNE (branch if not equal) and BRA (branch always).

Consider the following instructions:

BEQ label1

This BEQ instruction results in a branch to a memory location if the processor's Z flag is set to a value of "1". The instruction length is determined by the difference between the memory location of the BEQ instruction that is stored in the program counter relative to the branch location defined either by a label.

According to program analysis in various applications, most branch distances are within the general vicinity and branch directions are forward in order to execute if-then-else program codes. Therefore RX instruction architecture encodes the branch forward instructions (BEQ, BNE and BRA) with address distance of 10 byte or less in one byte.

The RX supports more branch operations including both in the forward and reverse directions from a memory address perspective. A two-byte instantiation can control forward or reverse branches in the range of -128 to +127 relative to the program counter. And the three-byte version stretches the range to -32768 to +32767.

The result is better performance on application code that occupies a smaller memory footprint.

3.6.5 Compare instruction

The code analysis revealed that the CMP instruction was the third most frequently used instruction. The instruction comprised 11% of the sample code. Moreover, the design team found a way to cut the instruction length in half relative to other CISC MCUs – yielding a 2-byte CMP instruction.

Instruction Format

Syntax	Processing Size	Operand src	Operand src2	Code Size (Byte)
CMP src, src2	L	#UIMM:4	Rs	2
	L	#UIMM:8*1	Rs	3
	L	#SIMM:8*1	Rs	3
	L	#SIMM:16	Rs	4
	L	#SIMM:24	Rs	5
	L	#IMM:32	Rs	6
	L	Rs	Rs2	2
	L	[Rs].memex	Rs2	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex*2	Rs2	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex*2	Rs2	4 (memex == UB) 5 (memex != UB)

RX Family RXv2 Instruction Set Architecture User's Manual: Software
(copyright ©2013 Renesas Electronics Corporation)

The CMP instruction is variable in length depending on the type of the operands. It is a tremendous advantage of a CISC instruction set to be able to use immediate values and operands stored in memory with instructions such as CMP. RISC requires that both operands be stored in registers.

There are three different ways to use CMP with a 2-byte instruction length. Register to register compares are always 2 bytes. But the RX also supports both compares using immediate values and operands from memory with 2-byte instructions.

Consider the following instruction:

CMP #7, R2.

The instruction compares an immediate value 7 with the data stored in R2. As long as the immediate value is 4 bits or less in size, the instruction requires only 2 bytes. But the implementation provides the flexibility to use immediate values as wide as 32bits. The instruction lengths scales from 2 to 6 bytes to support 4, 8, 16, 24, and 32bit immediate values.

The CMP instruction can also be implemented in 2 bytes for memory-to-register compare operations. Consider the following instruction:

CMP [R2], R3.

This instruction comparing the operand pointed to by R2 with the one stored in R3 always requires only 2 bytes. Again, however, the implementation offers flexibility. The instruction can be used with a

displacement value from the memory location stored in the register. The instruction length scales to 5 bytes to support 16bit displacements.

Almost all CISC architectures offer the flexibility illustrated here with CMP, which is a huge advantage of CISC relative to RISC.

3.6.6 3-operand instruction

The instruction implementation in the RX offers a variety of addressing modes and even a three-operand format. ADD is the fifth most regularly occurring instruction, making up 6% of the instructions in a typical program; therefore it was targeted for special treatment.

Consider the instructions:

ADD R1, R2, R3

and

ADD R1, R2.

The benefit of three-operand format is not to overwrite one of the source operands and program code can reuse the value of the source register. Both ADD instructions add the values in R1 and R2. The three-operand format stores the result in R3. The two-operand stores the result in R2 – overwriting one of the source operands. With embedded RISC processors that only support the two-operand format, there are times when an extra move instruction is required before or after the ADD because the program needs to preserve the data in the destination register before the ADD takes place as well as preserving the summed result.

The RX ADD instruction offers additional flexibility in that the first of the three operands can be an immediate value. RISC architectures would always have to load such an immediate value prior to executing the ADD.

The three-operand ADD is encoded in three bytes when each of the operands is a register. With an 8bit immediate value, the instruction still only requires 3 bytes. Larger immediate values can stretch the instruction length to 4, 5, or 6 bytes.

Like many CISC processors, the RX can encode a two-operand ADD instruction in two bytes when both operands are registers. But we devised 2-byte instructions both for ADDs involving an immediate value or data from a memory location.

A two-operand ADD instruction, in which the first operand is a 4bit immediate value and the second operand is a register, requires only two bytes. That is half the size of typical immediate-value ADD instructions. Larger immediate values stretch the instruction length to 3, 4, 5, or 6 bytes.

A two-operand ADD instruction in which the first operand is data in a memory location that is pointed to by a register also requires only two bytes. More complex versions can use a register storing a memory location, and an offset from that location. Such relative-addressing modes can result in 3-, 4-, or 5-byte instructions. For systems, the result is smaller code, less memory and therefore lower cost, and better performance.

Instruction Format

Syntax		Processing Size	Operand src	Operand src2	Operand dest	Code Size (Byte)
(1) ADD	src, dest	L	#UIMM:4	–	Rd	2
		L	#SIMM:8	–	Rd	3
		L	#SIMM:16	–	Rd	4
		L	#SIMM:24	–	Rd	5
		L	#IMM:32	–	Rd	6
		L	Rs	–	Rd	2
		L	[Rs].memex	–	Rd	2 (memex == UB) 3 (memex != UB)
		L	dsp:8[Rs].memex*	–	Rd	3 (memex == UB) 4 (memex != UB)
		L	dsp:16[Rs].memex*	–	Rd	4 (memex == UB) 5 (memex != UB)
(2) ADD	src, src2, dest	L	#SIMM:8	Rs	Rd	3
		L	#SIMM:16	Rs	Rd	4
		L	#SIMM:24	Rs	Rd	5
		L	#IMM:32	Rs	Rd	6
		L	Rs	Rs2	Rd	3

RX Family RXv2 Instruction Set Architecture User's Manual: Software
(copyright ©2013 Renesas Electronics Corporation)

3.6.7 Registers

We have investigated how general-purpose register configurations and operational codes for instructions are related. The number of registers in an instruction set architecture has a direct impact on code size because the register number bit field requires more bits in the operation codes to encode

support for more registers. But more registers are almost always better from a performance perspective. A greater number of registers eases register allocation, which means that the target program spends far less time shuffling data between memory and registers. Even CISC architectures that can directly operate on operands stored in memory still feature faster execution when operating on registers.

To perform an in-depth analysis on the optimal size of a register file, we ran benchmark tests using real code that was central to target markets such as office automation and consumer, industrial, and automotive fields. Figure 3.19 shows the analysis of the register file. The vertical axis on the left represents the relative amount of hardware volume needed to support the register file. Red curves on the right indicate the code size attributable to the number of registers. The green curve indicates the register-specified bit number in operation code.

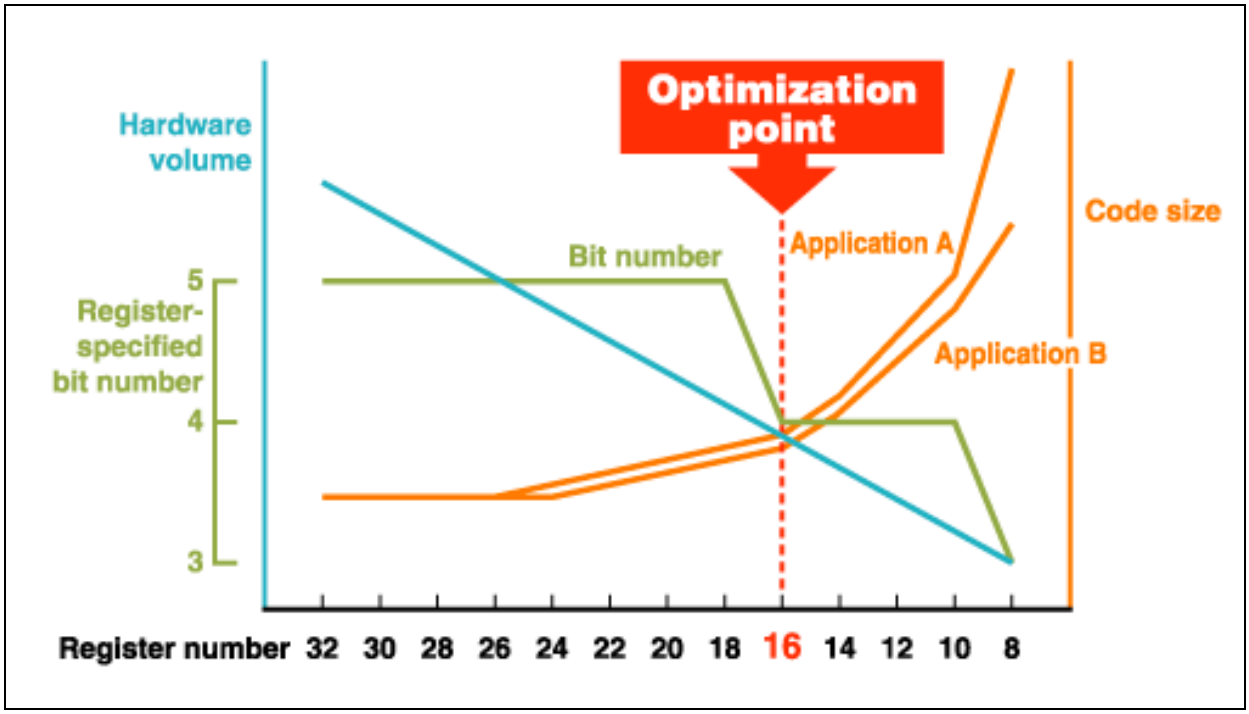


Figure 3.19 Analysis of general-purpose register configuration

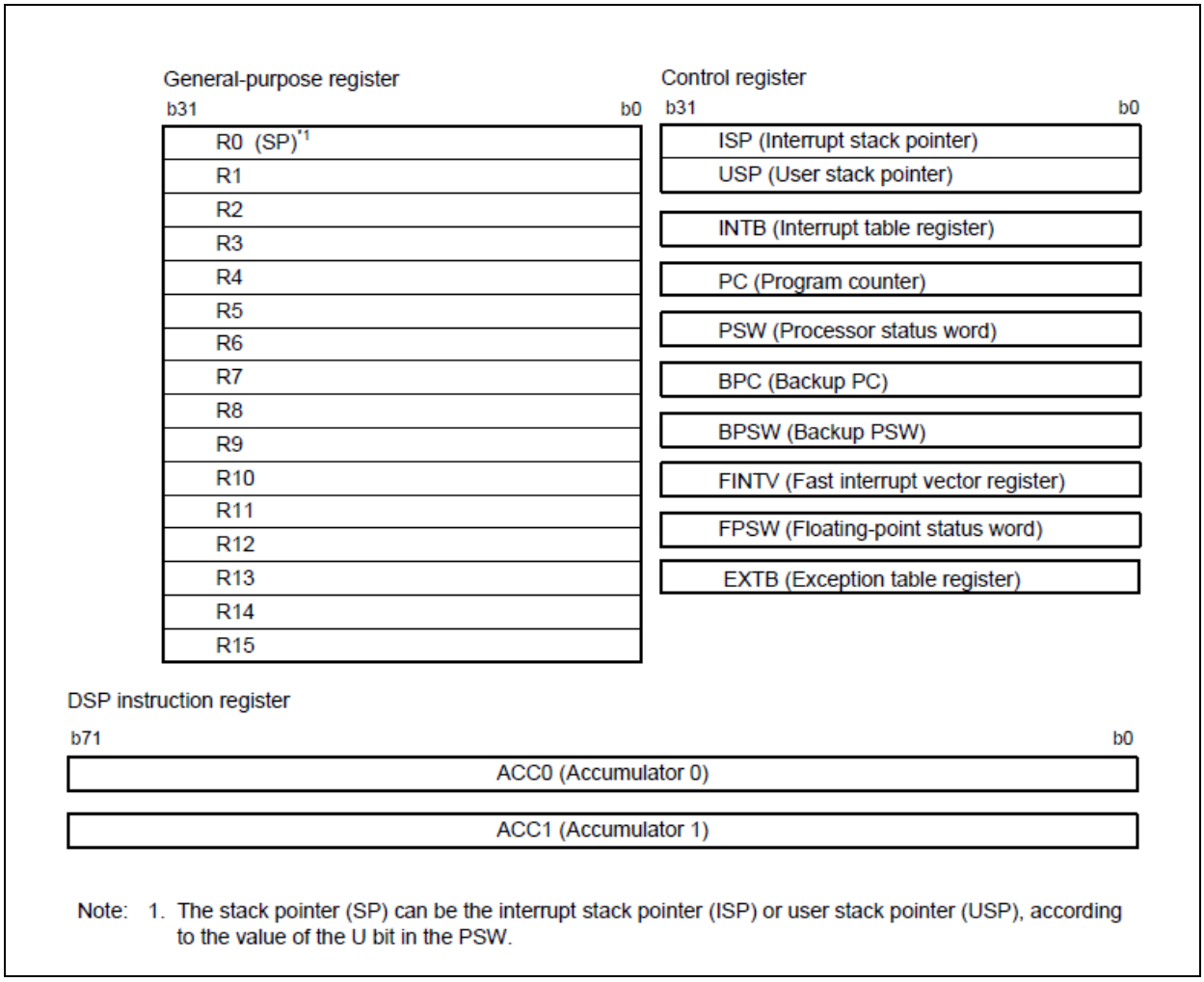


Figure 3.20 Register Set of the CPU

RX Family RXv2 Instruction Set Architecture User's Manual: Software
(copyright ©2013 Renesas Electronics Corporation)

Eight registers is too small to execute code in real applications in which save/restore operations occur quite frequently, which causes performance degradation and code size increase. The variable length instruction set allows only four bits of register-addressing fields. For register-to-register instructions, at least two register-specified fields are required. Five bits of operation code are specified for 32 registers.

In order to balance performance, hardware cost, and code density, the benchmarks led to the decision to include sixteen general-purpose registers in the RX architecture. As a result, the RX CPU has sixteen general-purpose registers, ten control registers, and two accumulators used for DSP instructions (Figure 3.20).

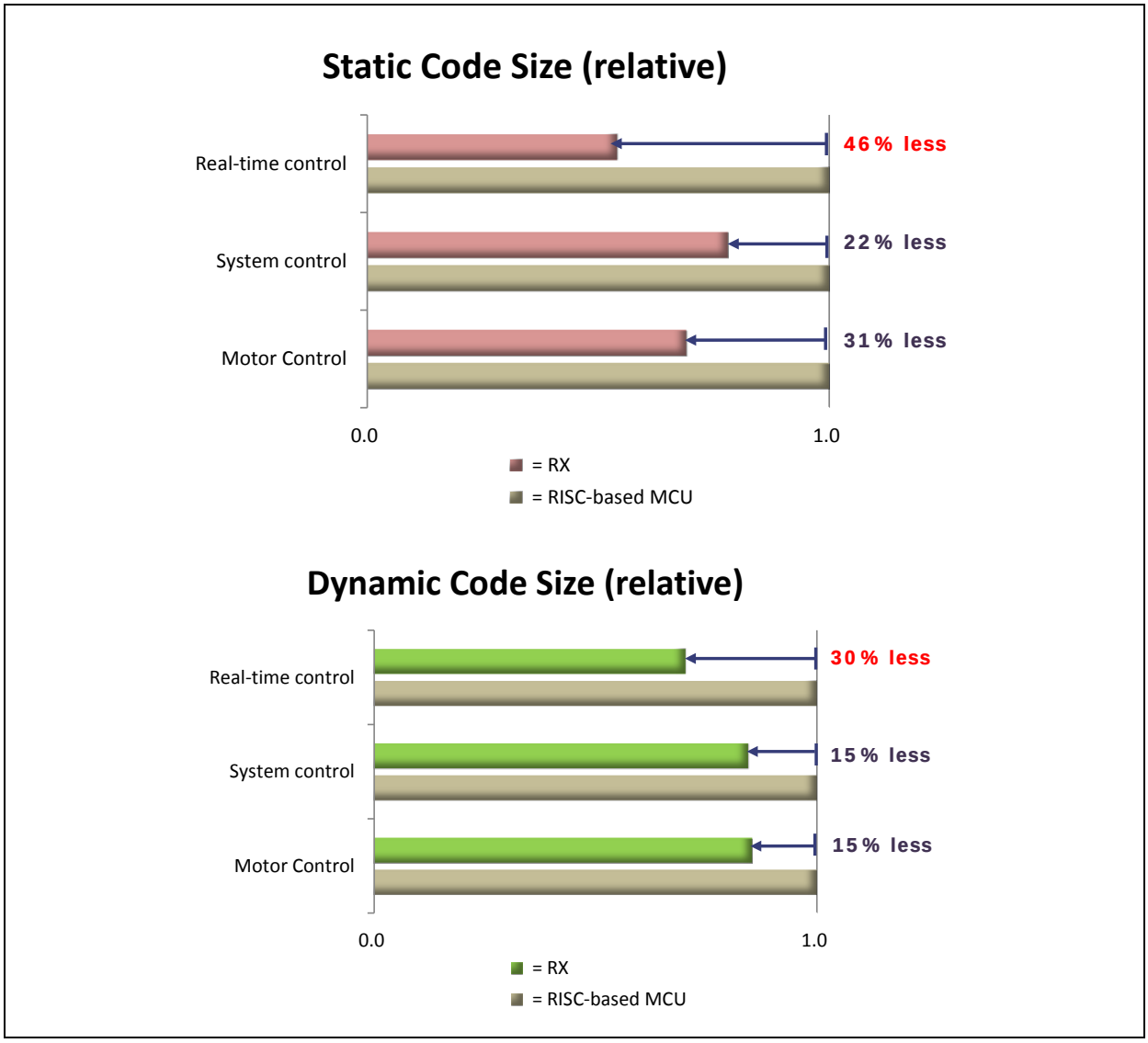


Figure 3.21 Code size analysis of the RX and a RISC-based MCU: Static Code Size (a) and Dynamic Code (b).

3.6.8 Code size evaluation

Figure 3.21 illustrates a code size analysis of the RX and a RISC-based MCU with three different types of applications, a real-time-control application, a motor-control application and system-control application. The implementation delivers up to 46% reduction in static code size, and up to 30% reduction in dynamic code size relative to RISC architectures. Small static code size makes a significant contribution in decreasing ROM size, and by extension, costs. Small dynamic code size delivers low power consumption as described in the section “Improving instruction fetch effectiveness”.

The RXv2 enhances DSP/FPU instructions to reduce the code size and to use the comparatively small number of sixteen registers more efficiently. The newly added 19 instructions are shown in red in Figure 3.16.

.

3.7 Related Works

Improving instruction fetch effectiveness is a key to reducing power consumption, and many instruction fetch ideas have been proposed for decades.

The conventional method of reducing power consumption of external memories is to integrate the cache memory on the chip to reduce the number of switched off-chip wires, which dramatically reduces system power consumption by 90%. Therefore reducing the cache miss rate has been the main topic of discussion of cache designers.

On the other hand, in microcontrollers, ROM and RAM are traditionally integrated on a chip. Elimination of external memories achieves low power consumption. However, today's devices demand much less power consumption because they depend on battery or solar power.

For embedded microprocessor-based systems, instruction fetching can contribute to a large percentage of system power (around 50%).

Several approaches have been proposed to reduce memory accesses including:

1) Program compression to reduce the number of bits fetched

2) Efficient instruction cache design to filter out accesses to main memory

Program compression

There are two major program compression techniques; one is code compression and the other is size reduction of instruction codes. Since memory accesses consume a significant amount of an embedded system's power, battery life can be extended by program compression.

Code compression architecture uses hardware to compress the most commonly executed instructions. This method reduces memory accesses per instruction [benini99]. A disadvantage of this method is the chip area and performance overhead caused by the decompression of the compressed codes.

Since the CPU is the main consumer of power, compression can result in significant power savings. If memory and/or cache are made smaller, their effective capacitance decreases, further decreasing power consumption [lekatsas00]. Because there are fewer transactions and the transactions are shorter, compression may also reduce program execution time.

Smaller code size derived from instruction set architecture provides direct benefit to save energy, which eliminate the need for compression hardware.

Efficient instruction cache design

Another approach to reducing memory accesses is efficient instruction cache design. Several tiny instruction cache architectures attract designers who need low energy or low power processors.

A filter cache is a small direct-mapped cache, which is placed between the CPU and the L1 cache. It utilizes standard tag comparison and miss logic. The filter cache is much smaller than the L1 cache; it has a faster access time and lower power consumption per access. However, it may suffer from a high miss rate and hence may decrease overall performance [kin97].

A loop cache is a small instruction buffer that is tightly integrated with the processor without tags. A loop cache controller is responsible for filling the loop cache when a simple loop, defined as any short backwards branch instruction, is detected [gross02v].

Another category of approaches is capitalizing on the common features of embedded applications by profiling, which can enable customization of an architecture to most efficiently execute a particular application [gross02c, cotterell02]. A very aggressive form of this type of architecture tuning involves creating a customized instruction set, known as an application-specific instruction set. Memory design, including a cache, is also customized for the application.

A branch target buffer is a common way to improve branch prediction performance [hennessy06]. To reduce the branch penalty for pipelines, the branch target buffer speculatively fetches instruction codes from memory before decoding the branch instructions. If the PC (program counter) of the fetched instruction matches a PC in the prediction buffer, the corresponding predicted PC is used for speculative instruction fetches. This mechanism causes extra memory accesses and increases power consumption. Therefore, branch target cache, which reuses instruction codes from memory, is necessary for low power consumption [Chihung99].

3.8 Summary

RXv2 is the new generation of RX processor architecture for microcontrollers with high-capacity flash memory. An enhanced instruction set and pipeline structure with an advanced fetch unit (AFU) provide an effective balance between power consumption performance and high processing performance. Enhanced instructions such as the DSP function and floating point operation, and a five-stage dual-issue pipeline synergistically boost the performance of digital signal applications. The RXv2 processor delivers 1.9 – 3.7x the cycle performance of the RXv1 in these applications. The decrease of the number of Flash memory accesses by AFU is a dominant determiner in reducing power consumption. The AFU of RXv2 benefits of reducing power consumption from adopting a branch target cache, which has a comparatively smaller area than that of a typical cache systems. High code density delivers low power consumption by reducing instruction memory bandwidth. The implementation of RXv2 delivers up to 46% reduction in static code size, and up to 30% reduction in dynamic code size relative to RISC architectures. RXv2 reaches 4.5 Coremark per MHz and operates up to 240MHz. The RXv2 processor delivers approximately more than 2.2 – 5.7x the power efficiency of the RXv1. Figure 3.22 shows a chip photograph of the test chip of the microcontroller with 4MB of built-in Flash memory. The chip was fabricated using a 40-nm low-power CMOS. This chip integrates one RXv2 processor that is connected to internal SRAM and Flash memories. This chip also has an internal multi-layer bus to connect the processor to peripheral IOs, and an interrupt controller unit (ICU).

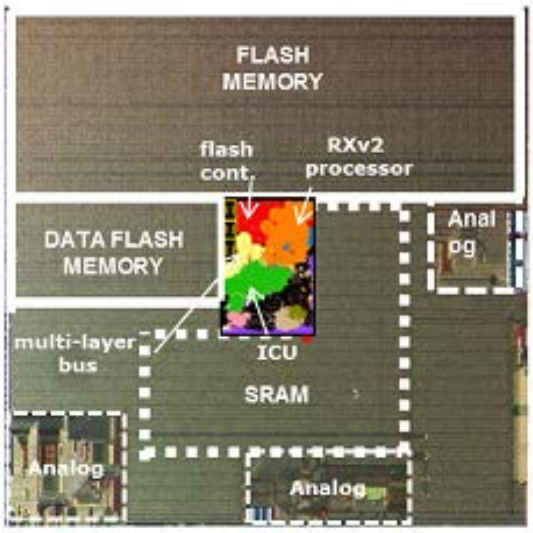


Figure 3.22 The test chip of the microcontroller with the RXv2 processor

The RXv2 microprocessor achieves the best possible computing performance in various applications such as building automation, medical devices, motor control, e-metering, and home appliances which lead to higher memory capacity, frequency and processing performance.

3.9 List of RX instruction Set

3.9.1 Arithmetic and logical instructions

Mnemonic	Function
ABS	Absolute value
ADC	Addition with carry
ADD	Addition without carry
AND	Logical AND
CMP	Comparison
DIV	Signed division
DIVU	Unsigned division
EMUL	Signed multiplication
EMULU	Unsigned multiplication
MAX	Selecting the highest value
MIN	Selecting the lowest value
MUL	Multiplication
NEG	Two's complementation
NOP	No operation
NOT	Logical complementation
OR	Logical OR
RMPA	Multiply-and-accumulate operation
ROLC	Rotation with carry to left
RORC	Rotation with carry to right
ROTL	Rotation to left
ROTR	Rotation to right
SAT	Saturation of signed 32-bit data
SATR	Saturation of signed 64-bit data for RMPA
SBB	Subtraction with borrow
SHAR	Arithmetic shift to the right
SHLL	Logical and arithmetic shift to the left
SHLR	Logical shift to the right
SUB	Subtraction without borrow
TST	Logical test
XOR	Logical exclusive or

3.9.2 Floating-point operation instructions

Mnemonic	Function
FADD	Floating-point addition
FCMP	Floating-point comparison
FDIV	Floating-point division
FMUL	Floating-point multiplication
FSUB	Floating-point subtraction
FSQRT	Floating-point square root
FTOI	Floating point to integer conversion
FTOU	Floating point to integer conversion
ITOF	Integer to floating-point conversion
ROUND	Conversion from floating-point to integer
UTOF	Integer to floating-point conversion

3.9.3 Data Transfer instructions

Mnemonic		Function
MOV		Transferring data
MOVCO		Storing with LI flag clear
MOVLI		Loading with LI flag set
MOVU		Transfer unsigned data
POP		Restoring data from stack to register
POPC		Restoring a control register
POPM		Restoring multiple registers from the stack
PUSH		Saving data on the stack
PUSHC		Saving a control register
PUSHM		Saving multiple registers
REVL		Endian conversion
REVW		Endian conversion
SC*Cnd*	SCGEU	Condition setting
	SCC	
	SCEQ	
	SCZ	
	SCGTU	
	SCPZ	
	SCGE	
	SCGT	
	SCO	
	SCLTU	
	SCNC	
	SCNE	
	SCNZ	
	SCLEU	
	SCN	
	SCLE	
	SCLT	
	SCNO	
STNZ		Transfer with condition
STZ		Transfer with condition
XCHG		Exchanging values

3.9.4 Branch Instructions

Mnemonic		Function
B*Cnd*	BGEU	Relative conditional branch
	BC	
	BEQ	
	BZ	
	BGTU	
	BPZ	
	BGE	
	BGT	
	BO	
	BLTU	
	BNC	
	BNE	
	BNZ	
	BLEU	
	BN	
	BLE	
	BLT	
	BNO	
BRA		Unconditional relative branch
BSR		Relative subroutine branch
JMP		Unconditional jump
JSR		Jump to a subroutine
RTS		Returning from a subroutine
RTSD		Releasing stack frame and returning from subroutine

3.9.5 Bit manipulation instructions

Mnemonic		Function
BCLR		Clearing a bit
BM*Cnd*	BMGEU	Conditional bit transfer
	BMC	
	BMEQ	
	BMZ	
	BMGTU	
	BMPZ	
	BMGE	
	BMGT	
	BMO	
	BMLTU	
	BMNC	
	BMNE	
	BMNZ	
	BMLEU	
	BMN	
	BMLE	
	BMLT	
	BMNO	
BNOT		Inverting a bit
BSET		Setting a bit
BTST		Testing a bit

3.9.6 String manipulation instructions

SCMPU	String comparison
SMOVB	Transferring a string backward
SMOVF	Transferring a string forward
SMOVU	Transferring a string
SSTR	Storing a string
SUNTIL	Searching for a string
SWHILE	Searching for a string

3.9.7 System control instructions

Mnemonic	Function
BRK	Unconditional trap
CLRPSW	Clear a flag or bit in the PSW
INT	Software interrupt
MVFC	Transfer from a control register
MVTC	Transfer to a control register
MVTIPL (privileged instruction)	Interrupt priority level setting
RTE (privileged instruction)	Return from the exception
RTFI (privileged instruction)	Return from the fast interrupt
SETPSW	Setting a flag or bit in the PSW
WAIT (privileged instruction)	Waiting

3.9.8 DSP function instructions

Mnemonic	Function
EMACA	Extend multiply-accumulate to the accumulator
EMSBA	Extended multiply-subtract to the accumulator
EMULA	Extended multiply to the accumulator
MACHI	Multiply-Accumulate the high-order word
MACLH	Multiply-Accumulate the lower-order word and higher-order word
MACLO	Multiply-Accumulate the low-order word
MSBHI	Multiply-Subtract the higher-order word
MSBLH	Multiply-Subtract the lower-order word and higher-order word
MSBLO	Multiply-Subtract the lower-order word
MULHI	Multiply the high-order word
MULLH	Multiply lower-order word and higher-order word
MULLO	Multiply the low-order word
MVFACGU	Move the guard longword from the accumulator
MVFACHI	Move the high-order longword from accumulator
MVFACLO	Move the lower-order longword from the accumulator
MVFACMI	Move the middle-order longword from accumulator
MVTACGU	Move the guard longword to the accumulator
MVTACHI	Move the high-order longword to accumulator
MVTACLO	Move the low-order longword to accumulator
RACL	Round the accumulator longword
RACW	Round the accumulator word
RDACL	Round the accumulator longword
RDACW	Round the accumulator word

Chapter 4

PEACH: A Multicore Communication SoC with PCI Express I/F

4.1 Introduction

The eight-core communication SoC, code-named “PEACH”, with four 4x PCI Express rev.2.0 ports, realizes a high performance, power-aware, highly dependable network. The network uses PCI Express not only for connecting peripheral devices but also as a communication link between computing nodes. This approach opens up new possibilities for a wide range of communications. Recent trends in using computing clusters point to a growing demand for high-compute-density environments in various application fields such as server appliances including distributed Web servers. Distributed Web servers need many server nodes and low-latency and high-bandwidth network for operating a massive amount of Web services, including distribution of high-definition movies. In these computing clusters, power consumption and system cost have increased. Therefore, it’s vital to downsize computing cluster without losing high dependability, including fault tolerance.

To realize high-performance, power-aware, and highly dependable network, we have proposed a small computing cluster for embedded systems, called PEARL (PCI Express Adaptive and Reliable Link) [hanawa10].

Commodity network devices such as Gigabit Ethernet (GbE) and InfiniBand aren’t sufficient for small computing clusters. InfiniBand is a switched fabric communication link used in high-performance

computing and enterprise data centers. It achieves high reliability but power consumption is relatively high [infiniband]. The external switching devices are needed to connect between nodes, and they restrict flexibility of network topology and scalability. GbE is a cost and power rival of InfiniBand. However, GbE does not match InfiniBand's transmission performance.

To achieve both high performance and low power consumption, PEARL uses PCI Express [pcie06], a high-speed serial I/O interface standard in PCs, not only for connecting peripheral devices but also as a communication link between computing nodes. To implement PEARL, we've developed a communication device called PEACH (PCI Express Adaptive Communication Hub), which acts as a switching device. PCI Express transfers packets point-to-point bi-directionally with high bandwidth. However, it connects only between a Root Complex (RC) and Endpoints (EPs).

Therefore, we can't connect PCI Express interfaces on PCs to one another, because every node CPU in the computing node is an RC. To solve this problem, PEARL equips each node CPU with a network interface card with PEACH. A PCI Express cable connects the node CPUs to one another. To pair a RC with EPs at each end of the PCI Express cable, PEACH can switch the RC port and EP ports to connect two computing nodes peer-to-peer. Thus, PEACH can address two computing nodes as peers, breaking the traditional PCI Express limit of only linking to a single master.

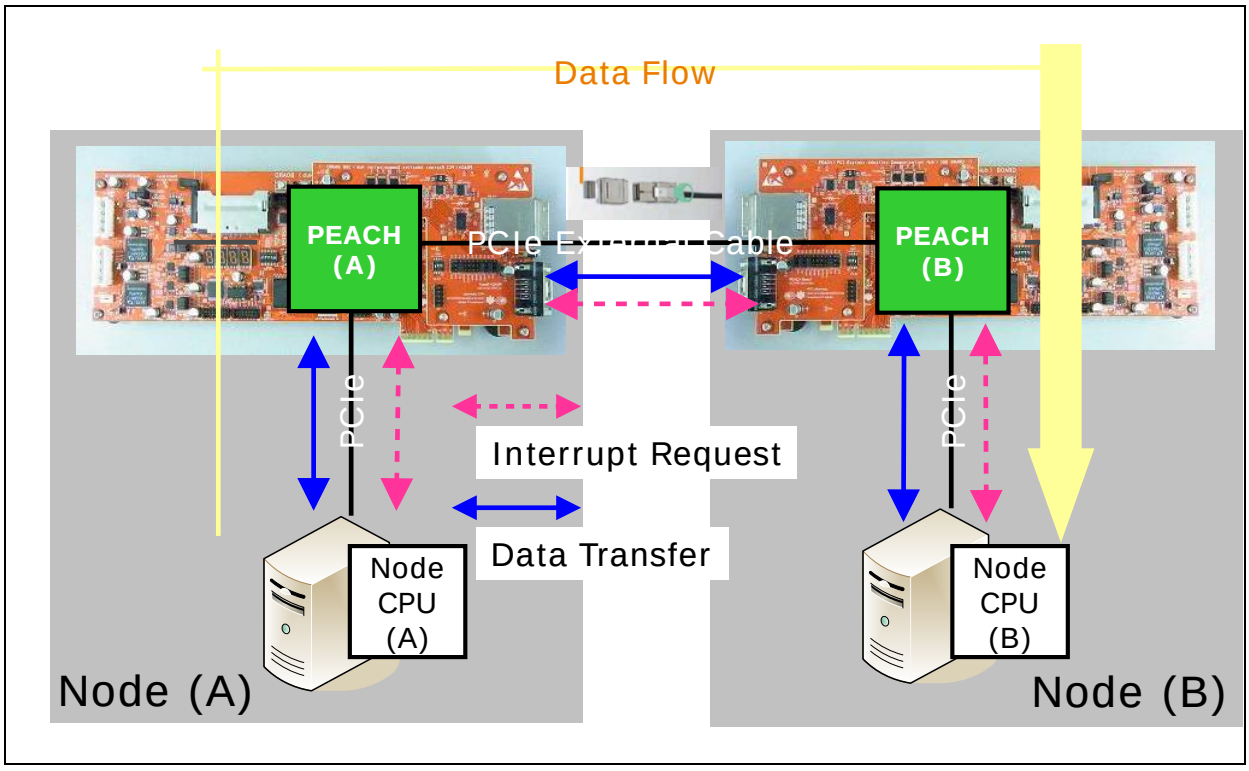


Figure 4.1 The communication link, PEARL.

A network interface card with the network device, PEACH, can be inserted into a PCI Express slot on a computing node's motherboard (Figure 4.1).

4.2 PEACH Architecture

4.2.1 PEACH overview

PEACH, has four PCI Express Revision 2.0 ports with four lanes each and employs an eight-core control processor [otani11f]. Using PEACH in the proposed network offers several advantages. Four PCI Express ports can broaden the scope of network topology selection. The high bandwidth of 20 Gbps/port equals that of InfiniBand DDR 4X. The multicore control processor performs fault handling, system monitoring, and logging for dependability. The multicore processor also controls the network system for power awareness.

Figure 4.2 shows a PEARL network system prototype. PEACH behaves as a communication interface to other computing nodes as well as a communication switch. Figure 4.2 illustrates adaptive routing under a normal network condition (a), detour routing in a fault network condition (b), and an example of an eight-node network (c). PEACH acts as a communication link and connects nodes of the network via its four PCI Express ports.

In Figure 4.2 (a), PEACH connects four network nodes via its four PCI Express ports. One adjacent nodes is a node CPU, and the others are PEACH chips. When PEACH 0 receives a request from a node CPU via the PCI Express port, PEACH 0 generates a packet header, which is then sent to the appropriate destination port. When PEACH 1 receives a packet from a node, PEACH 1 analyzes the packet header, and PEACH 1 forwards the packet to another node or passes it to the node CPU.

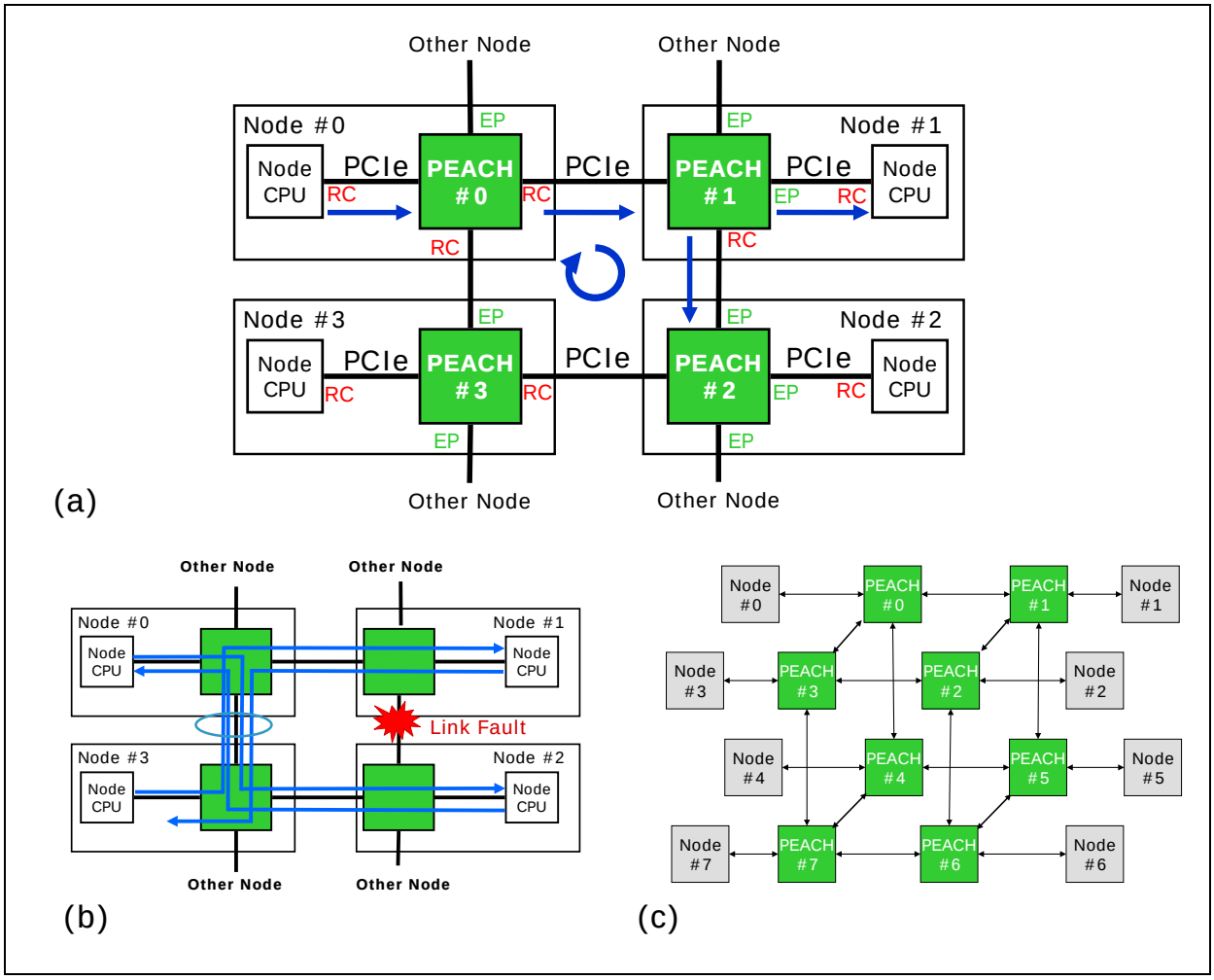


Figure 4.2 Neighbor communication on PEARL.

The hardware automatically processes the error-detection, flow-control, and retransmission-control functions in the PCI Express specifications. InfiniBand supports only link-by-link cut-off, so only one faulty lane must causes a link to go down. In contrast, when a link error that can't be automatically corrected occurs, PEACH reduces the number of the PCI Express port's lanes to remove the defective lane by reinitializing the link. This InfiniBand enhancement can provide higher network reliability.

This chip integrates an eight-core control processor, four PCI Express ports, and an intelligent interrupt control unit (ICU).

Detour routing is applied in a fault condition to bypass faulty links and nodes, and it enables network function recovery (Figure 4.2(b)). PEACH continuously monitors the system and dynamically performs both adaptive routing, to meet power and performance demands, and detour routing to achieve a highly dependable network.

Although the number of nodes in PEARL is theoretically limitless, our design target is a network with 16 nodes.

4.2.2 Chip Architecture

Figure 4.3 shows PEACH's primary functional unit. This chip includes two blocks, the control processing block and the transfer processing block, which are connected with a bus bridge. Figure 4.4 shows the chip micrograph. The test chip was fabricated in a 45nm low-power CMOS (8 layers, triple-Vth).

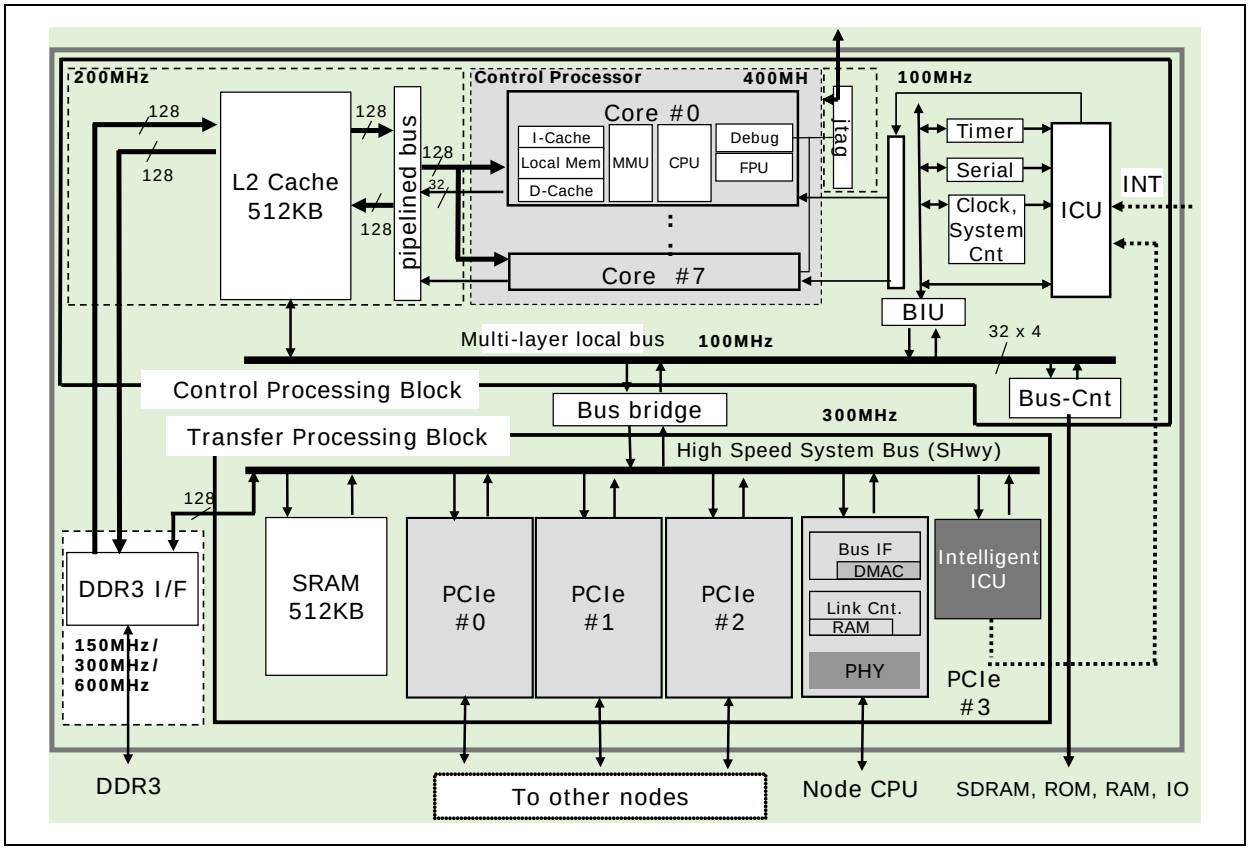


Figure 4.3 PEACH block diagram.

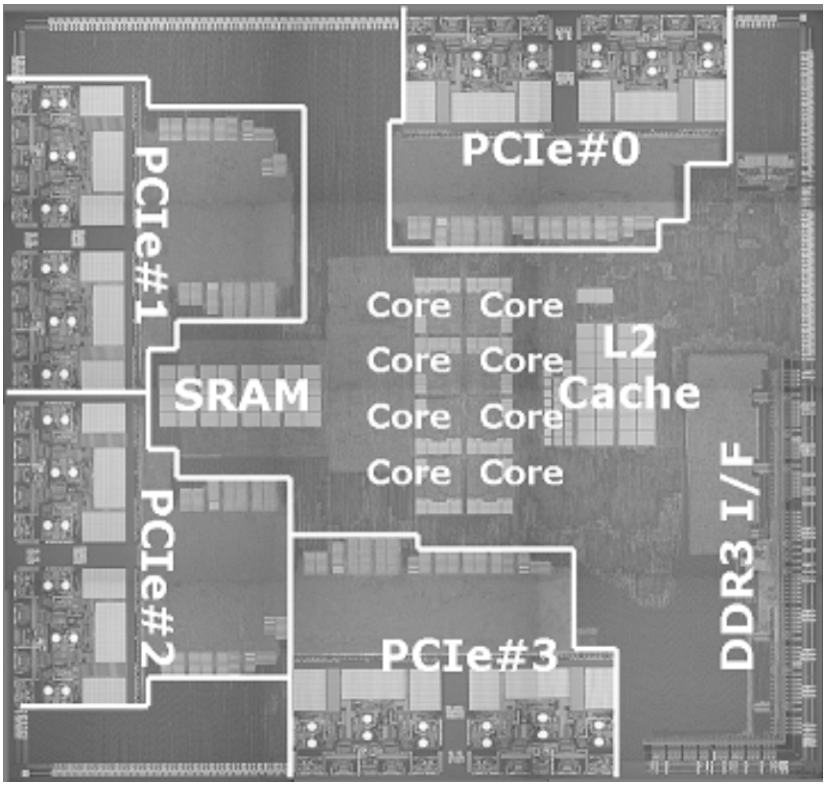


Figure 4.4 PEACH micrograph.

The transfer processing block has four PCI Express ports, each of which can transfer packets using up to four lanes; 512Kbytes of static RAM (SRAM) allocated for temporary packet storage; and an intelligent interrupt control unit (ICU) in close liaison with the multicore processor and the PCI Express interface [otani11n]. A high-speed internal system bus connects all the main modules in the transfer processing block. The intelligent ICU also supports the fast automatic data-transfer function that offloads interrupt services from the multicore processors.

The control processing block performs data-processing and data-flow control, which consists of adaptive network routing and packet header analysis. In the control processing block, a cache coherence mechanism connects the eight cores to a common pipelined bus [kaneko04]. Each core is synthesizable and includes a floating-point unit (FPU), a memory management unit (MMU), three 8Kbyte memories for Level 1 (L1) instruction and data caches, and a local memory. The control processor is a symmetric multiprocessor (SMP) that supports a core grouping mode that divides cores into several groups [kondo08].

The pipelined bus is connected to a 512Kbyte L2 cache. This pipelined bus with a large bus width (a 128-bit read bus and a separate 32-bit write bus) reduces its bus traffic and connects directly to an internal multilayer bus. The 256Mbyte DDR3-600 interface is accessed via both the control processing block and the transfer processing block in parallel. This high-speed, large memory contributes to improving the chip performance. The DDR memory is also used as a large packet buffer if the packet size is larger than 512Kbyte SRAM.

PEACH's multicore processor offers an effective paradigm for fast packet processing. In a multicore system, we must carefully consider the hardware and software architecture. We can assign network packet processing from a specific PCI Express port to dedicated cores, to bind specific tasks and specific cores. By effectively distributing processing on a multicore processor, we can realize high traffic rates from multiple 20-Gbps throughputs on multiple PCI Express ports.

4.2.3 PCI Express interface with up-configuration function

Table 4.1 describes the PCI Express interface's features. Each PCI Express port has a link controller, PHY, a local DMA controller (DMAC), and local packet buffer RAM. The latest Revision 2.0 standard has transfer rate of up to 5.0Gbps, double that of the Revision 1.1 (2.5Gbps). Revision 2.0 supports both 2.5Gbps and 5.0Gbps transfer rates because of compatibility with Revision 1.x. Furthermore, PCI Express specifications govern a procedural step in going from 2.5Gbps up to 5.0Gbps. The total transfer rate to each destination is 20Gbps, and the theoretical peak bandwidth is actually 2Gbps due to 8-bit and 10-bit encoding for the embedded clock and error detection. PEACH with four PCI Express ports realizes a high-performance communication of 4 x 20Gbps and a power efficiency of 0.04W/Gbps.

Chip Characteristic	Description
Clock frequency	Internal: 400 MHz max. External bus: 100 MHz
Processor	8core, SMP L1-cache:8kB(I)+8kB(D), LM:8kB, MMU, FPU
Core	32-bit Processor (400 MHz max.)
Memory	L2 cache: 512 kB Internal SRAM: 32 kB, 512 kB
DRAM I/F	DDR3-600 I/F x 1, SDRAM I/F x 1
PCIe I/F	PCI Express standard Rev.2.0 Transfer speed: 5.0 GT/s, 2.5 GT/s per lane 4 lanes (20 Gbps) x 4 ports Upconfiguration function Automatic retransmission function Selectable Root Complex / Endpoint
Intelligent Interrupt Control Unit	Transfer address, size information register x 3 Initiate data transfer function
Bus	Packet router Multi-layer bus (4-layer) Pipelined bus

Table 4.1 PEACH Chip Features

InfiniBand DDR 4X has a high bandwidth of 20Gbps and a low latency of 2μs. The subnet manager provides automatic fault recovery [infiniband]. However, overall system power consumption increases because a controller chip and a switch each consume 3 to 5 watts per port. Multiple switches, which are necessary for fault tolerance, would run counter to cost reduction and low power consumption. The power efficiency of InfiniBand 4X is 0.083W/Gbps [qlogic]. Thus, PEACH provides 51.5% better power efficiency than InfiniBand 4X (Table 4.2).

	4x InfiniBand	PEARL
Network Device	Dedicated Circuit InfiniBand	PEACH PCI Express
Power Efficiency [W/Gbps]	0.083	0.040 51.5%

Table 4.2 Comparison of Power Efficiency

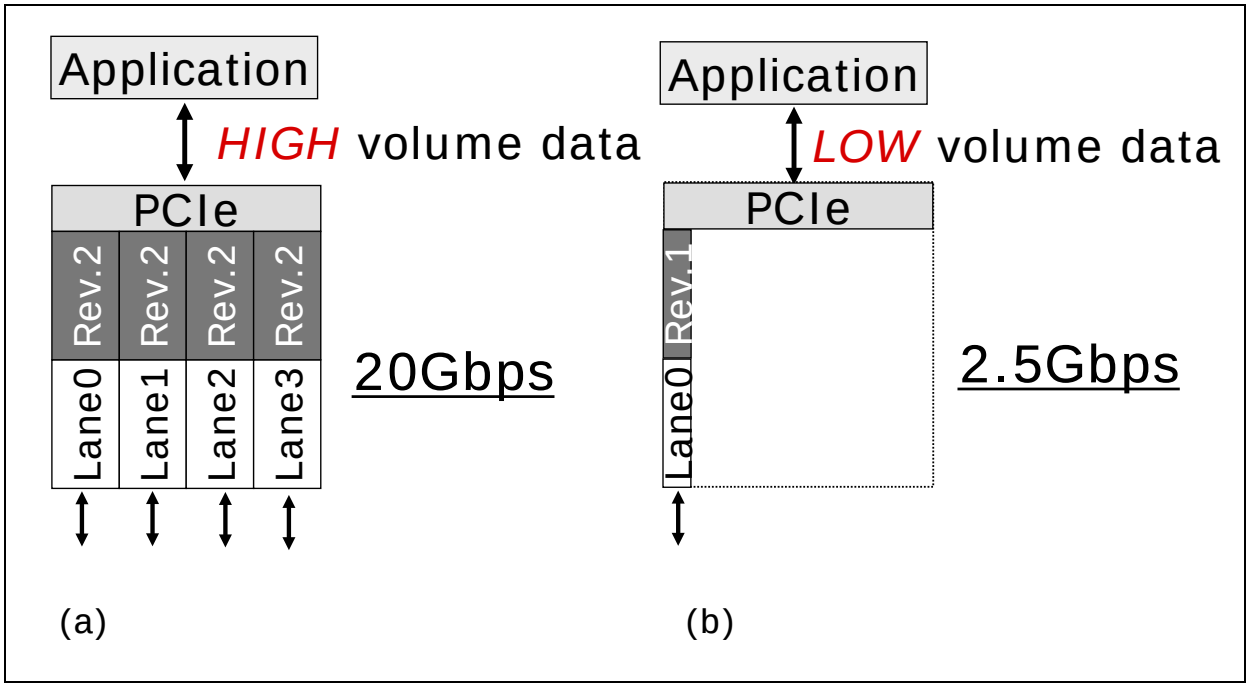


Figure 4.5 PCI Express up-configuration function by software control. (a) Maximum data transfer rate (b) Low power consumption.

4.2.4 PCI Express up-configuration function

InfiniBand offers restrictive power-aware control of link-by-link power cut-off. In contrast, PCI Express has an effective power-aware control that can change the number of lanes and the lane speed on the fly, across the link and nodes. We use a PCI Express up-configuration function that lets us switch the transfer rate and the number of lanes in response to a software bandwidth change (Figure 4.5). When the required transfer volume is higher, the PCI Express port operates at the full of 20Gbps. When the transfer volume is lower, only one lane operates at 2.5Gbps for low power consumption.

Table 4.3 compares the comparison of the PCI Express PHY power consumption. In low power consumption mode, using the PCI Express port of 2.5Gbps provides 76% less power consumption than that of 20Gbps. The maximum transfer rate using the PCI Express port of 20Gbps provides 52% better power efficiency compared to that of the low power consumption of 2.5Gbps.Figure 4.6 shows the power consumption of PCI Express PHY at each requested transfer volume. When the required transfer volume is lower than 2.5Gbps, using one PCI Express port of 2.5Gbps provides the lowest power

consumption. When the required transfer volume is larger than 2.5Gbps, using PCI Express ports of 5.0 Gbps is worthwhile.

	No. of lanes		
Lane Speed	**4 lanes**	**2 lanes**	**1 lane**
5Gbps	**20Gbps** 1.00	0.50	0.28
2.5Gbps	0.84	0.42	**2.5Gbps** 0.24

Power Efficiency of PCIe PHY (W/Gbps)

(1.00/20) / (0.24/2.5) = ***0.52***

5Gbps@4 lanes 2.5Gbps@1 lane

Table 4.3 Power Consumption of PCI Express PHY (Normalized)

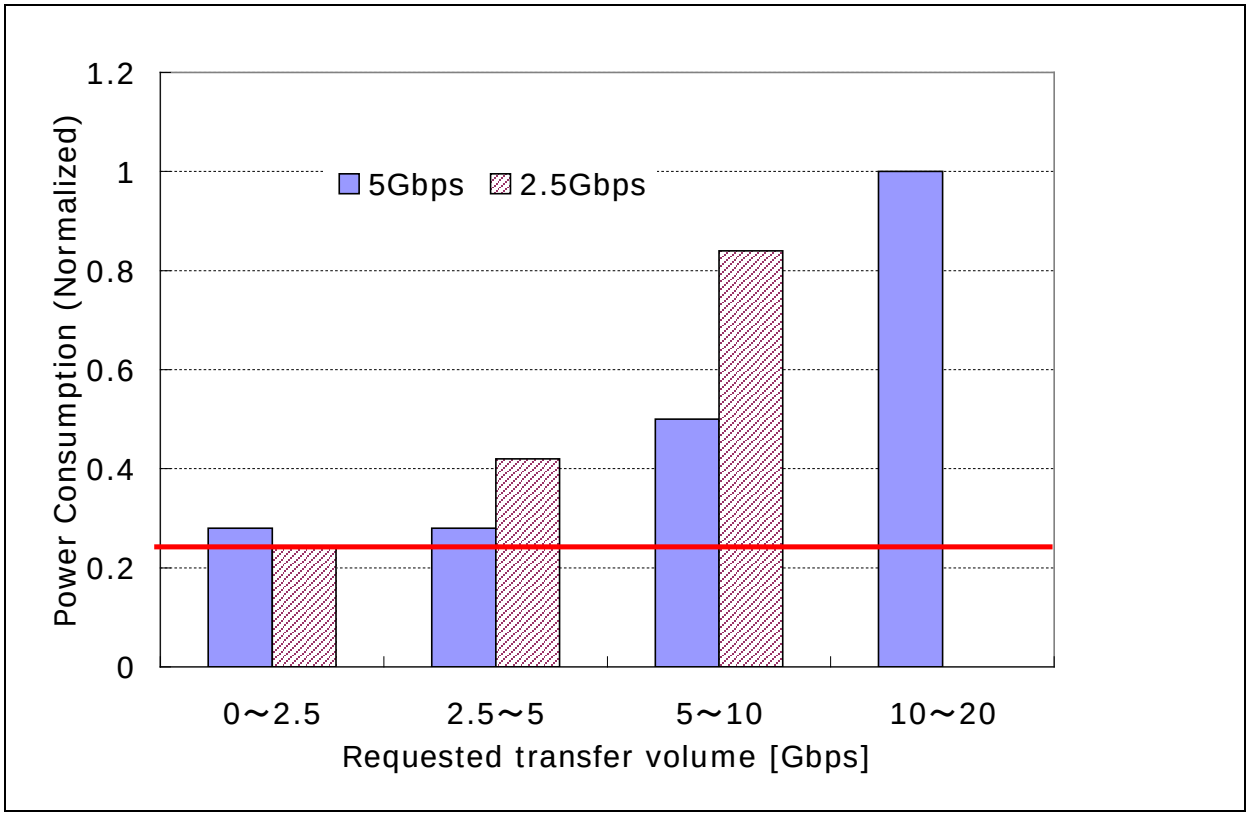


Figure 4.6 Power Consumption of PCI Express PHY (W) at each requested transfer volume.

4.2.5 Intelligent Interrupt Controller

Figure 4.7 shows the intelligent ICU block diagram. The intelligent ICU communicates with PCI Express ports within the PEACH. It also communicates with adjacent PEACH chips and nodes. Both communications use message passing via the high-speed system bus and via PCI Express. The intelligent ICU can also send a message signaled interrupt (MSI) packet to the adjacent node via PCI Express. To notify the intelligent ICU that PCI Express DMA transfer is complete or that there are PCI Express errors, PCI Express Link sends interrupt requests directly to the intelligent ICU.

The intelligent ICU's key features are an

- Interrupt relay function
- Inter-chip interrupt function
- Fast automatic data transfer function

All functions are used in inter-node communication.

The interrupt relay function relays interrupt requests from the PCI Express interface in the transfer-processing block to the cores via the ICU in the control-processing block in PEACH. It sends these interrupt requests as notifications that the PCI Express linkup or PCI Express DMA transfer processing is completed.

An inter chip interrupt function sends information such as a notification of the chip-to-chip data transfer is completed. An adjacent chip connected to PEACH via PCI Express can write a control register in the intelligent ICU to assert an interrupt request to a core.

The fast automatic data-transfer function automatically handles transfer processing without using cores in PEACH.

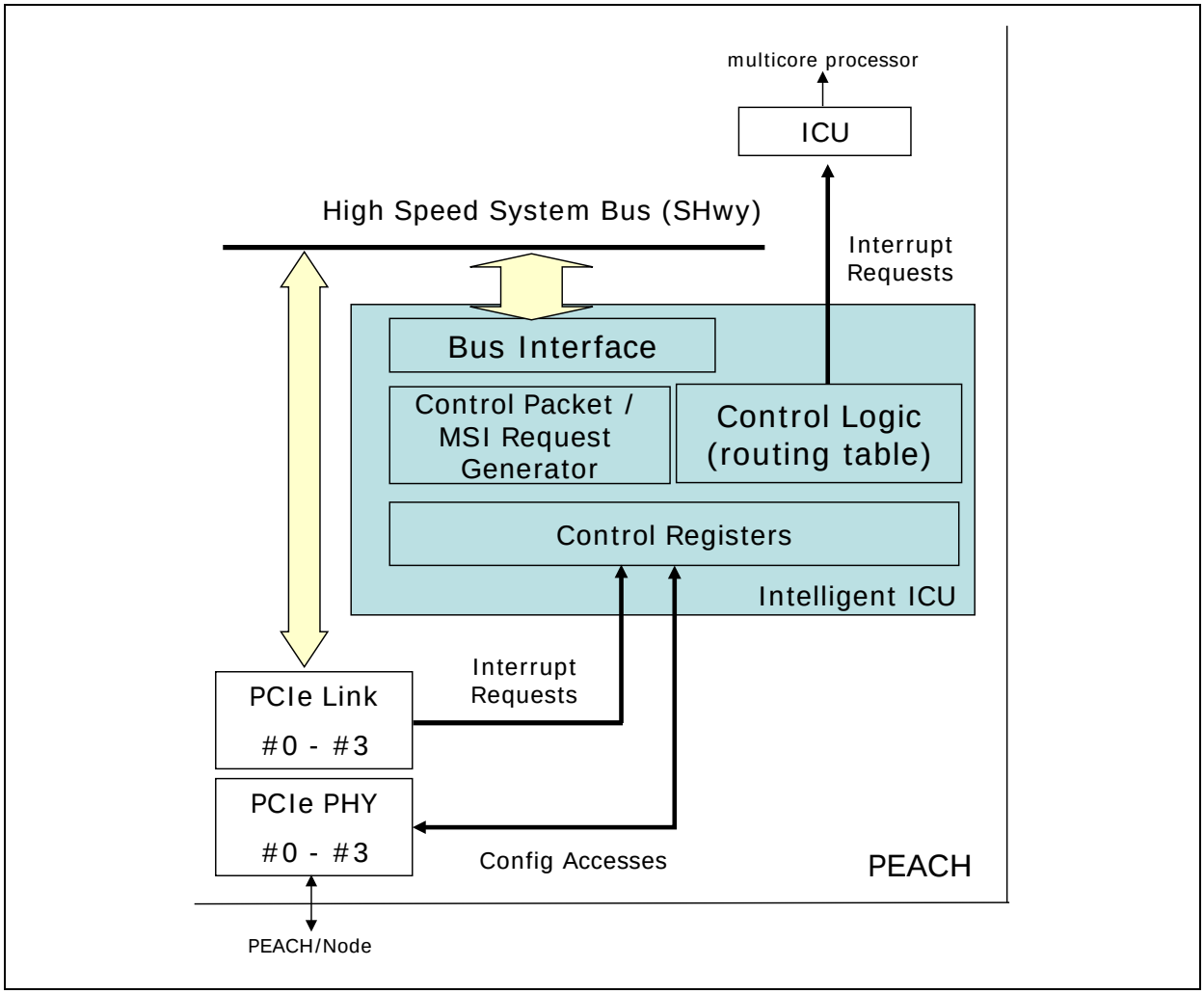
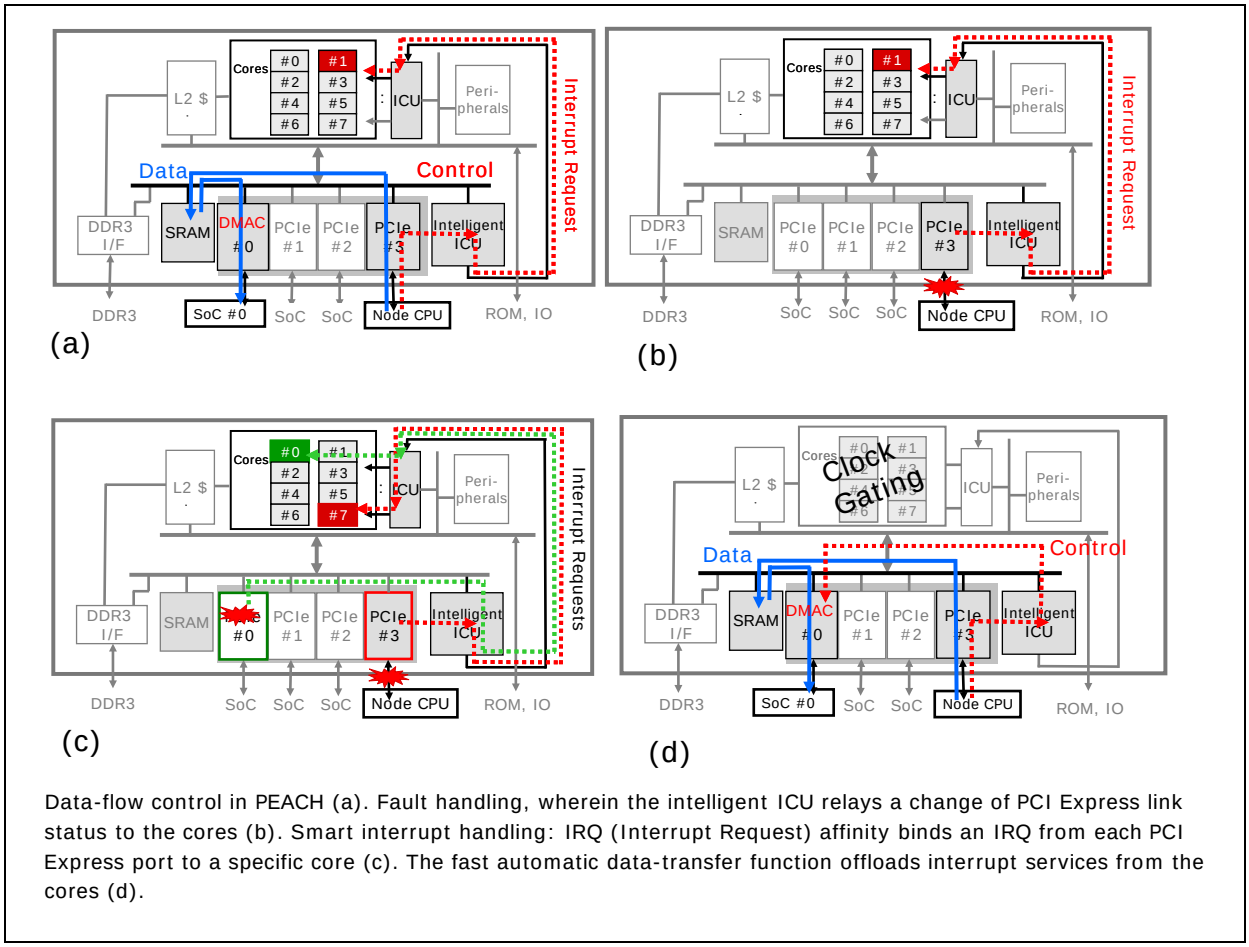


Figure 4.7 Block Diagram of Intelligent ICU

4.3 Network Managing

4.3.1 Data Flow Control

IRQ (Interrupt Request) affinity on Linux lets programs specify which core services a given interrupt. In PEACH, IRQ affinity binds an interrupt from each PCI Express port to a specific core in a one-to-one relationship. The network packet is directed to the desired core. By using this distributed processing, PEACH can process a packet efficiently. Furthermore, a snooping group of cores alleviates snooping overhead, because cores can be snooped only from other cores in the same group. Eliminating unnecessary internal snoop transaction improves the communication services' stability.



Data-flow control in PEACH (a). Fault handling, wherein the intelligent ICU relays a change of PCI Express link status to the cores (b). Smart interrupt handling: IRQ (Interrupt Request) affinity binds an IRQ from each PCI Express port to a specific core (c). The fast automatic data-transfer function offloads interrupt services from the cores (d).

Figure 4.8 Efficient packet processing and fault handling in PEACH.

Figure 4.8 (a) illustrates the data-flow control in PEACH. The solid dotted arrows indicate data flow. When PCI Express 3 receives data from a node CPU, the SRAM temporarily stores the data. After that, the data is sent to another PEACH via the appropriate destination port (PCI Express 0).

The dotted arrows indicate control flow. Devices connected to PEACH via PCI Express can send control packets to the intelligent ICU. The node CPU sends control packets and an interrupt request packet to establish communication. The intelligent ICU relays this interrupt request to a core. In the interrupt handler, the core analyzes the packets, performs an address transformation, and launches the DMAC in PCI Express.

Smart interrupt handling, such as quick error response and fault-handling speedup is essential for dependability. Therefore, good load balancing and performance tuning requires control wherever interrupt services are performed. IRQ affinity assigns a specific core to a PCI Express port to process interrupt service tasks requested only by that PCI Express port. The system software makes the core

idle steadily except during an interrupt services. There's no overhead of a context switch from a previous process, and the core can smoothly move to the interrupt processing, which speeds up the interrupt response time (Figure 4.8(c)). Smart interrupt handling also supports the fast automatic data-transfer function (Figure 4.8 (d)). The intelligent ICU can transfer data without using the cores' interrupt services by automatically performing address transformation and handling the DMAC in PCI Express. Figure 4.8 (b) shows fault handling for dependability. When communication is broken up because of a fault on a link or an adjacent node, PCI Express sends an interrupt request to a core via the intelligent ICU. The core starts error recovery by removing the defective lane or applying detour routing.

Figure 4.9 shows two data transmission flows - processor mode using interrupt services (a) and intelligent ICU mode using fast automatic data transfer (b). This chart indicates data transmission flows from Node CPU0 to Node CPU1 via PEACH A and PEACH B. All communication packets between nodes are sent via PCI Express.

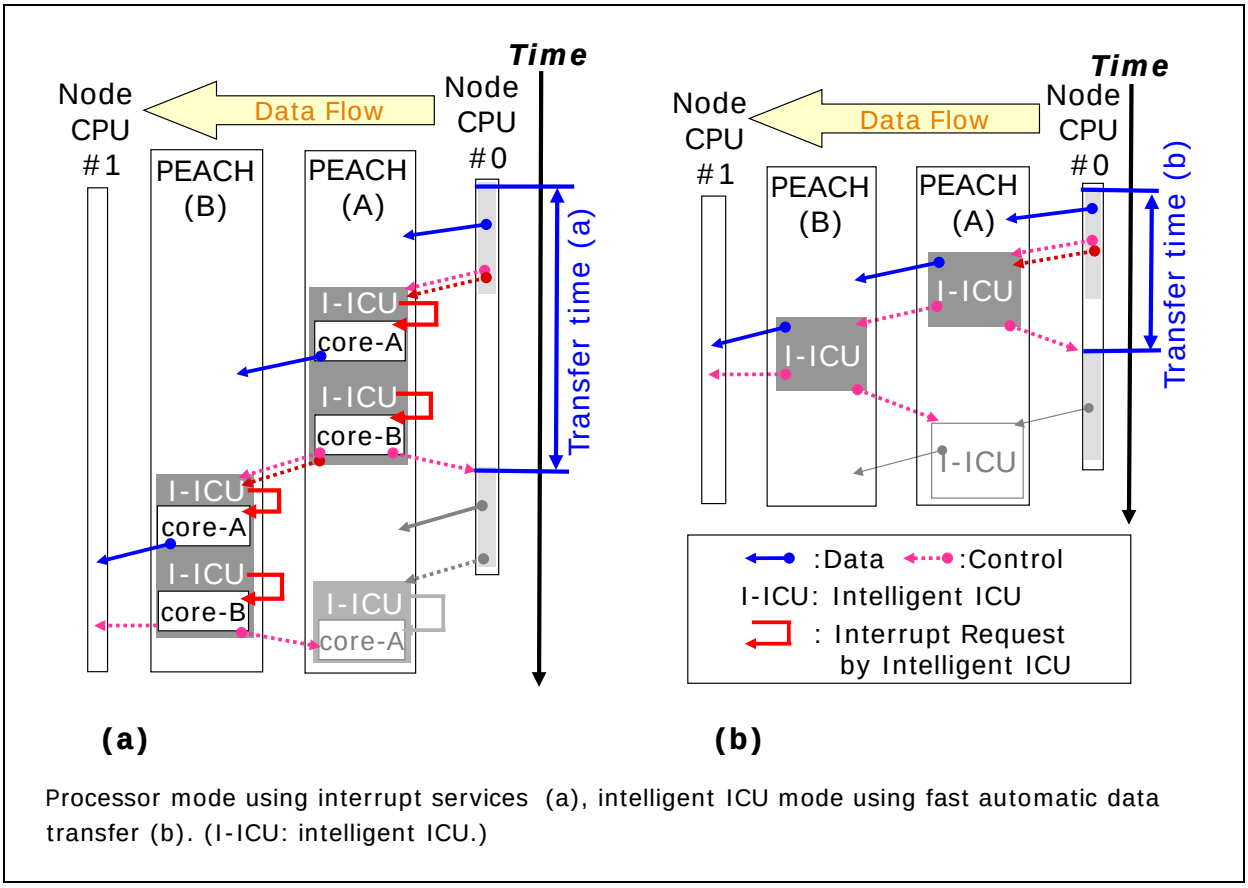


Figure 4.9 Two data transmission flows: Processor mode using interrupt services

Figure 4.9 (a) shows the data transmission flow using core interrupt services. After Node CPU0 sends a data packet and a control packet, it sends an interrupt request packet to the intelligent ICU in PEACH A to establish a communication channel. The intelligent ICU relays the interrupt request and control packet to a core in PEACH A. In the interrupt handler (core-A), the core analyzes the packets including the data's source, destination addresses, and size, and then update the packet headers and sends the packets to the destination.

The core launches the DMAC in PCI Express and transfers data to PEACH B. PCI Express notifies the core that data transfer is completed via the intelligent ICU. In the interrupt handler (core-B), the core sends a control packet and an interrupt request packet to PEACH B, and an end packet to Node CPU0. PEACH B acts in a similar manner to PEACH A and Node CPU1 finally receives data.

Although packet processing executed in the multicore processor is flexible, we can't avoid interrupt processing overhead. The intelligent ICU has a routing table and handles route computation automatically, which can reduce transfer latency. Figure 4.9 (b) shows how the fast automatic data-transfer function handles transfer processing without using the multicore processor. Node CPU0 sends an initiate data-transfer request packet to PEACH A. The intelligent ICU in PEACH A launches the DMAC in PCI Express and automatically transfers data to PEACH B. The intelligent ICU in PEACH B acts similarly, and Node CPU1 finally receives the data.

The intelligent ICU's fast automatic data-transfer function can dramatically reduce transfer processing time by 20% under normal network operation (Figure 4.10). Because it adopts a multicore processor and the intelligent ICU, PEACH acts as an intelligent network device.

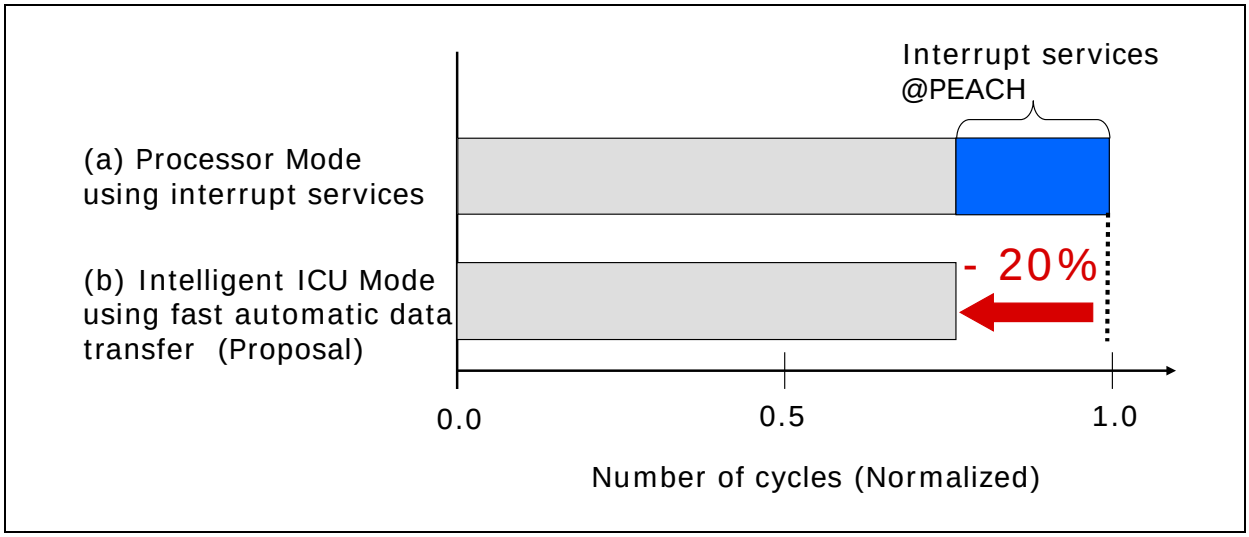


Figure 4.10 The intelligent ICU's fast automatic data-transfer function improves transfer latency.

4.3.2 PEARL network route construction

A network manager runs as a user-land program on each Linux on PEACH chip. Each PEACH chip has a routing table in a driver. PCI Express notifies the daemon of a link-status change using Linux's sysfs interface.

There are two data transmission flows. Intelligent ICU mode can transfer data quickly, but it has only a fixed routing table. Processor mode provides a flexible routing for error handling or a start-up sequence. On start-up, a master node does a route search and makes the routing table under the assumption of intelligent ICU mode. When a fault occurs, the multicore processor acts as a backup, or overwrites routing tables to modify the route.

4.3.3 Network system power management

Each daemon program on PEACH monitors network status and sends information to elected master node. The master node makes a power-aware order to network, and each network manager on PEACH changes PCI Express link configuration. On the basis of the application's demand, the network manager can change the network link performance. PEACH's multicore processor monitors the network status using a demon program, managing PCI Express physical layer (PHY) performance using the up-configuration function and processor power-management such as clock-gating.

An important point here is that adopting a multicore processor can provide fine-grained control of PCI Express configuration and reduce the overall system's power consumption.

4.4 Evaluation System

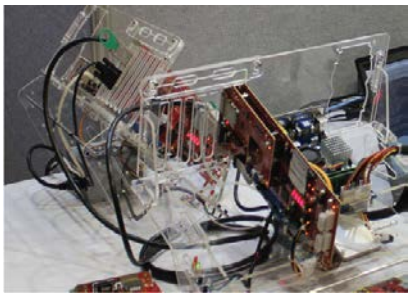
4.4.1 PEARL system board

On the basis of the component descriptions we've discussed, we've developed a six-node prototype of our PEARL network system. Figure 4.11 (c) shows a photograph of a PCI Express x4 host adapter board that has a PEACH chip and PCI Express external cable connectors [pcie07j]. The board also has a Compact-Flash card slot, 4Mbyte flash memory and two 128Mbyte DDR3 memories. The Compact-

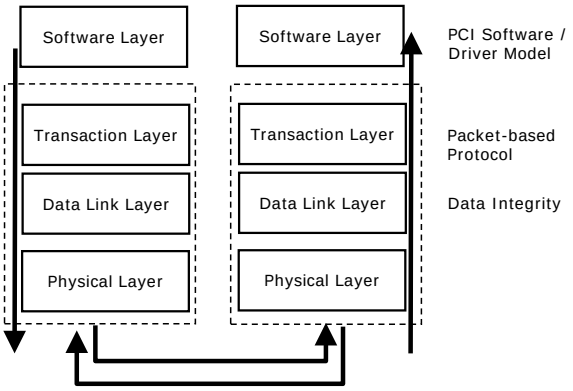
Flash card contains a Linux ext3 file system, including Linux kernel 2.6.35, and is also used for storing log information. Linux runs on a multicore processor in PEACH on a stand-alone basis, booting by loading the Linux kernel image on the Compact-Flash card. This host adapter board can be inserted into a PCI Express slot on a computing node's motherboard (Figure 4.11 (a)). The CPU hotplug on Linux can dynamically suspend and resume a core responding to system load, which are useful for power awareness.

The PCI Express architecture consists of four discrete logical layers (Figure 4.11(b)). From the bottom up, they are the physical layer, the data-link layer, the transaction layer, and the software layer.

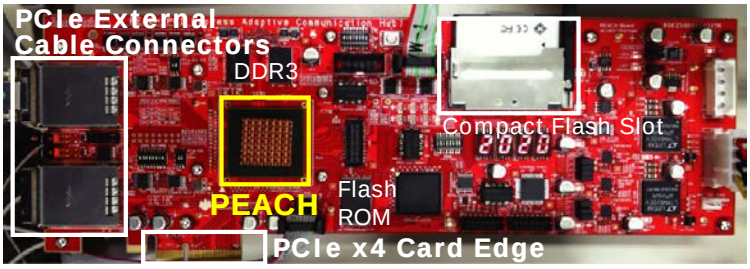
The software layer generates read and write requests that are transported by the transaction layer. The transaction layer manages the transactions for communication, such as read or write to/from memory, message passing, or configuration. The data-link layer handles link management, including packet sequencing and data integrity, which includes error detection and error correction. The physical layer comprises all circuitry, including a driver with impedance matching and input buffers, parallel-to-serial and serial-to-parallel conversion, and PLLs. Each PCI Express port has a PHY, a data-link layer controller (MAC), and a transaction layer controller as hardware modules. Figure 4.11(d) shows six-node prototype of a PEARL network system.



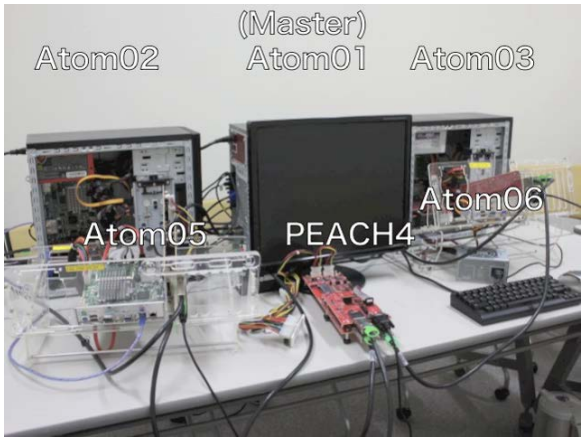
(a)

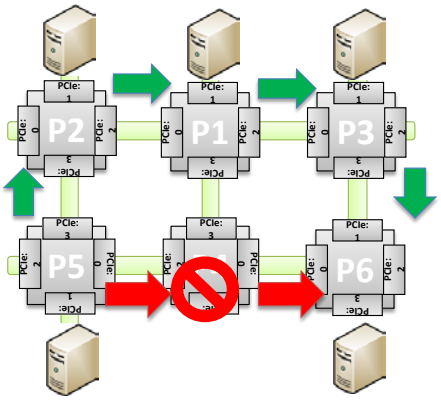


(b)



(c)





(d)

Six-node prototype

The PEARL evaluation system (a), PCI Express logical layers (b), a PCI Express x4 host adapter board (c), Six-node prototype of a PEARL network system (d).

Figure 4.11 Prototype of PEARL network system.

4.4.2 Switching time of PCI Express up-configuration function

Table 4.4 shows the measurement results of the PCI Express up-configuration function switching time. Whereas the time required to increase the lane speed is 6.5 μs, the time required to decrease it is 3.8 μs, because the PCI Express PHY needs extra time to gain lane speed (Table 4.4 (a)). Whereas the time required to shrink the number of lanes is 4.6 μs, the time required to expand it that is almost 9.1 μs (Table 4.4 (b)). The minimum latency of DMA transfer between PEACH chips that is also measured in this evaluation system is 1.0 μs. Thus, PEACH can perform power-aware control with fine-grained operation.

(a) Switching time of the lane Speed

Lane Speed	Time [us]
2.5Gbps → 5.0Gbps	6.5
5.0Gbps → 2.5Gbps	3.8

(b) Switching time of the number of lanes

No. of lanes	Time [us]		
To \ From	1	2	4
1	---	4.6	4.6
2	9.1	---	4.6
4	9.1	9.0	---

Table 4.4 PCI Express up-configuration function-switching time.

4.5 Related Works

Lack of low power and high performance network technology for dependable embedded system is obstacle. As mentioned in this section, Infiniband is frequently used as a low latency and high bandwidth network for high performance computing clusters, but its large power dissipation prevent adapting for embedded systems. GbE is another candidate for low cost networks but it does not match performance. RI2N (Redundant Interconnection with Inexpensive Network) is a fault tolerant and high performance interconnection network based on the multi-link of GbE [okamoto07]. The power consumption of a controller chip for GbE is much smaller than that of Infiniband. RI2N is needed to

enhance throughput to adapt embedded networks, which causes larger power consumption. Moreover, GbE is essentially for long distances; therefore the communication latency using GbE is relatively large. Therefor the conventional networks described above are not appropriate for low-power and high performance network.

Another candidate for a PCIe-based network for dependable embedded systems is a standard specification called ASI (Advanced Switching Interconnect) which interconnects multiple host computers and I/Os. [asi03, dolphinics]. ASI's target is an interconnected 68-node network and it requires more complicated hardware than PCIe. ASI is mainly used for server I/O connections, and it is not applicable for dependable embedded systems.

4.6 Summary

PEACH and PEARL open up new possibilities for a range of communications by extending PCI Express packet transmission to internode communication. PEACH's performance advantage, power awareness and high dependability are the result of the combination of PCI Express, the intelligent ICU and the multicore processor. We're currently improving and expanding firmware including drivers and user communication libraries. PEARL resulted from the research area of DEOS project [deos]. The PEACH board is positioned as a hardware platform for DEOS project and is expected to be adopted in many dependable high-end embedded systems, which will spur upgrades to technology innovations in this area.

Chapter 5

A Heterogeneous Multicore SoC for Secure Multimedia Applications

5.1 Introduction

Digitalization of media has spread rapidly and music and images are also becoming more highly defined. As a way to easily distribute these digital contents, not only the disc packaging such as CDs, DVDs and blue-ray discs, but also network delivery services are gaining popularity. To further expand network delivery services, we have to establish secured accounting systems.

Digital content protection standards such as DTCP-IP, Windows Media DRM (Janus) and Broadcast Flag have been established. However, in each case, decryption software is executed on non-secure hardware. As a result, a vulnerability arises in which an encryption key can be disclosed or code can be easily modified to access data without authorization.

In a secured accounting system, we have to download encrypted contents from a content server, and a decryption key from a payment server so that decoding and payment can be performed in a secure multimedia processor. The decoded data in the processor can be played on a digital TV or a music player. Therefore, we need to develop a system that processes the decoding and the payment atomically.

In a conventional system, the decryption and decoding operations are performed individually on different chips. When the encrypted contents are delivered, they are decrypted and restored to their original plain data format using the decryption key. Subsequently, the video data is decoded and images and audio are sent to audio/video output.

However, we currently have a system problem that decryption key and decrypted contents are at risk for being stolen. Because decryption software is executed on non-secure hardware, the decryption key and decrypted contents could be disclosed without authorization.

To realize a secure system, the best solution is to integrate all components in one chip. But, this is difficult to achieve with current silicon-process technology to at a reasonable cost.

To solve these security and cost problems, a multicore SoC with SiP (System in a Package) technology and an evaluation system (Figure 5.1) has been developed. In this paper, we propose a novel secure system using our SoC and software solution.

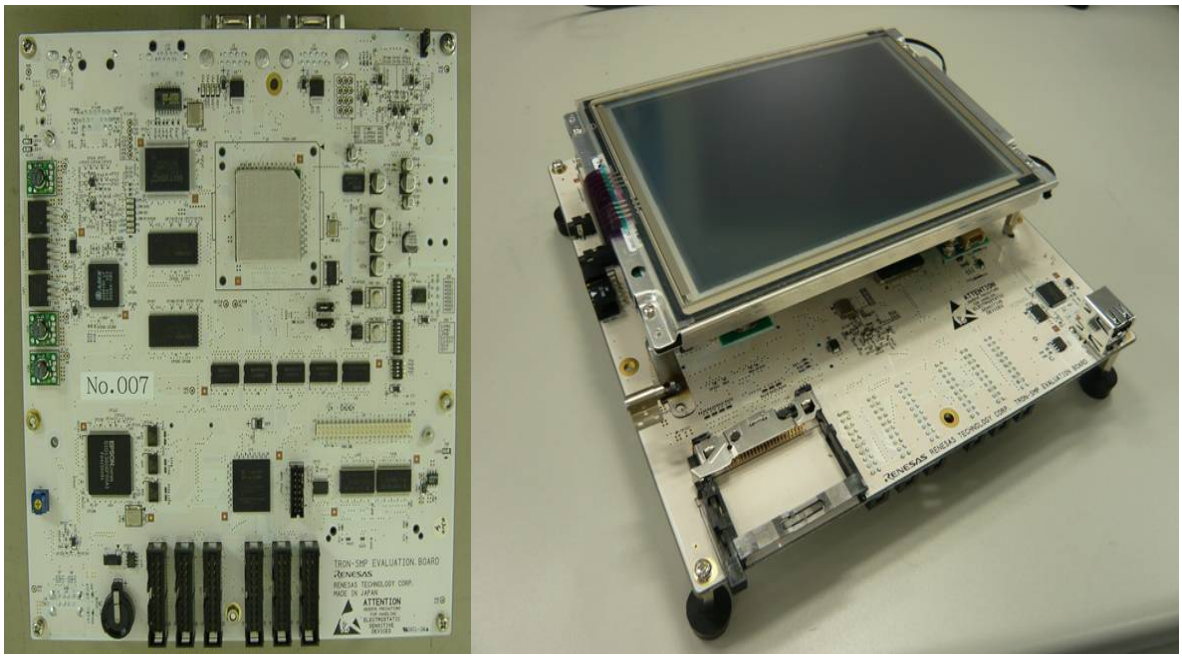


Figure 5.1 Implemented secure media system board

5.2 Secure Media System

5.2.1 Concept of the secure media system

The proposed concept (Figure 5.2) consists of the following.

1. Atomic operation of payment and viewing
2. Multicore SoC and SiP for faster communication and decryption
3. Hardware / software virtualization for strong security

1) Atomic operation of payment and viewing

The problem with a conventional system is that payment, decryption and image processing are themselves large monolithic side-attack targets. Atomic operation of these processes eliminates problems of payment omission and copyright infringement from the illegal copying of data. In addition, the multicore SoC with SiP provides both tamper resistance and high performance because all communication routes are wired in the chip.

2) Multicore SoC, DRAM, and Flash memory in one package (SiP) for faster communication and decryption

Faster communication between external devices and faster decryption are indispensable when dealing with digital contents including motion video formats like MPEG. A multifunction motion video decoder is integrated on the heterogeneous multicore SoC to be compatible with MPEG-2/H.264/VC-1 on DTV (digital television) and DVD (digital video disc). A symmetric-key cryptography accelerator for decoding multimedia contents and a public key encryption IP for payment and user confirmation are also integrated (Figure 5.3).

3) Hardware and software virtualization for strong hardware/software security

To achieve a secured system, this SoC virtualizes hardware resources and an OS (Operating System) and applications are prohibited from accessing hardware resources directly. The most distinguishing feature of the system is that the multimedia block and the secure block are isolated and communication between these blocks is executed on the virtualization layer.

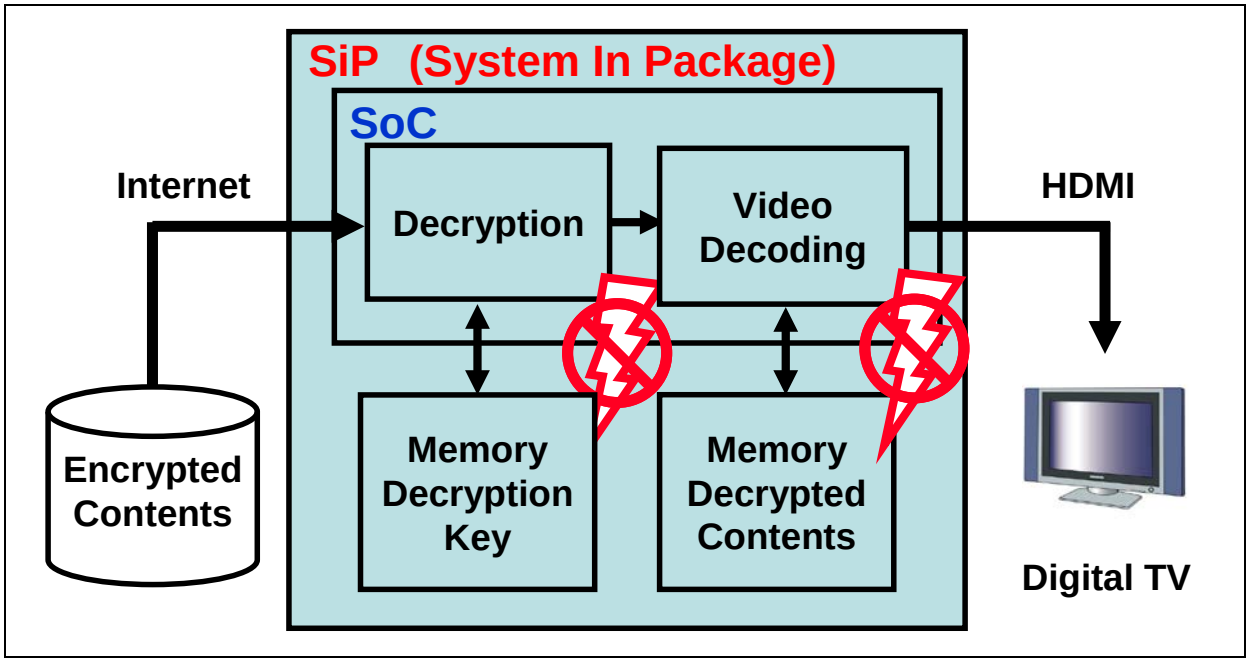


Figure 5.2 Concept of the secure media system.

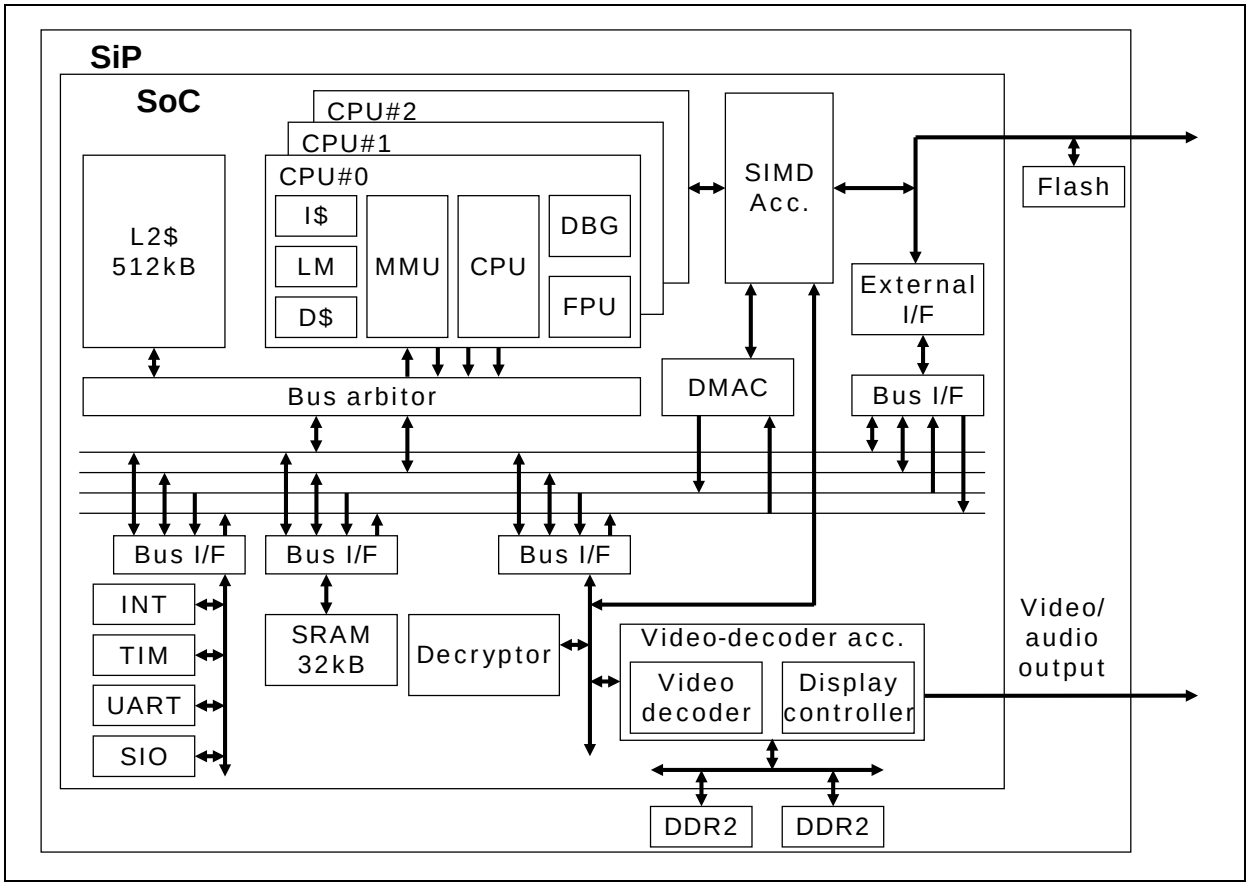


Figure 5.3 Block diagram of the SoC

5.2.2 SoC Overview

We have developed the SoC that adopts heterogeneous multicore architecture. Table 5.1 and Table 5.2 show the system block diagram and the functional features, respectively. The multicore SoC we have developed integrates three types of processing units: two specific-purpose accelerators for a decryption and a high-resolution multifunction video decoder; one general-purpose SIMD accelerator for image filtering [noda07]; and three CPUs for data-flow control and data processing.

The three CPUs are connected to a common pipelined bus with a cache coherence mechanism [kaneko04]. Each CPU is 32-bit RISC architecture and a synthesizable processor, which includes a double floating point processing unit, a memory management unit, three 8kB memories for level one instruction cache, level one data cache, and local memory, and a debug module. The CPU is a 7-stage dual-issue pipelined processor.

The CPU block is a three way conventional SMP from a coherence perspective and supports a unique CPU grouping mode that divides CPUs into several groups [kondo08]. CPUs can only be snooped from other CPUs in the same group. The pipelined bus, which supports the modified, exclusive, shared or invalid (MESI) protocol, is connected to a 512kB L2 cache. This pipelined bus with a large bus width (a 128-bit read bus and a 32-bit write bus, separately) reduces internal bus traffic and is directly connected to the multi-layer system bus.

These three types of processors are interconnected on this chip with a high-bandwidth multi-layer system bus. Three CPUs communicate via this multi-layer system bus through the L2-cache. An embedded SRAM, internal I/O, special-purpose accelerators and the general purpose SIMD accelerator, are all connected by this multi-layer system bus. This multi-layer system bus provides a sufficient transfer rate by accessing these resources in parallel.

CPU	32-BIT RISC PROCESSOR(270MHZ) X 3 SMP L1-CACHE:8KB(I)+8KB(D),LM:8KB, MMU, FPU
Memory	**L2-cache : 512kB** **Internal SRAM : 32kB**
General purpose accelerator	**SIMD Processor (270MHz)** **2b-PE x 640, I-SRAM : 32kB, D-SRAM : 80kB**
Video-decode accelerator	**Decoding feature :** **MPEG-2 MP@HL, MP@ML** **H.264/AVC (MPEG-4 AVC) HP@L4.1, MP@L4.1** **VC-1 AP@L3** **Resolution : 1920 pixels x 1080 lines**
Decryption accelerator	**AES-CBC 128-bit, AES-CTR 128-bit,** **AES-CMAC 256-bit**
Bus	**Multi-layer bus (4-layer)** **Pipelined bus/Fly-by bus**

Table 5.1 Functional features of the SoC

TECHNOLOGY	90NM GENERIC CMOS (8 LAYERS)
Chip Size	**6.35 x 6.35**
Clock frequency	**Internel: 270MHz max** **External bus: 135MHz**
Power supply	**Core: 1.0V , I/O: 3.3V , DDR2: 1.8V (Vref=0.9)**
Power consumption	**2.0 W**
SiP	**29 x 29 mm2 729pin FCBGA** **4 chips in a package** **- Multicore SoC : 8.00x8.00mm2** **- DDR2 SDRAM : 256MBx2, 8.39x8.58mm2** **- Flash Memory : 32MB, 5.74x7.64mm2**

Table 5.2 Physical features of SoC and sip

5.2.3 Physical Integration of the SoC and the SiP

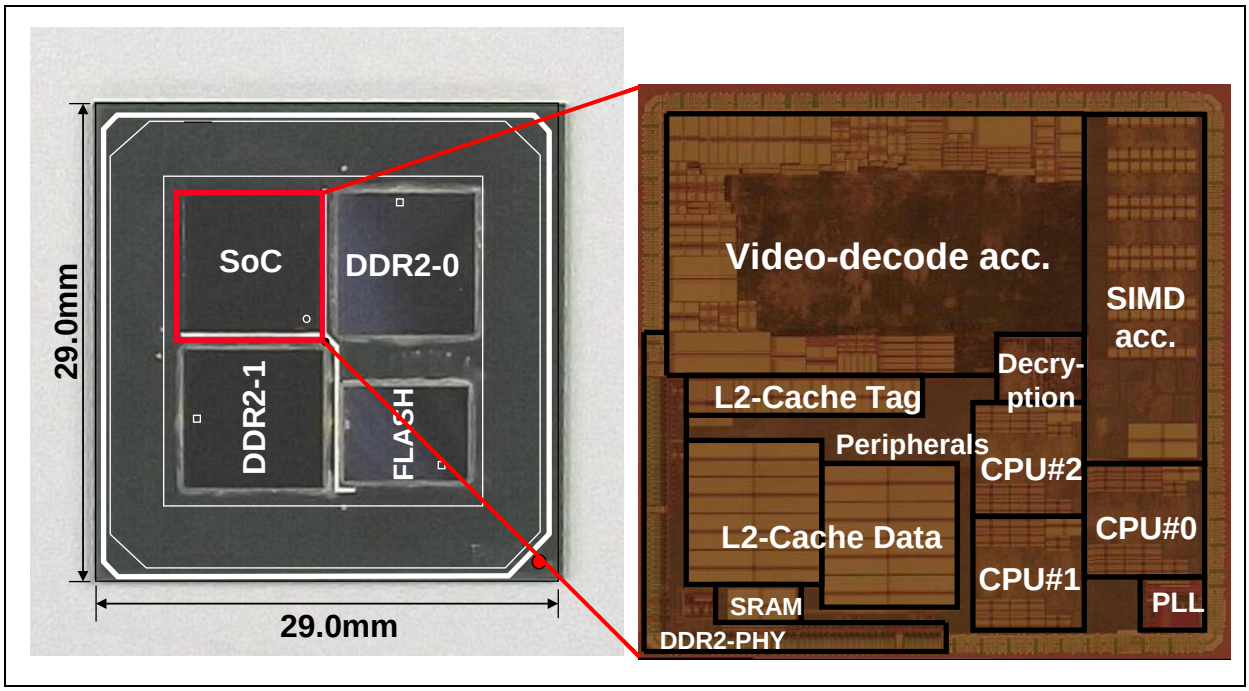


Figure 5.4 Micrograph of SoC and SiP.

Figure 5.4 shows physical features of the SoC and the SiP. All logic circuits on the chip were constructed using standard cells except for the memories, DDR-PHY, and PLL. The chip was fabricated in a 90nm generic CMOS-process with eight layers of copper interconnects. The chip integrates 4.35 million gates and 1.1MB of memory in a 6.35mm x 6.35mm logic area. The die is packaged in a Flip-Chip Ball Grid Array with 729 pins.

The SoC, two DDR2 SDRAMs and a Flash memory are enclosed in this package. The size of the package is 29mm x 29mm (Table 5.2). The DDR2 SDRAMs are placed symmetrically against the SoC so that the length between the SoC and each memory is equal. We adopted flat structure for the package because of low power consumption and security.

5.2.4 Protection by Software

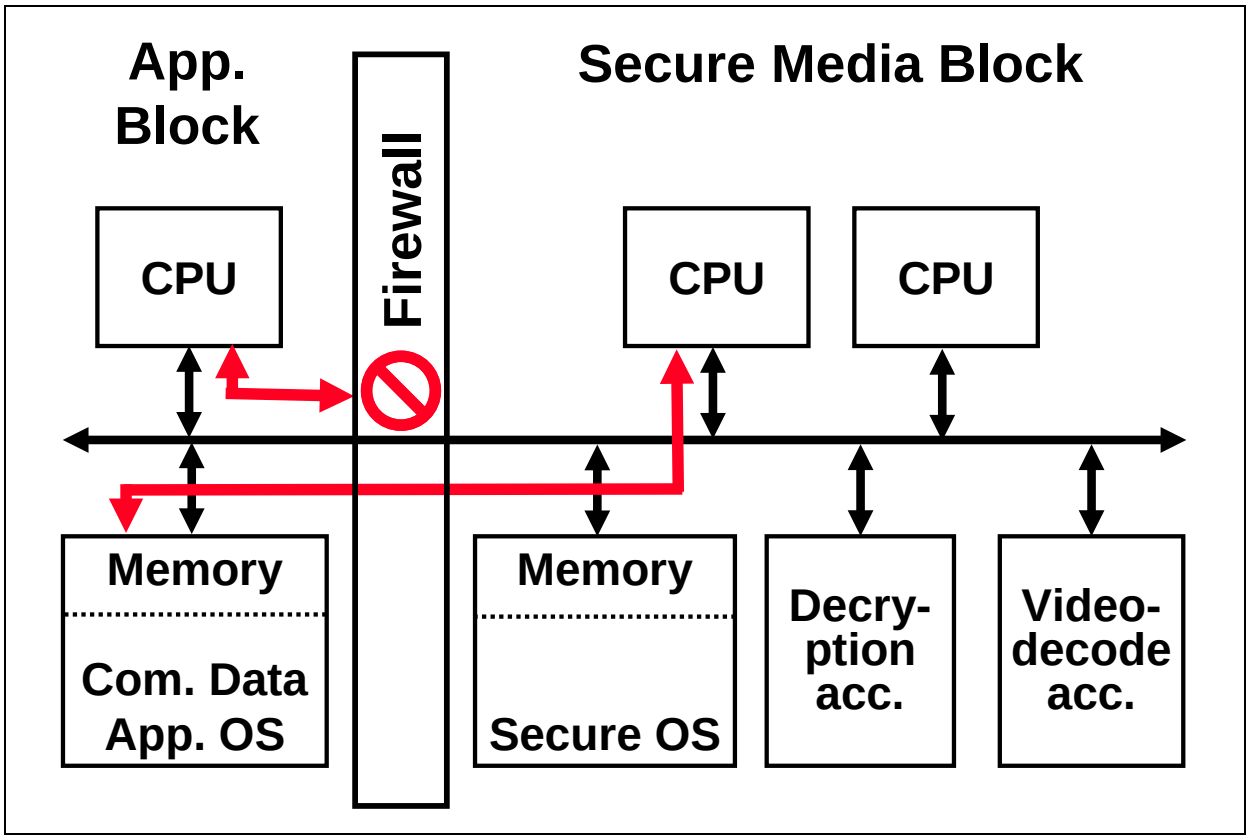


Figure 5.5 Protection by software

In order to safely decrypt encrypted contents downloaded off the Internet, this multicore system is divided into two blocks from a software point of view. One is the secure media block and the other is the application block. In each block, each OS runs independently. This technology is the micro clustering model in which multiple CPUs are divided into groups and multiple OSs run on each group simultaneously.

The secure media block consists of a video decode accelerator and two CPUs and other modules. A Real Time OS (RTOS) atomically executes decryption, image decoding and display processes. In the application block, a general purpose OS such as Linux handles the file management, GUIs (graphical user interfaces) and network operations. User applications also run on the general purpose OS.

To isolate the secure media block and the application block effectively, we set up a firewall between the secure and the application blocks using software (Figure 5.5). The multicore hypervisor is software which can provide the operating system with virtual hardware or limit its access to memory. The multicore hypervisor also prevents application programs and the application OS from accessing

memory on the SiP. In this system, communication between the secure OS and the application OS is realized by calling the OS communication API (application programming interface), which is the fully protected pathway. Encrypted data obtained by software in the application block is passed to the secure block using this secure OS communication API.

The cooperation of the multicore hypervisor and the micro clustering model improves tamper-resistance. This software system, which consists of the multicore hypervisor and the micro clustering, is described in Chapter 5.3.

5.3 Multicore Hypervisor

5.3.1 Micro Clustering Model

The “micro clustering model” is a novel technique that organizes multiple CPUs into groups with multiple OSs running on each group simultaneously. CMPs (chip multiprocessor) have been widely used, because they can achieve both high performance and low power consumption. SMPs, which consists of identical processors, are established in server-class applications using multithreading in order to increase throughput. In embedded CMP systems, not only SMP technology but also an asymmetric multicore approach is vital to maximize performance.

To reduce inter OS communication overhead of the asymmetric multicore system, we designed hardware software system in which OSs share hardware resources except CPUs while keeping hardware costs low. And we call this hybrid operating system platform “micro clustering model”, in which multiple processors are in a single chip and multiple OSs share key components. In this system, RTOSs and general purpose OSs run independently on each processor. The multiple operating system platform can make the appropriate allocation of roles such as a general purpose OS and a RTOS among multiple OSs.

The multicore hypervisor is software for a multicore system that can provide the OSs with virtual hardware and a way to communicate with each other OSs (Figure 5.6). When the system configuration is changed, it supports standardizing interfaces on logical models in order to smooth out hardware discrepancies.

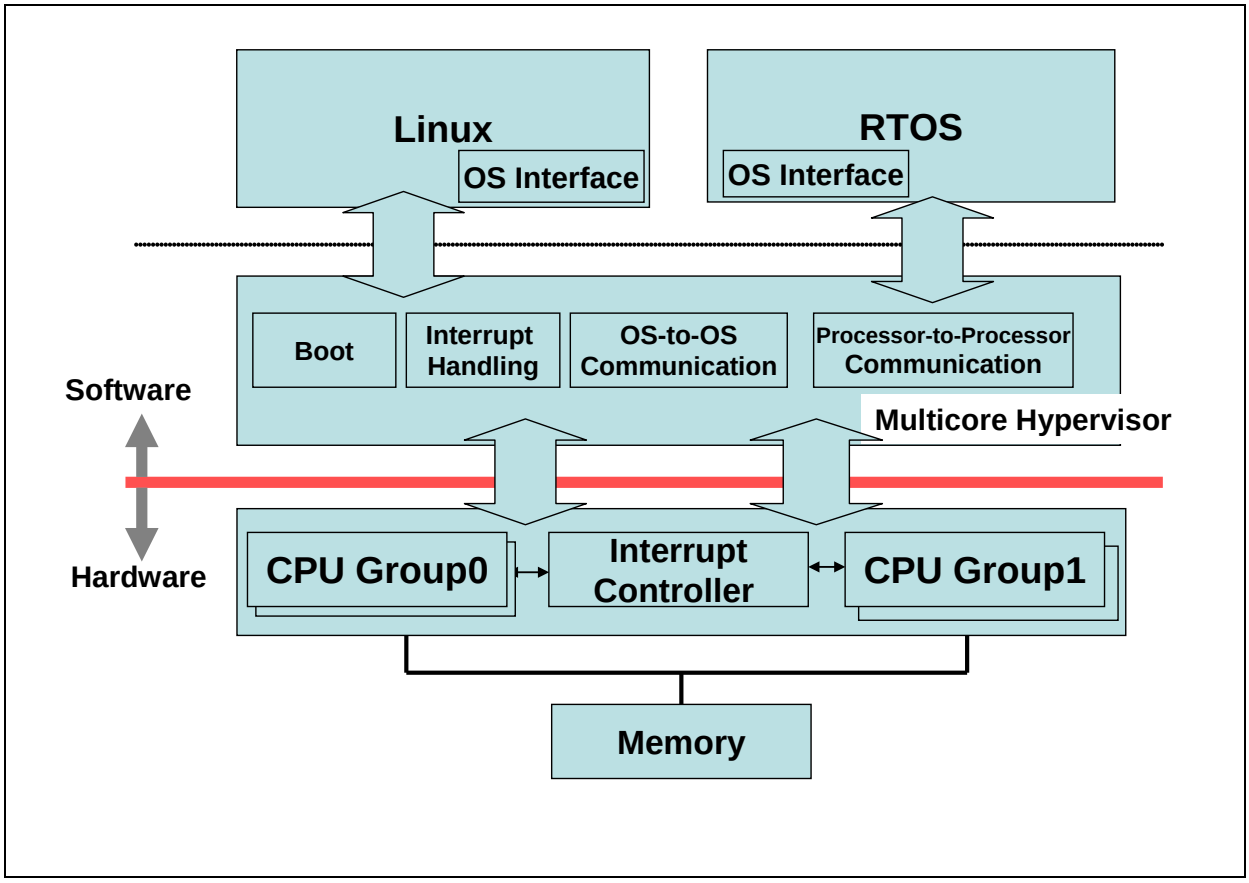


Figure 5.6 Multicore hypervisor and micro clustering model.

The multicore hypervisor is based on a para-virtualization model, which is employed to improve performance by presenting each VM (virtual machine) with an abstraction of the hardware that is similar but not identical to the underlying physical hardware. Not fully emulating hardware results in low performance overhead.

5.3.2 Functions of the multicore hypervisor

The multicore hypervisor manages the collaboration and cooperation of the processors. OSs execute the rest of the functions directly. The multicore hypervisor provides the following functions:

1. Startup: OS startup on multiple processors, control of a startup sequence and startup timing
2. Communication/synchronization: inter-OS communication and synchronization on multiple processors.
3. Interrupt handling: external interrupt distribution to multiple processors.

5.3.3 Startup sequence

The startup controls boot operations of the OSs on the processors. The configuration of the multicore hypervisor organizes multiple processors into groups with multiple OSs, startup sequence and startup timing for each processor.

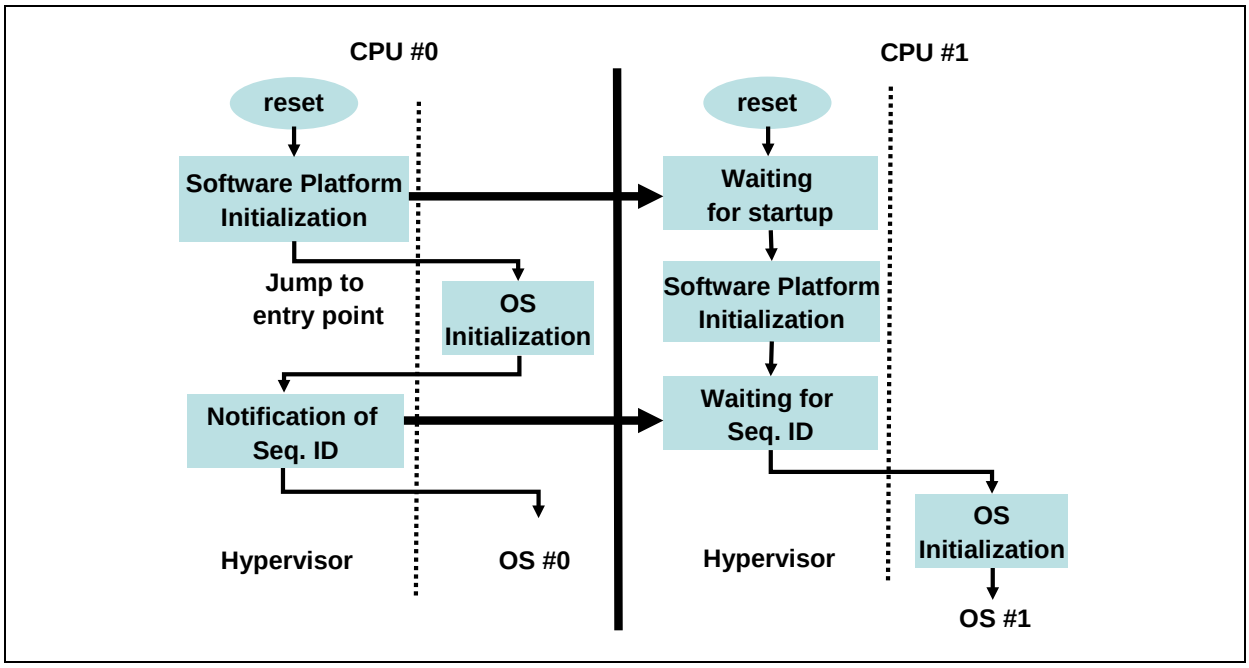


Figure 5.7. Startup sequence.

Figure 5.7 illustrates the 2CPU startup sequence. First, the multicore hypervisor executes hardware initialization for its operation (software platform initialization). Next, the multicore hypervisor jumps to the entry point of OS#0. The multicore hypervisor does not execute the bootstrap loader. So, when needed, the entry point must be the start address of the bootstrap loader. Then, OS#0 starts running on CPU#0.

The startup timing control of OS#1 is realized by identifying the waiting sequence ID. CPU#0 sends completion signal of the waiting sequence ID to CPU#1.

The processor status after a reset is handled in two ways:

A) One processor starts operating and the other processors remain at reset
The running processor activates the other processors.

B) All processors start operating simultaneously

Even if all processors start operating simultaneously, the specific predefined processor makes on-going requests to other processors to synchronize.

5.3.4 Inter-OS Communication

For inter-OS communication and synchronization, the multicore hypervisor supports two functions:

A) Pipe Communication

Pipe communication using a simplex byte stream provides 1 to 1 OS communication services. The number of pipes is determined by the system configuration of the multicore hypervisor. These N pipes have the pipe IDs from 0 to N-1. The data written to a pipe is sent to the OS on the other end via a pipe buffer in the shared memory. All service-calls of the multicore hypervisor use non-blocking communication. Therefore, event notifications to OSs are in call-back mode.

B) Semaphore Communication

Semaphores are a synchronization mechanism in OSs. The number of binary semaphores is determined by the system configuration of the multicore hypervisor. These N binary semaphores have semaphore IDs from 0 to N-1. Control data is placed in the shared memory and is operated by the multicore hypervisor on each processor.

Pipe communication requires pipe control data and pipe buffers to be shared by processors. This system has 8KB pipe buffers. Semaphores have semaphore control data as inter processor shared data. Both pipes and semaphores use common event mechanisms of the multicore hypervisor. These event mechanisms realize inter processor notifications using inter processor interrupts.

5.3.5 Interrupt handling

In a multiple OS environment, external interrupts like IO interrupts need to be sent to the OS to which they are assigned. Sometimes multiple IO interrupts share one interrupt request when hardware resources are limited. To accommodate this requirement, the interrupt handling on the multicore hypervisor supports a function that transfers interrupt request accepted by the processor to the other processors. The inter OS communication are operated by the inter processor interrupt services in the multicore hypervisor.

Figure 5.8 illustrates the interrupt control procedure. The interrupt control table defines which OS is associated with each interrupt cause. This table is placed in the shared memory.

When an interrupt occurs, the multicore hypervisor accesses the interrupt control table and determines if it needs to handle the interrupt or not. If so, the multicore hypervisor jumps to the hypervisor interrupt handler. If not, the multicore hypervisor checks the OS ID that is associated with the interrupt cause. When the OS ID indicates an OS that runs on this processor, the multicore hypervisor calls the OS's interrupt handler. When the OS ID indicates a OS that runs on another processor, it notifies the other processor that is associated with the interrupt cause using the inter processor interrupt.

The interrupt services are implemented via the hook method in order to minimize OS modifications. The OS sets interrupt vector addresses on the processor using service calls to the multicore hypervisor. When the hypervisor interrupt handler calls the OS's interrupt handler, the multicore hypervisor uses the interrupt vector address sent by the OS.

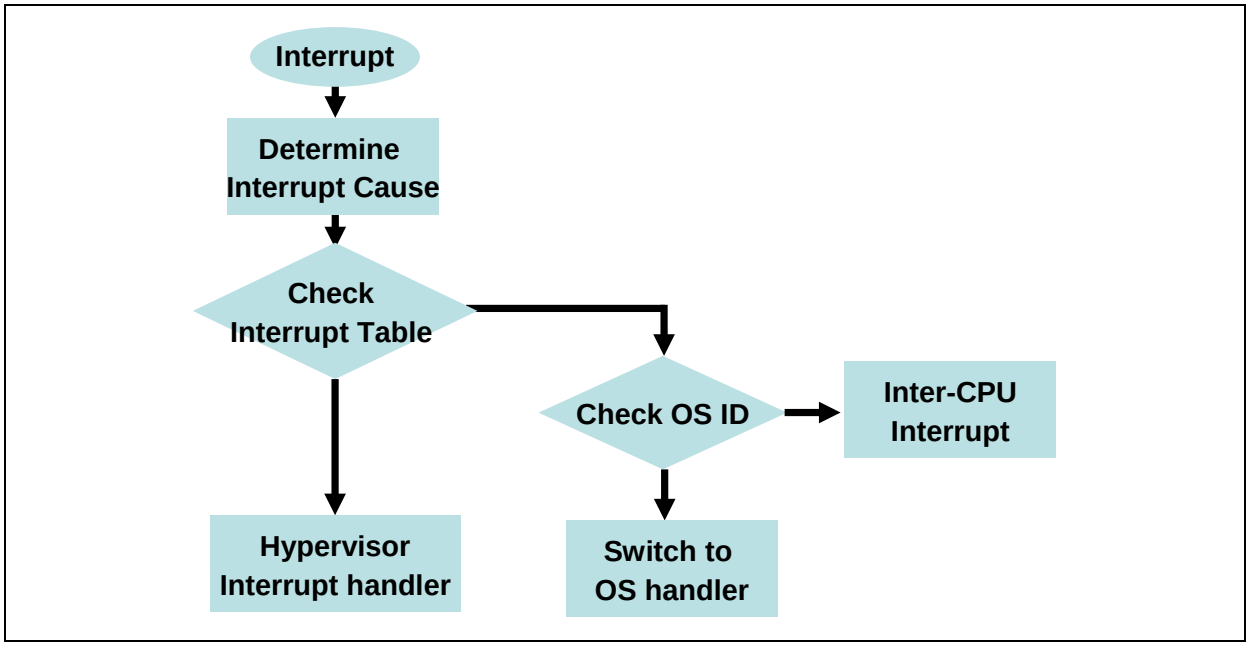


Figure 5.8. Interrupt operation.

5.3.6 Hypervisor Operating System

Three types of OSs run on the multicore hypervisor: Linux (2.6.18), T-Kernel (1.01), TOPPERS/JSP kernel (1.4.2). Any combination of these OSs is theoretically operable. The combination that is shown in エラー! 参照元が見つかりません。 has been verified on our system.

T-Kernel is a real-time operating system for T-Engine project [tengine]. The TOPPERS/JSP kernel is a real-time kernel that is in conformity with the μITRON4.0 specification [toppers].

PROCESSOR #0	PROCESSOR #1
Linux	**Linux**
T-Kernel	**T-Kernel**
T-Kernel	**Linux**
TOPPERS/JSP	**TOPPERS/JSP**
TOPPERS/JSP	**Linux**

Table 5.3 Combinations of Supported OSs

5.4 System Software

5.4.1 Software Architecture

Figure 5.9 shows the software architecture and the micro clustering architecture of the system. The software architecture consists of four layers. From the bottom up, they are the hardware layer, the hypervisor layer, the OS layer and the user layer, which includes the main task, the video task and the audio task. The multicore hypervisor is on the hypervisor layer, which is a thin software layer inserted between the hardware and the OS layer.

The secure media block and the application block are effectively isolated by a firewall. An application program such as a player with a GUI reads the content data and sends it through the secure interface. The secure interface which is strongly encrypted is the only pathway between the application block and the secure media block. Undetectable malware on the application block cannot tamper with the secure block’s status or attack resources on the secure media block. In addition, the multicore hypervisor can allocate hardware resources like memory between virtual blocks. Therefore, it provides the flexibility to adjust the system to the most suitable combination for an application’s demands.

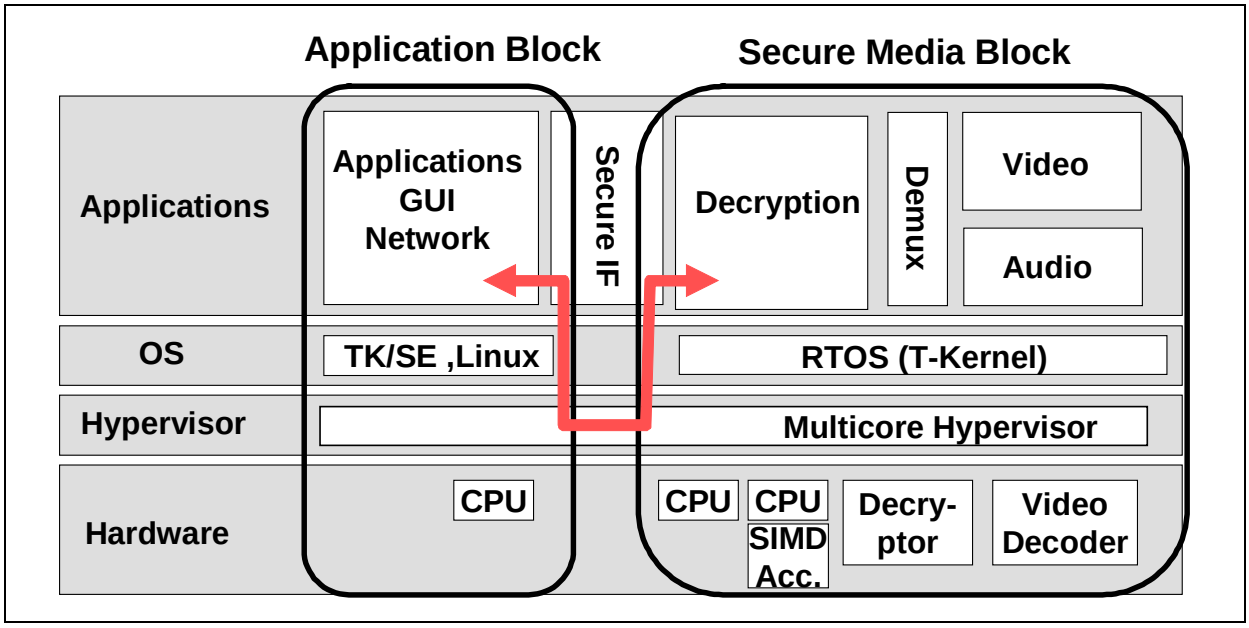


Figure 5.9. Software layer configuration

5.4.2 Secure media block software

Figure 5.10 illustrates the task structure in the secure media block. The main task of the secure media block receives content data and the payment information from the application block which constructs the secured VPN (Virtual Private Network) with content servers.

The main task in the secure media block consists of four major tasks: media task, demux task and video decode task, audio decode task.

The main task invokes the media task and the media task invokes the demux task. The media task receives 2KB of content data in one read from the Internet via a task in the application block, and decrypts it using the decryptor. The decrypted plain data is sent to the demux task. The demux task recognizes the file format and separates data into video data and audio data. The demux task starts the video decode task and the audio decode task and sends them the separated data. The video decode task sends data to the video decode accelerator. The audio decode task decodes the audio data using the AAC decode middleware. All tasks in the secure block run on the T-kernel RTOS that handles time scheduling of tasks.

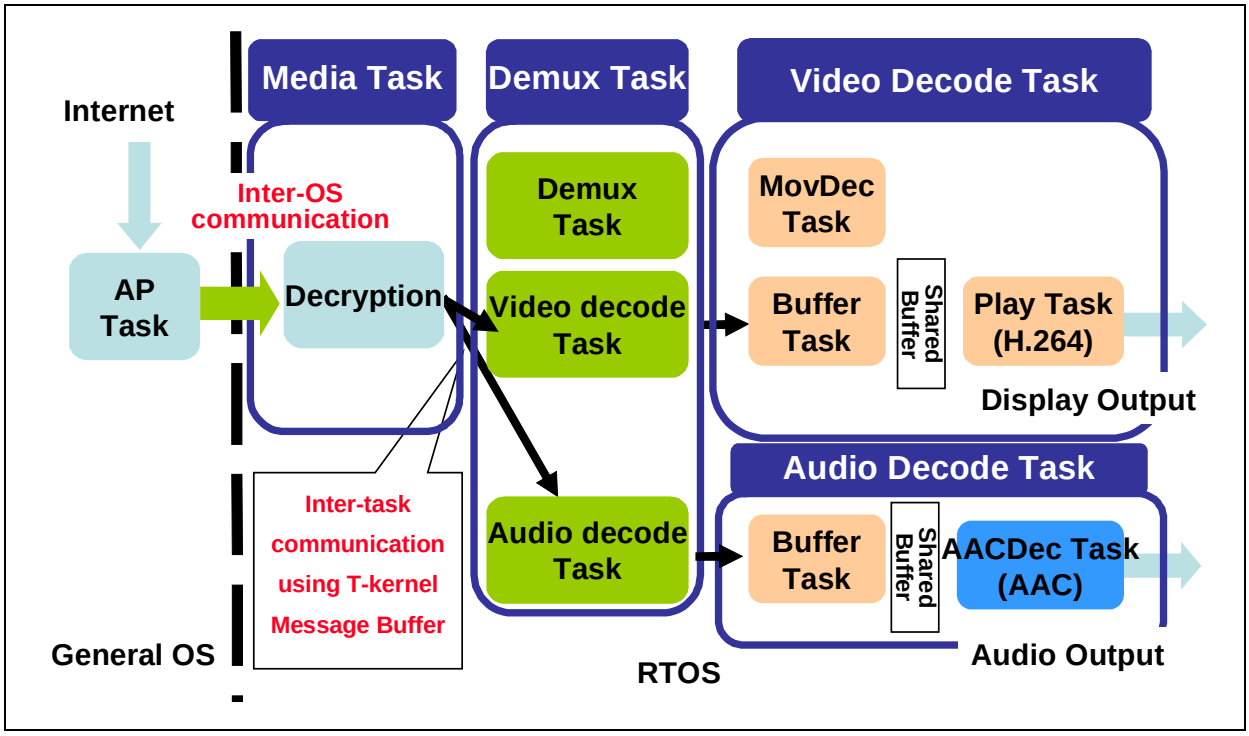


Figure 5.10. Task structure in the secure media block

5.4.3 Task mapping

In our system, one CPU is mapped to the application block and two CPUs are mapped to the secure media block. Two RTOSs run in the secure media block. Figure 5.11 shows the task mapping on each CPU. In the secure media block, AAC decode task runs on CPU#2. All tasks except AAC decode task run on CPU#1. The video data and the audio data are demultiplexed on CPU#1. The video data is sent to the video decode accelerator. The audio data is sent to AAC decode task on CPU#2 via inter OS communication and is decoded there. Figure 5.12 illustrates the decoding data flow of an encrypted MP4 container file, which consists of H.264 video data and AAC audio data.

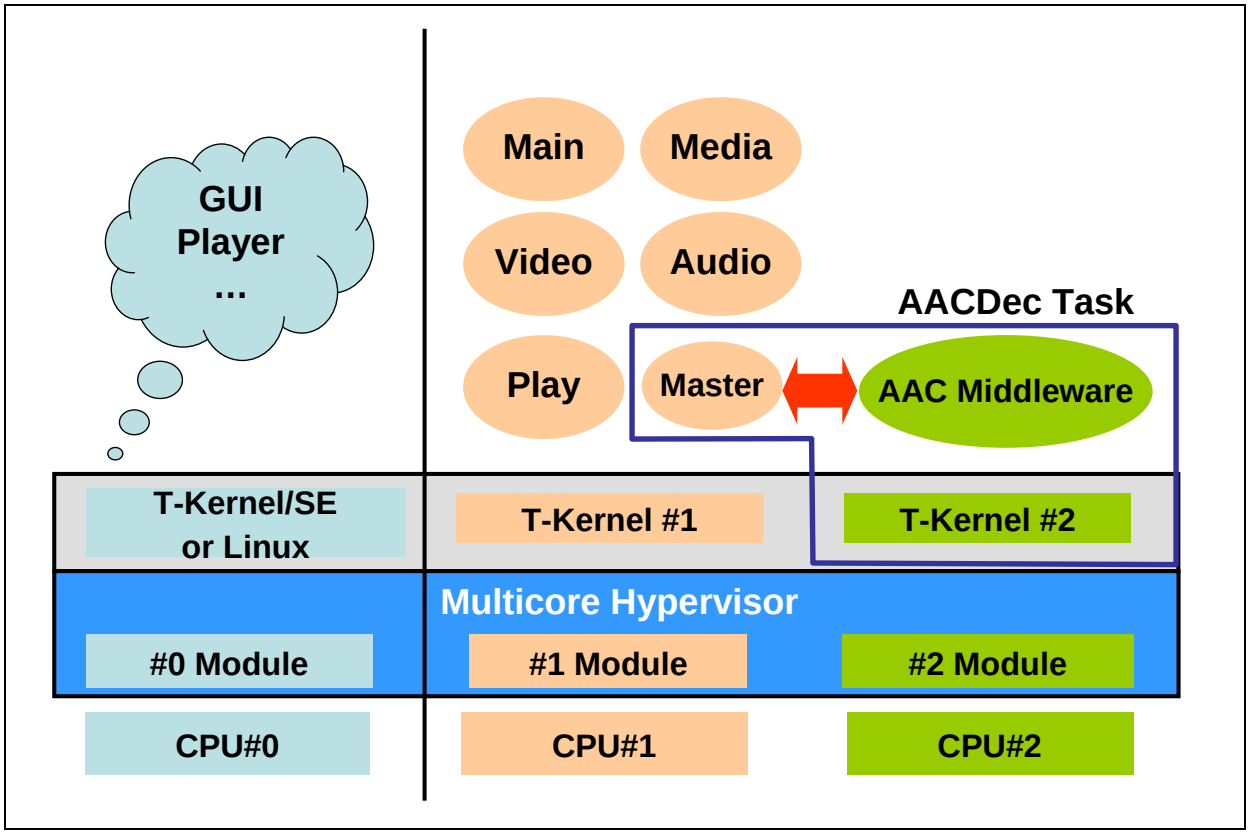


Figure 5.11. Task mapping in the secure media block

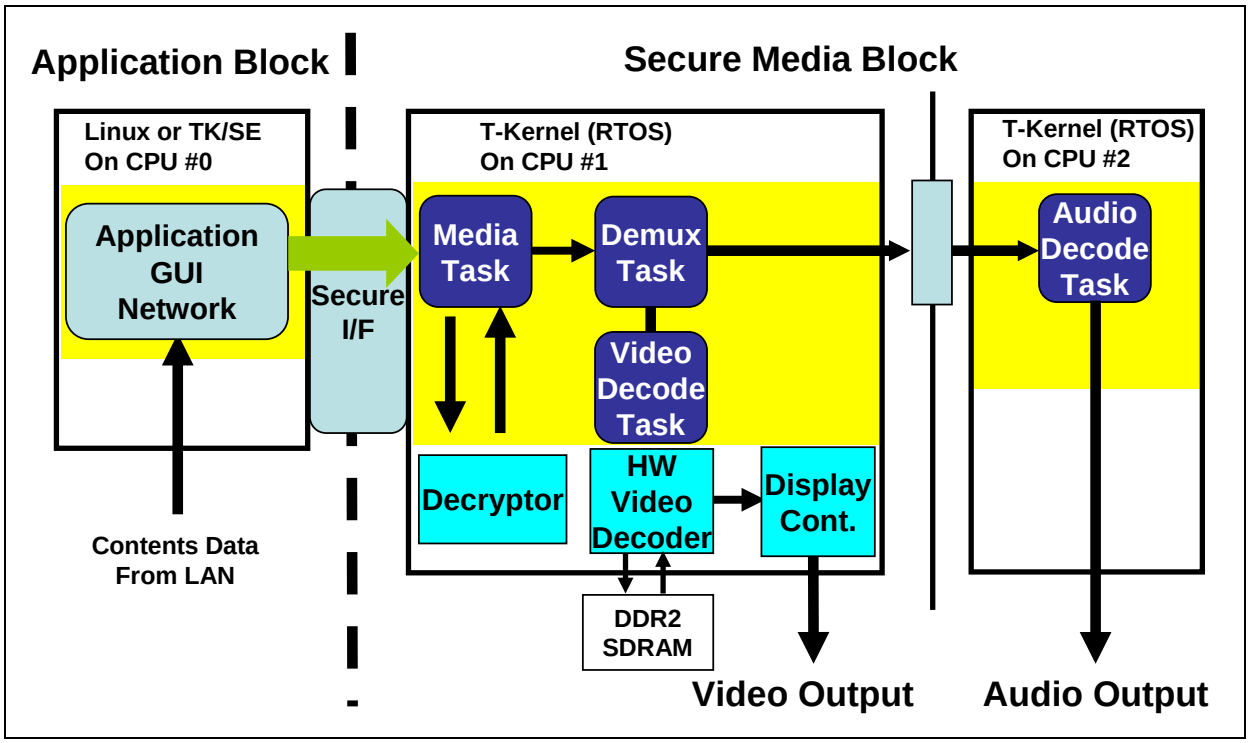


Figure 5.12. Data-flow of decoding MP4 container file.



Figure 5.13. Implemented evaluation system

5.5 System Evaluation

5.5.1 Evaluation system

Based on the component descriptions in Chapter 5.2, Chapter 5.3 and Chapter 5.4, we have developed an evaluation system which can decode encrypted high definition video data and audio data in real time (Figure 5.13). This secure media system is connected to content servers via the Internet. The encrypted contents are distributed via a private Giga bit Ethernet line.

Figure 5.14 shows the block diagram of the evaluation system. The evaluation system board has the SiP, a LAN controller, a USB controller and an LCD controller. The touch screen and the LCD panel provide a man-machine interface. The HD video images are displayed via HDMI interface.

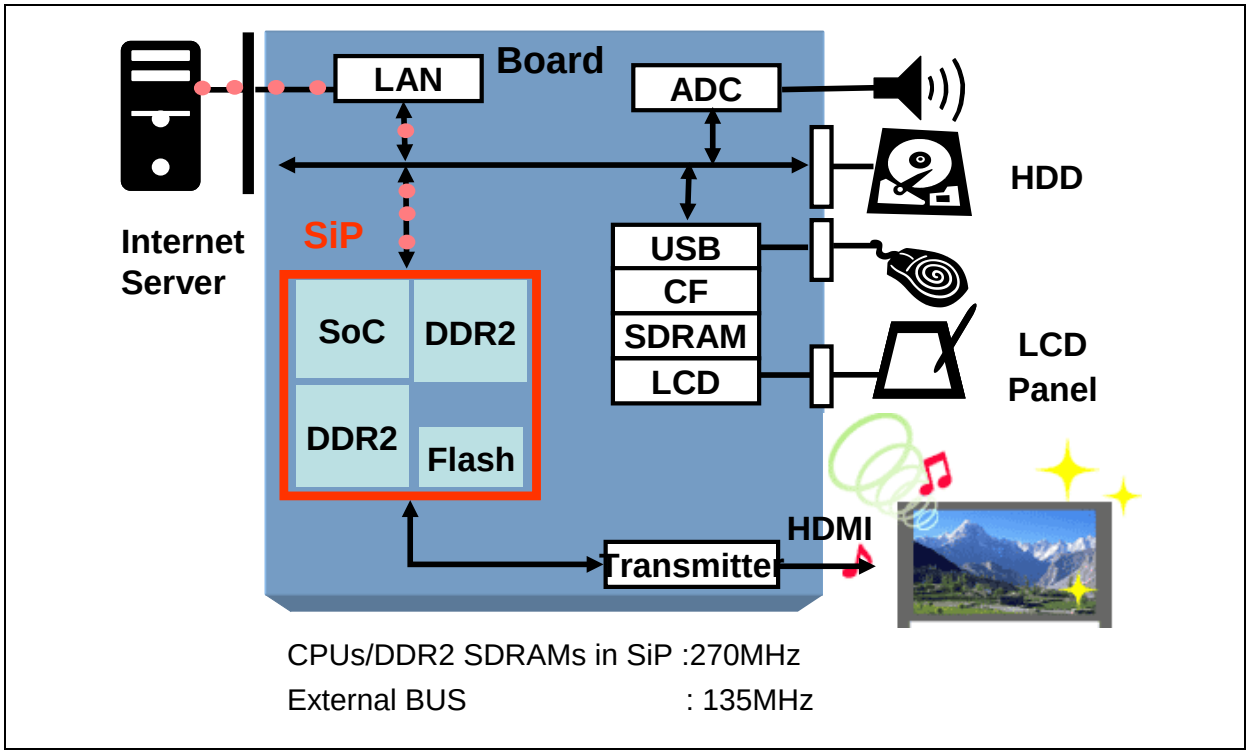


Figure 5.14. Block diagram of the evaluation system

5.5.2 Evaluation results

To analyze the workload of the tasks, we evaluated the 2-CPU version that one CPU was mapped to the application block and one CPU was mapped to the secure media block. The 2-CPU version puts CPU#2 into suspend mode and all tasks run on CPU#1. The comparisons of the workload of CPU#1 in the secure media block when playing a HD MP4 content data are shown in

and Figure 5.15.

The video decode task uses the hardware accelerator to process HD contents. So the execution time of the AAC decode task is the longest task in the configuration of 2-CPU version. In the proposed 3-CPU version, this AAC decode task is assigned to CPU#2 for the load balancing. As a result, the execution time of CPU#1 is reduced by 74%. This significantly reduces the overall execution time.

Figure 5.16 shows the workload ratio of CPU#1 and CPU#2 in the original version. The CPUs at 270MHz work only 68% of the total processing time. The rest of the time (32%) is idle.

These results show the following,

1. The multicore SoC and accelerators satisfy performance requirements to play streamed HD video.
2. The multicore hypervisor has sufficient secure communication bandwidth between the application block and the secure media block.

TASK NAME	2 CPU VERSION	3 CPU VERSION
Audio Trans task	**0.1%**	**0.1%**
Media task	**31.5%**	**43.4%**
Demux task	**1.9%**	**2.6%**
AACDec task	**31.7%**	**0.7%**
Play task	**6.2%**	**12.7%**
Buffer task	**9.7%**	**14.2%**
Video decode task	**12.7%**	**18.1%**
Audio decode task	**1.0%**	**1.2%**
(Idle)	**5.2%**	**7.0%**

Table 5.4 Workload of CPU#1 in Secure Media Block

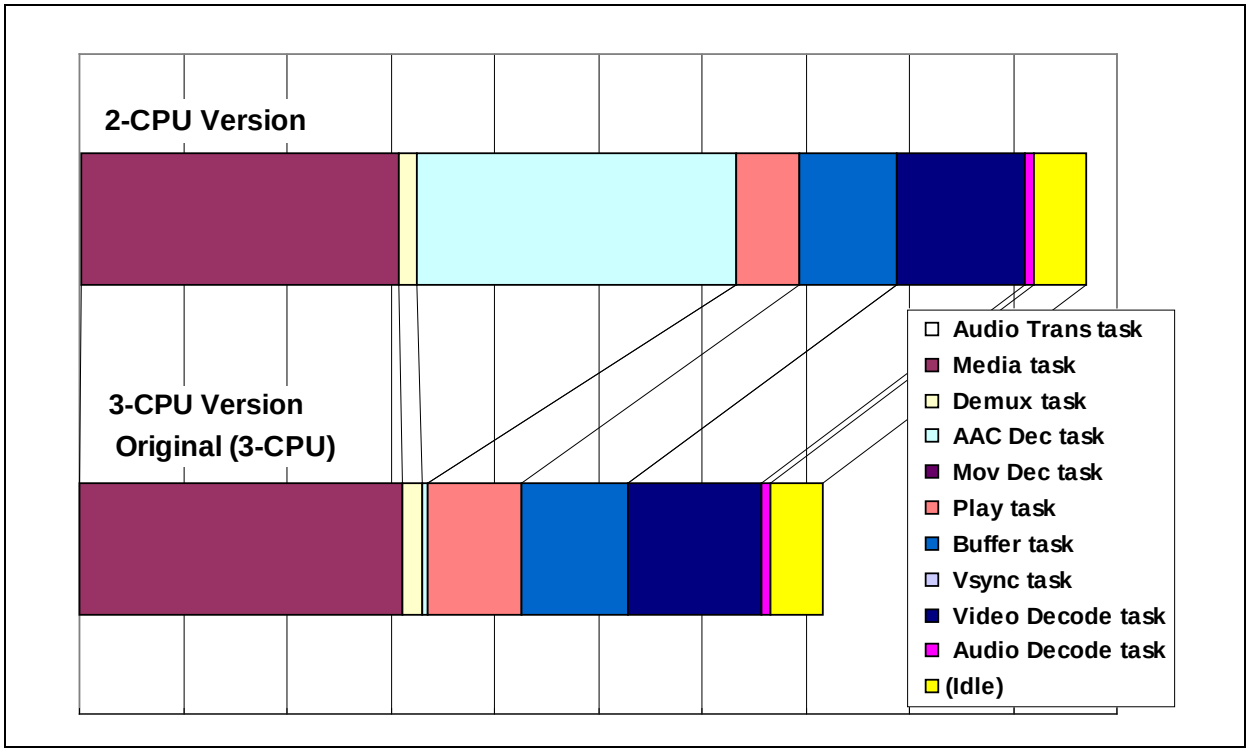


Figure 5.15. Comparison of workload of CPU#1.

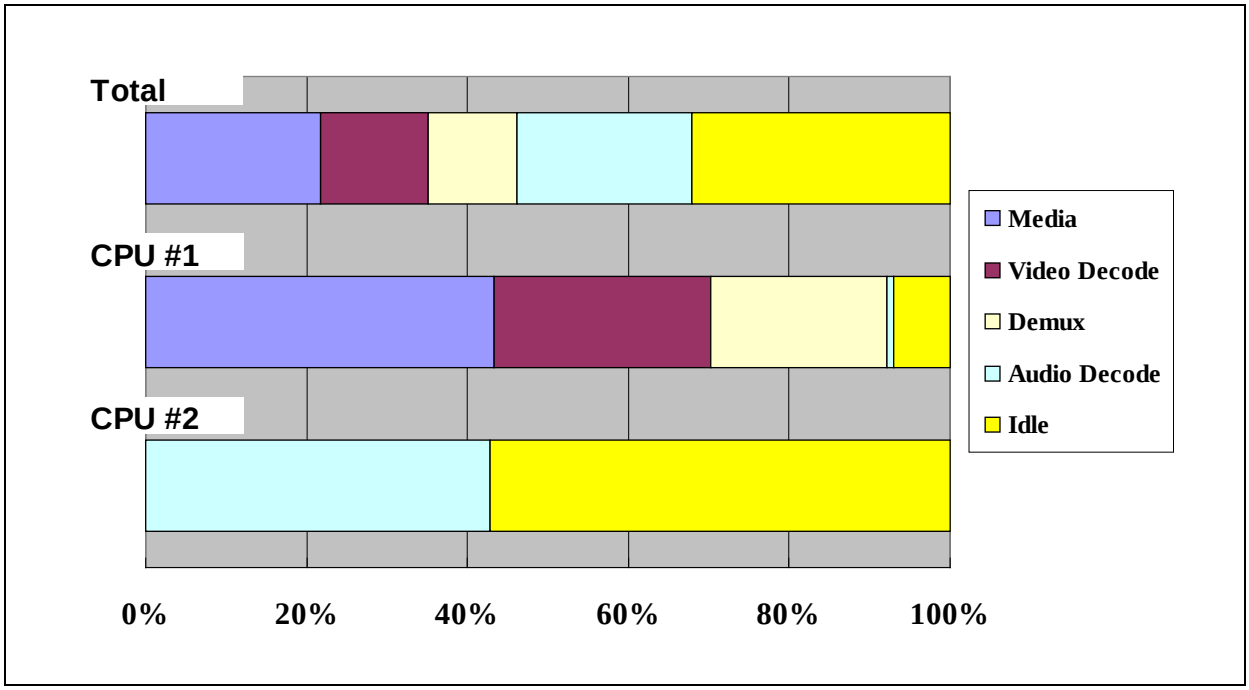


Figure 5.16. Workload balance.

5.6 Related Works

The single-chip multiprocessor is a key solution to obtaining high performance without simply increasing the clock frequency. Around 2000, thread-level parallelism (TLP) was introduced and some chips with TLP designed for embedded systems [strik00, koyama01, nishi00]. However, those multiprocessors were not available for general-purpose use, and their applications were limited. On the other hand, some single-chip multiprocessors such as Stanford hydra [hammond00], IBM Power4 [diefendorff99], with cache coherency protocol, were presented. The processor in Stanford hydra relied on invalidation-only coherence protocol and it did not meet the performance requirement. IBM Power4 employed a multichip module (MCM) package and it was not available for embedded systems because of the large power dissipation. Compared to these multiprocessors, a single-chip multicore processor, Renesas M32R, with advanced coherent cache, met both low power consumption and high performance specification [kaneko03]. Thereafter, embedded single-chip multiprocessors were inspired by this work. Renesas added SH architecture for its multicore processor lineup [ito08] and ARM produced MPcore supported by NEC Electronics [hirata07].

From the viewpoint of software development, SMP-support OSs like Linux was available. This multiprocessor was capable of running single-processor applications without modifications. That is a key feature of the single-chip multiprocessor.

A hybrid OS environment on a single-chip multiprocessor was introduced. This environment supported both a real-time OS and a general-purpose OS. These two types of OSs operated independently on a single-chip multiprocessor to improve performance of interrupt responses and application processing [endo04]. A prototype of a multicore hypervisor was already implemented to support two OSs.

Virtualization, it was originally from mainframe and server systems, has been spread to embedded systems since around 1990. Virtualization is realized by emulating virtual hardware on physical hardware in order to execute several different OSs. There are two types of hypervisor, type1 [vmwarevsphere, xenserver, hyperv, kvm] and type2 [xen]. Type1 hypervisor is suitable for embedded systems because type1 is a special OS that executes directly on physical hardware and provides better performance and a smaller code size. Thereafter, multicore hypervisor was begun to adopt to security [kondo09]. Multicore hypervisor for security is now commercialized [mentoreh].

5.7 Summary

A secure multimedia system for high-definition multimedia applications using a multicore SoC with SiP and software system virtualization has been developed. The evaluation board can decode encrypted high definition video data and audio data in real time and go directly to HDTV. Therefore, no plain content data can exit from the secure media block. This is an important feature of secure media systems.

To achieve a secure multimedia system, the multicore hypervisor virtualizes hardware resources and prohibits operating systems and applications from accessing hardware resources directly. The security of the system is the result of the cooperation between the hardware and software. The proposed system provides a solution to protect contents and to safely deliver secure sensitive information when processing billing in digital content delivery.

Chapter 6

Conclusions

IoT has become inevitable for the infrastructures of our societies.

We proposed a compact, low power processor core and its multicore approach to realize four key technologies for IoT: network technology to link one device another, technology to control sensors, motors and other devices, and low power consumption technology to raise energy efficiency and security technology.

In summary, the main contributions of this dissertation are;

- RX processor core which is suitable for IoT were introduced. The RX processor instruction set architecture and its microarchitecture can achieve lower power consumption and boost performance.
- Eight-core communication SoC with PCI Express interface were presented. The multicore SoC can realize a high performance, power-aware, highly dependable network.
- A secure multimedia system by using heterogeneous multicore SoC and software virtualization were presented.

6.1 Future work

Even though we have made significant outcomes of efficient microprocessors for IoT, there are lots of works to be done in this research area. The following are some of the key research items based on the conclusion of this thesis:

- Improved real-time performance of microcontrollers: Downsizing systems is a big trend of our society. From embedded systems' point of view, microcontrollers need to meet the demand of larger systems. Even though, multicore approach is one of the solutions, quick response in large-scale systems is a problem incapable of solution. It is necessary to explore hardware accelerations of context switch or processing of an operating system.

- Efficient hardware modules for MBD (Model Based Development): MBD began to spread in developing large-scale systems. It is interesting to explore how efficiently divide system functions described in MBD into software and hardware. It is also interesting to explore appropriate and efficient hardware modules used in this method.

References

[asi03] Advanced Switching Core Architecture Specification, PICMG, December 2003.

[bray91] B. K. Bray, M. J. Flynn, “Strategies for branch target buffers,” *Stanford University Technical Report No. CSL-TR-91-480*, June 1991.

[bryzek14] J. Bryzek, “TSensors for Abundance, Internet of Everything and Exponential Organizations,” in *TSensors Summit Munich*, 2014.

[bunda93] J. Bunda, D. Fussell, W. C. Athas, “16-Bit vs. 32-Bit Instructions for Pipelined Microprocessors,” in *Proc. the 20th International Symposium on Computer Architecture*, pp. 237-246, May 1993.

[burgess94] B. Burgess, M. Alexander, Ying-Wai Ho, S.P. Litch, S. Mallick, D. Ogden, Sung-Ho Park, J.s Slaton, “The PowerPC 603 microprocessor: a high performance, low power, superscalar RISC microprocessor”, in *COMPCON Spring '94, Digest of Papers.*, pp.300-306, February 1994.

[chihung99] C. Chi-hung and Y. Jun-Li, “Load-balancing branch target cache and prefetch buffer,” International Conference on Computer Design, 1999.

[cotterell02] S. Cotterell and F. Vahid, “Tuning of loop cache architectures to programs in embedded system design,” in *Proc. 15th International Symposium on System Synthesis*, Oct. 2002.

[deos] “Dependable Embedded OS R&D Center and DEOS Project,” Available: http://www.dependable-os.net/index-e.html

[dolphinics] “Dolphin high performance PCI and PCI Express products,” Available: http://www.dolphinics.com/products/

[endo04] Y. Endo, N. Sugai, Y. Yamaguchi and H. Kondo, “A Hybrid OS Environment on Single-Chip Multiprocessor – System Architecture -,” *Information Processing Society of Japan*, 2D-5, Mar.2004.

[gross02v] A. Gordon-Ross and F. Vahid, “Dynamic loop caching meets preloaded loop caching-a hybrid approach,” in *Proc., IEEE International Conference on VLSI in Computers and Processors*, 2002

[gross02c] A. Gordon-Ross, S. Cotterell, and F. Vahid, “Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example,” in *Computer Architecture Letters*, 2002

[halfhill09] T. R. Halfhill, “EEMBC’s Dhrystone Killer,” *Microprocessor Report*, June 8 2009.

[hammond00] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and Olukoutun, “The Stanford hydra CMP,” *IEEE Micro*, vol. 20, pp. 71–84, March 2000.

[hanawa10] T. Hanawa, T. Boku, S. Miura, M. Sato, K. Arimoto, “PEARL: Power-Aware, Dependable, and High-Performance Communication Link Using PCI Express,” in *Proc. IEEE/ACM International Conference Green Computing and Communications*, pp. 284-291. December 2010.

[hennessy06] J.L. Hennessy, D.A. Patterson, “Computer Architecture: A Quantitative Approach, 4th edition Appendix J: Survey of Instruction Set Architectures”, Morgan Kaufmann, San Mateo, CA, 2006.

[hirata07] K. Hirata, “ARM11 MPCore: The streamlined and scalable ARM11 processor core,” in *Asia and South Pacific Design Automation Conference*, Jan. 2007.

[hyperv] “Hyper-V”, Microsofot, Available: https://technet.microsoft.com/en-us/windowsserver/dd448604.aspx

[infiniband] “The Infiniband architecture specification,” Infiniband Trade Association, Available: http://www.infinibandta.org/specs/

[ito08] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, T. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, H. Kasahara, “An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler,” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp.90–91, Feb. 2008.

[kaneko03] S. Kaneko, K. Sawai, N. Masui, K. Ishimi, T. Itou, M. Satou, H. Kondo, N. Okumura, Y. Takata, H. Takata, M. Sakugawa, T. Higuchi, S. Ohtani, K. Sakamoto, N. Ishikawa, M. Nakajima, S. Iwata, K. Hayase, S. Nakano, S. Nakazawa, O. Tomisawa, and T. Shimizu, “A 600 MHz single-chip multiprocessor with 4.8 GB/s internal shared pipelined bus and 512 kB internal memory,” in *IEEE int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 254-255, Feb. 2003.

[kaneko04] S. Kaneko, H. Kondo, N. Masui, K. Ishimi, T. Itou, M. Satou, N. Okumura, Y. Takata, H. Takata, M. Sakugawa, T. Higuchi, S. Otani, K. Sakamoto, N. Ishikawa, M. Nakajima, S. Iwata, K. Hayase, S. Nakano, S. Nakazawa, K. Yamada, T. Shimizu, "A 600MHz Single-Chip Multiprocessor with 4.8GB/s Internal Shared Pipelined Bus and 512kB Internal Memory," *IEEE J. Solid-State Circuits*, vol. 39, no. 1, pp. 184-193, January 2004.

[kin97] J. Kin, M. Gupta and H. William, "The filter cache: an energy efficient memory structure," in *Proc. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, 1997.

[kondo08] H. Kondo, M. Nakajima, N. Masui, S. Otani, N. Okumura, Y. Takata, T. Nasu, H. Takata, T. Higuchi, M. Sakugawa, H. Fujiwara, K. Ishida, K. Ishimi, S. Kaneko, T. Itoh, M. Sato, O. Yamamoto, and K. Arimoto, "Design and Implementation of a Configurable Heterogeneous Multicore SoC With 9 CPUs and 2 Matrix Processors," *IEEE J. Solid-State Circuits*, vol. 43, no. 4, pp. 892-901, April 2008.

[kondo09] H. Kondo, O. Yamamoto, S. Otani, N. Sugai, and T. Shimizu, "Software architecture of a secure multimedia system using a multicore SoC and software virtualization," in IEEE Int. Conf. Consumer Electronics, Dig. Tech. Papers, pp. 1-2, Jan. 2009.

[koyama01] T. Koyama, K. Inoue, H. Hanaki, M. Yasue, and E. Iwata, "A 250-MHz single-chip multiprocessor for audio and video signal processing," *IEEE J. Solid-State Circuits*, vol. 36, pp. 1768–1774, Nov. 2001.

[kvm] "Kernel Virtual Machine", Available: http://www.linux-kvm.org/page/Main_Page

[lekatsas00] H. Lekatsas, J. Henkel, W. Wolf, "Code compression for low power embedded system design,"in *Proc. Design Automation Conference*, pp. 294 - 299, 2000.

[linley10] The Linley Group, "ARM's Digital Signal Controller," *Microprocessor Report*, December 2010.

[mentoreh] "Mentor Embedded Hypervisor," Mentor Graphics, Available: https://www.mentor.com/embedded-software/hypervisor/, 2013

[mips13] MIPS32 microAptiv UC Processor Core Family Software User's Manual, Revision 01.01, 2013.

[nishi00] N. Nishi, T. Inoue, M. Nomura, S. Matsushita, S. Torii, A. Shibayama, J. Sakai, T. Ohsawa, Y. Nakamura, S. Shimada, Y. Ito, M. Edahiro, M. Mizuno, K. Minami, O. Matsuo, H. Inoue, T.

Manabe, T. Yamazaki, Y. Nakazawa, Y. Hirota, Y. Yamada, N. Onoda, H. Kobinata, M. Ikeda, K. Kazama, A. Ono, T. Horiuchi, M. Motomura, M. Yamashina, and M. Fukuma, “A 1 GIPS 1Wsingle-chip tightly-coupled four-way multiprocessor with architecture support for multiple control flow execution,” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 418–419, Feb. 2000.

[noda07] H. Noda, M. Nakajima, K. Dosaka, K. Nakata, M. Higashida, O. Yamamoto, K. Mizumoto, T. Tanizaki, T. Gyohten, Y. Okuno, H. Kondo, Y. Shimazu, K. Arimoto, K. Saito, and T. Shimizu, “The design and implementation of the massively parallel processor based on the matrix architecture,” *IEEE Journal of Solid-State Circuits*, vol.42, No.1, pp. 183-192, January 2007.

[okamoto07] T. Okamoto, S. Miura, T. Boku, M. Sato, and D. Takahashi, “RI2N/UDP: High bandwidth and fault-tolerant network for a PC-cluster based on multi-link Ethernet,” in *The Workshop on Communication Architecture for Clusters (CAC) in IPDPS*, pp. 1–8, Apr. 2007.

[openfab] OpenFabrics, Alliance “OpenFabrics Enterprise Distribution (OFED),” Available: http://www.infinibandta.org/specs/

[otnai11f] S. Otani, H. Kondo, I. Nonomura, M. Uemura, Y. Hayakawa, T. Oshita, S. Kaneko, K. Asahina, K. Arimoto, S. Miura, T. Hanawa, T. Boku, and M. Sato, “An 80Gbps dependable communication SoC with PCI Express I/F and 8 CPUs,” in *IEEE International Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 266-267, February 2011.

[otani11n] S. Otani, H. Kondo, I. Nonomura, T. Hanawa, S. Miura, T. Boku, “Peach: A Multicore Communication System on Chip with PCI Express,” *IEEE Micro*, vol. 31, no. 6, pp.39-50, 2011.

[otani13] S. Otani, N. Ishikawa, H. Kondo, “RXv2 processor core for low-power microcontrollers,” in *Proc.Cool Chips XVI*, April 2013.

[pcie06] “PCI Express Base Specification, Rev. 2.0,” PCI-SIG, December 2006. Available: http://www.pcisig.com/

[pcie07j] “PCI Express External Cabling Specification, Rev. 1.0,” PCI-SIG, January 2007. Available: http://www.pcisig.com/specifications/pciexpress/pcie_cabling1.0

[pcie07a] “PCI Express Card Electromechanical (CEM) Specification, Rev.2.0,” PCI-SIG, April 2007.

[qlogic] “QLogic TrueScale™ InfiniBand, the Real Value,” Available: http://www.qlogic.com/

[renesas13] RX Family RXv2 Instruction Set Architecture User’s Manual: Software, Rev.1.00, 2013

[sugure04] Y. Sugure, S. Takeuchi, Y. Abe, H. Yamada, K. Hirayanagi, A. Tomita, K. Hagiwara, T. Kataoka, T. Yamazaki1 and T. Shimura, “A Very-Low-Latency Superscalar Microcontroller for Automotive, Industrial, and PC-Peripheral Applications”, *Cool Chips VII*, April 2004.

[strik00] M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and G. van Rootselaar, ”Heterogeneous multiprocessor for the management of real-time video and graphics streams,” *IEEE J. Solid-State Circuits*, vol. 35, pp. 1722–1731, Nov. 2000.

[tengine] “T-Engine Forum,” Available: http://www.t-engine.org/english/whatis.html

[toppers] “TOPPERS Project,” Available: http://www.toppers.jp/en/index.html

[vmwarevsphere] “VMware vSphere,” VMware, Available: http://www.vmware.com/products/vsphere/

[xenserver] “XenServer,” Citrix, Available: http://www.citrix.com/products/xenserver/overview.html

[xarm10] ARMv7-M Architecture Reference Manual, Issue D, ARM Limited, 2010.

[xen] The xen project. Available: http://www.xen.org/.

Publications

Major Papers

[1] S. Otani and H. Kondo, “RX v2: Renesas’s New-Generation MCU Processor,” *IEICE Transactions, Vol. E98-C, No. 7*, pp. 544-549, Jul. 2015.

[2] S. Otani, H. Kondo, I. Nonomura, T. Hanawa, S. Miura and T. Boku, “Peach: A Multicore Communication System on Chip with PCI Express,” *IEEE Micro, vol. 31, no. 6*, pp. 39-50, Nov.-Dec. 2011.

[3] H. Kondo, S. Otani, M. Nakajima, O. Yamamoto, N. Masui, N. Okumura, M. Sakugawa, M. Kitao, K. Ishimi, M. Sato, F. Fukuzawa, S. Imasu, N. Kinoshita, Y. Ota, K. Arimoto, and T. Shimizu, “Heterogeneous Multicore SoC With SiP for Secure Multimedia Applications,” *IEEE J. Solid-State Circuits, vol. 44, no. 8*, pp. 2251-2259, Aug. 2009.

[4] H. Kondo, M. Nakajima, N. Masui, S. Otani, N. Okumura, Y. Takata, T. Nasu, H. Takata, T. Higuchi, M. Sakugawa, H. Fujiwara, K. Ishida, K. Ishimi, S. Kaneko, T. Itoh, M. Sato, O. Yamamoto, and K. Arimoto, “Design and Implementation of a Configurable Heterogeneous Multicore SoC With 9 CPUs and 2 Matrix Processors,” *IEEE J. Solid-State Circuits, vol. 43, no. 4*, pp. 892-901, April 2008.

[5] S. Kaneko, H. Kondo, N. Masui, K. Ishimi, T. Itou, M. Satou, N. Okumura, Y. Takata, H. Takata, M. Sakugawa, T. Higuchi, S. Ohtani, K. Sakamoto, N. Ishikawa, M. Nakajima, S. Iwata, K. Hayase, S. Nakano, S. Nakazawa, K. Yamada and T. Shimizu, "A 600MHz Single-Chip Multiprocessor with 4.8GB/s Internal Shared Pipelined Bus and 512kB Internal Memory," *IEEE J. Solid-State Circuits, vol. 39*, no. 1, pp. 184-193, Jan. 2004.

Major Conferences

[1] S. Otani, N. Ishikawa, H. Kondo, “RXv2 processor core for low-power microcontrollers,” in *Proc. Cool Chips XVI*, April 2013.

[2] S. Otani, H. Kondo, I. Nonomura, Ikeya, A., K. Asahina, K. Arimoto, S. Miura, T. Hanawa, T. Boku, and M. Sato, "An 80 Gbps dependable multicore communication SoC with PCI express I/F and intelligent interrupt controller," in *IEEE int. Cool Chips XIV*, pp. 1-3, 20-22, April 2011.

[3] S. Otani, H. Kondo, I. Nonomura, A. Ikeya, M. Uemura, Y. Hayakawa, T. Oshita, S. Kaneko, K. Asahina, K. Arimoto, S. Miura, T. Hanawa, T. Boku, and M. Sato, “An 80Gb/s Dependable Communication SoC with PCI Express I/F and 8 CPUs,” in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 266-267, Feb. 2011.

[4] H. Kondo, O. Yamamoto, S. Otani, N. Sugai, and T. Shimizu, “Software architecture of a secure multimedia system using a multicore SoC and software virtualization,” in *IEEE Int. Conf. Consumer Electronics, Dig. Tech. Papers*, pp. 1-2, Jan. 2009.

[5] H. Kondo, M. Nakajima, S. Otani, O. Yamamoto, N. Masui, N. Okumura, M. Sakugawa, M. Kitao, K. Ishimi, M. Sato, F. Fukuzawa, K. Inaoka, Y. Saito, K. Arimoto and T. Shimizu, "Heterogeneous multicore SoC for secure multimedia applications," in *Custom Integrated Circuits Conf. Dig. Tech. Papers*, pp. 675-678, Sept. 2008.

[6] S. Kaneko, K. Sawai, N. Masui, K. Ishimi, T. Itou, M. Satou, H. Kondo, N. Okumura, Y. Takata, H. Takata, M. Sakugawa, T. Higuchi, S. Ohtani, K. Sakamoto, N. Ishikawa, M. Nakajima, S. Iwata, K. Hayase, S. Nakano, S. Nakazawa, O. Tomisawa, and T. Shimizu, "A 600 MHz single-chip multiprocessor with 4.8 GB/s internal shared pipelined bus and 512 kB internal memory," in *IEEE int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 254-255, Feb. 2003.