Doctoral Dissertation

A Study on Automatic Generation of Embedded Processors and Real-Time OS Adapted to Applications

Tetsuo Miyauchi

Supervisor: Professor Kiyofumi Tanaka

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2021

Abstract

In recent years, the use of embedded microprocessors has been increasing with prevailing IoT (Internet of Things) and RTOSs are commonly used to develop a real-time system effectively. By using an RTOS, a complicated procedure can be divided into tasks with real-time scheduling based on the preemptive and priority-based task scheduling, and a real-time application can be developed easily with using RTOS functions such as task management, task dependent synchronization, synchronization and communication. While there are advantages for using an RTOS, an RTOS itself consumes additional memory, computational resources and power. Based on these points, our research objectives are as follows: (1) Build an application adaptive processor core. (2) Remove unused codes in RTOS kernel, while leaving necessary functions such as checking possible errors. (3) Implement hardware RTOS to reduce the amount of software resources and execution time. (4) Remove unused codes in a hardware RTOS as well as a software-only RTOS. (5) Build automatic development environment with which we can perform the items above.

In order to achieve the objectives, we adopted MIPS32 architecture for a processor core and illustrated the method for analyzing an application program and generating the application adaptive processor core circuit. In addition, we implemented two- to eight-core multi core processor on an FPGA and showed eight-core processor can be implemented on a relatively small FPGA with application adaptive processor cores.

Regarding RTOS, we proposed a framework to generate application adapted hardware RTOS and software-only RTOS. For the specification of an RTOS, we adopted μITRON4.0 for the research as it is widely used and its specification is open in public. We propose the methods, “Removing Unnecessary Codes Caused by Fixed Attributes” and “Removing Unnecessary Codes Caused by the Way of Calling”, for generating an application adaptive RTOS kernel. For the former method, as each system call is specified with attributes through parameters in a configuration file, functions which are not specified in the configuration file can be deleted from the RTOS kernel. For the latter method, error codes for system calls are defined in the RTOS specification whereas codes for checking errors which never occur in the application program remain in some cases. Since those codes are redundant when an application program is fixed, it is shown that how unnecessary error checking can be removed. In addition, we explained the structure of the hardware RTOS, which consists of RTOS Hardware Wrapper and RTOS Hardware Core. We propose an environment to generate an application adaptive processor core and a hardware/software-only RTOS kernel in a fully automatic manner.

For the evaluation of the effect of the proposal, we applied the proposed methods to several application programs and measured FPGA resources, RTOS kernel execution time and the size of the software parts. As a result, it can be seen that the hardware resources and the size of a software part of an RTOS kernel are reduced, and that the system call execution time is improved.

Keyword: processor, MIPS, RTOS, μITRON, configuration, system call, FPGA, adaptation

Acknowledgments

I am most grateful to all people who have supported me to submit this dissertation, directly and indirectly. Especially, I would like to express my heartfelt gratitude to my supervisor, Professor Kiyofumi Tanaka of Japan Advanced Institute of Science and Technology (JAIST) for his patient guidance. My study began at the discussion with Professor Tanaka for my project paper's theme in my master course. Without his orientation, I could not complete the dissertation. I was always given useful comments in detail when I submitted papers until just before the deadline. Attending international conferences with him was also good experiences and I obtained lots of valuable advices for the presentation. I would also like to appreciate Professor Mineo Kaneko of JAIST, advisor for minor research project. He enlightened me as to suggestions for the dissertation especially for related work as well as my minor research project. I appreciate Professor Yasushi Inoguchi of JAIST, second supervisor, who gave me helpful comments, which increases the value of the dissertation. Furthermore, I appreciate Professor Yasuo Tan of JAIST and Professor Hironori Nakajo of Tokyo University of Agriculture and Technology for their comments on the dissertation. I learned a lot from their comments. In addition, I would like to thank all of professors in JAIST whose lectures I attended. All lectures without exception were very interesting and helpful for my background of continuous research. I also thank all members of Tanaka laboratory in Tokyo satellite and I feel it was precious experiences to have participated in a summer workshop and joint seminar with members in the Ishikawa campus. Their studies stimulated me to keep studying. Finally, I would like to express my appreciation for my family, who has encouraged me and supported my daily life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In recent years, the use of embedded microprocessors has been increasing with prevailing IoT (Internet of Things). There are too many home appliances with embedded micro processors to list. For example, a smart phone, a television set, a pot, a cooker, a smart speaker, an air conditioner, a printer, a copy machine, a washing machine, a refrigerator, a microwave oven and so on have embedded processors. In an automobile, more than a couple dozen microprocessors are used to control a vehicle. From the perspective of hardware, in the embedded system, when the size of a device and substrate increases, the cost of the appliance also increases so that reducing the size is essential for a product.

Since a processor embedded in an appliance works only for a dedicated system, designing an application specific processor has been studied. ASIP in the literature [1],[2] is one of the examples of a study for such a processor. In the literature [3], to reduce the size of memory, changing the CPU bit-width was studied.

From the perspective of software, RTOS (Real Time Operating System) is commonly used for the embedded system. One of the commonly used RTOS specification is μITRON in Ref.[4], which has task management functions, task dependent synchronization functions, synchronization and communication functions. With an RTOS, it is possible to easily abstract hardware, use synchronization/communication and benefit from real-time scheduling based on the preemptive and priority-based task scheduling. In addition, a complicated processing can be divided into tasks.

Here, we illustrate a simple example in Fig.1.1 to show the advantages of using task division. There are three tasks, Task A, Task B and Task C. The priority of Task A is the highest, Task B is the middle and Task C is the lowest. We consider the following case. Firstly, Task C works and wakes up Task A. As the priority of Task A is the highest, Task A is executed just immediately after it is woken up. Task A wakes up Task B, but Task B does not work immediately since the priority of Task A is higher than Task B. In execution of Task A, Task A waits for a semaphore due to no resources of the semaphore. As the Task B is ready to execute, Task B is executed and returns a semaphore resource. In this case, Task A resumes executing as Task A is the highest priority. When a procedure of Task A is finished, Task A goes to sleep, then Task B resumes executing. In terminating a procedure of Task B, Task B goes to sleep and Task C executes again. An example of a task code is shown below.

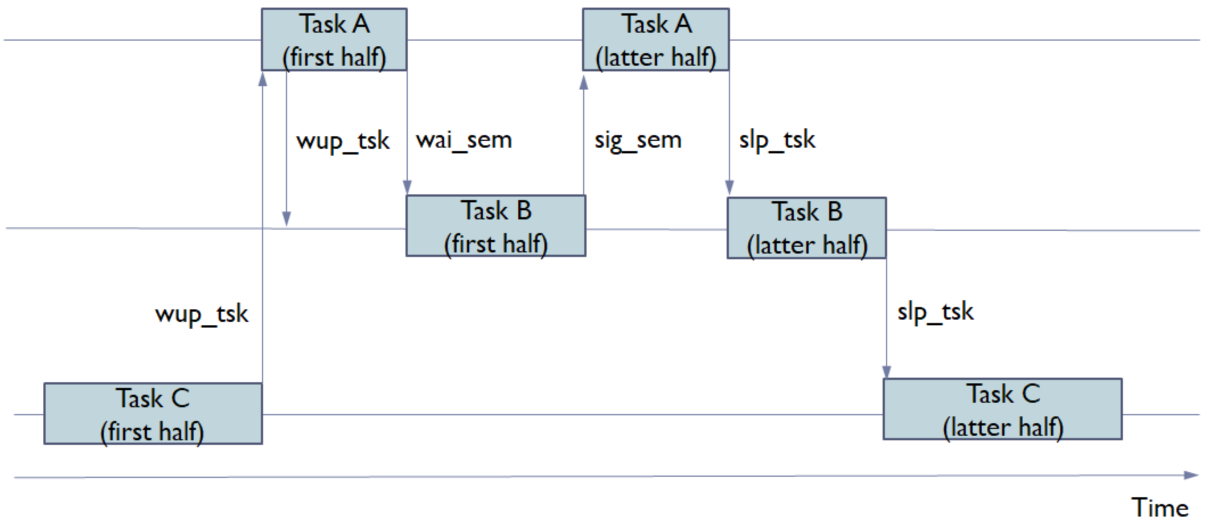


Figure 1.1: Task sequence.

Example of a task set (task A) (excerpt).

```
/* priority: high */
void taska()
{
    ER ercd;

    while(1) {
        /* wait for wake up */
        ercd = slp_tsk();

        /* wake up task B */
        ercd = wup_tsk(TASKB);

        /* Process for task A */
        /* ... */
        /* wait for semaphore */
        ercd = wai_sem(SEM_ID1) ;

        /* Post process */
        /* ... */

    }
}
```

Example of a task set (task B)(excerpt).

```
/* priority : middle */
```

```
void taskb()
{
    ER ercd;

    while(1) {
        /* wait for wake up */
        ercd = slp_tsk();

        /* Process for task B */
        /* ... */
        ercd = sig_sem(SEM_ID1) ;

        /* Post process */
        /* ... */

    }
}
```

Example of a task set (task C)(excerpt).

```
/* priority : low */
void taskc()
{
    ER ercd;

    while(1) {
        /* Process for task C */
        /* ... */

        /* wake up task A */
        ercd = wup_tsk(TASKA);

        /* Post process */
        /* ... */
    }
}
```

`taska()`,`taskb()`, and `taskc()` are the source code of Task A, Task B, and Task C, respectively. Initially all tasks are started, but Task A and Task B go to the sleep state, so that Task C starts to execute as Fig.1.1. Each task invokes system calls to behave as described above. The source code reveals that just invoking system calls can realize the task transition. An example of implementing the same procedure without using RTOS is as follows.

Example of a procedure w/o RTOS (main).

```
int firstflag_a = TRUE; /* TASK A */
int firstflag_b = TRUE; /* TASK B */
int firstflag_c = TRUE; /* TASK C */
int state_taska = NOTREADY;
int state_taskb = NOTREADY;
int state_taskc = READY;

int sem1 = 0; /* semaphore */

int main()
{

    while(1) {
        if (state_taska == READY) {
            taska();
        } else if (state_taskb == READY) {
            if (state_taska == WAIT) {
                taska();
            }
            taskb();
        } else if (state_taskc == READY) {
            if (state_taska == WAIT) {
                taska();
            }
            if (state_taskb == WAIT) {
                taskb();
            }
            taskc();
        } else {
            /* there is no ready task */
            continue;
        }
    }
    return(0);
}
```

Example of a procedure w/o RTOS (Task A).

```
void taska()
{
    if (firstflag_a == TRUE) {
        /* first half */
        firstflag_a = FALSE;

        if (sem1 == 0) {
            /* semaphore is not acquired */
            /* return to the main loop */
            /* task A state is WAIT */
            state_taska = WAIT;
            return;
        } else {
            /* semaphore is acquired */
            sem1 = 0;
            /* do a job for Task A */

            /* finish task A */
            /* As latter half of task A is finished, */
            /* do again from the first half */
            firstflag_a = TRUE;
            state_taska = NOTREADY;
            return;
        }
    } else {
        /* latter half */
        if (sem1 == 0) {
            /* semaphore is not acquired */
            /* return to the main loop */

            /* task A state is WAIT */
            state_taska = WAIT;
            return;
        } else {
            /* semaphore is acquired */
            sem1 = 0;
            /* do a job for Task A */

            /* finish task A */
            /* As latter half of task A is finished, */
            /* do again from the first half */
            firstflag_a = TRUE;
            state_taska = NOTREADY;
            return;
        }
    }
}
```

Example of a procedure w/o RTOS (Task B).

```
void taskb()
{
    if (firstflag_b == TRUE) {
        /* first half */
        firstflag_b = FALSE;
        /* Return semaphore to execute task A */
        sem1 = 1;
        return;
    } else {
        /* latter half */
        firstflag_b = TRUE;
        /* task B is finished */
        state_taskb = NOTREADY;
        return;
    }
}
```

Example of a procedure w/o RTOS (Task C).

```
void taskc()
{
    if (firstflag_c == TRUE) {
        /* first half */
        firstflag_c = FALSE;
        /* wake up task A */
        state_taska = READY;
        /* wake up task B */
        state_taskb = READY;

        return;
    } else {
        /* latter half */
        firstflag_c = TRUE;

        return;
    }
}
```

`taska()`,`taskb()`, and `taskc()` are functions working in a similar way. As can be seen from the code, it needs global variables to control the task behavior, such as `firstflag_a`, `firstflag_b`, `firstflag_c`, `state_taska`, `state_taskb` and `state_taskc`. As each function executes two parts of the function, `firstflag_a`, `firstflag_b` and `firstflag_c` are used for controlling the task transition. `state_taska`, `state_taskb` and `state_taskc` store the current task state. `sem1` is used to store the count of the semaphore. In addition `main` function is necessary to invoke each procedure. Generally speaking, it is recommended that the module coupling should be low in designing software, whereas the example without RTOS needs several global variables to control and communicate functions, so that high module coupling is inevitable. In an actual system, much more global variables are necessary so that the source code will become more complicated, which makes readability and maintainability worse.

As explained above, using RTOS has advantages to developing embedded systems. In the literature [5], the advantages and disadvantages of using RTOSs for small microcontroller system development are discussed. The literature illustrates that software productivity can be improved with using RTOSs in system development. However, when RTOSs are used for small microcontrollers, the literature also described disadvantages that an RTOS itself consumes additional memory, computational resources and power. As an example of a small microcontroller, the flash ROM size of a microcontroller family in Ref.[6], having various applications, ranges from 1Kbyte to 512Kbytes, which cannot be ignored for the memory consumption of an RTOS kernel.

Since using RTOS has benefit in designing embedded applications as mentioned above, our motivation for the research comes from the requirement for mitigating overhead of an RTOS. An RTOS kernel following μITRON4.0 specification in Ref.[7] is usually provided as a library format, so that only specific system calls which are actually used in an application program are linked with an application program, which means a program code for unused system calls does not occupy the memory space. However, the unit of the selection of codes is a whole system call, which is too rough for the limited memory resource. In order to reduce the overhead of software part, hardware RTOS is effective. Since hardware resource is related to cost of the production, reducing hardware resources is also required. Based on the points above, when an application program is fixed, our research objectives are as follows.

- Build an application adaptive processor core.
- Remove unused codes in RTOS kernel, while leaving necessary functions such as checking possible errors.
- Implement hardware RTOS to reduce the amount of software resources and execution time.
- Remove unused codes in a hardware RTOS as well as a software-only RTOS.
- Build automatic development environment with which we can perform the items above.

In this dissertation, in order to evaluate the proposed method, we implement a product to which adaptation is applied on an FPGA device. While FPGA devices including microprocessor hard cores are prevailing in recent years, since the target of this study includes small/low-cost IoT devices, we consider low-cost FPGA devices without microprocessor hard cores as the subject of this dissertation. Here, we call it “adaptation” to eliminate unnecessary functions for an application. A product to which adaptation is applied is called an “adaptive” or “adapted” product. The following Chapters explain our proposal to achieve the objectives above in detail.

1.2 Contribution

Since the process described in this dissertation can be applied for generating an application adaptive processor core and RTOS, the main contribution of this dissertation is summarized as follows.

- Propose a method for generating an application adaptive processor core automatically.
- Illustrate how we configure a fine-grained RTOS kernel.
- Explain the way to create hardware RTOS and configure application adaptive hardware RTOS automatically.
- Show the automatic development environment.

Since the number of IoT devices including processors is increasing extremely, a development environment for producing low cost and effective devices in a short period of time is desired. Recently, advances of technology make it possible to implement functions which a target system needs in a device, which is called SoC (System on Chip). While functions embedded in a device depend on a target system, a processor core and RTOS are commonly used technology so that it can be said that this dissertation contributes to revealing the method for creating application adaptive devices.

1.3 Structure of Dissertation

This dissertation is organized as follows. Related work regarding adaptation of a processor core and OS kernel, software overhead mitigation, and hardware implementation for RTOS functions is reviewed in Chapter 2. We explain about construction of automatic development environment for adaptive processor core in Chapter 3. In this chapter, a processor structure for our target is explained and how we analyze an application program and construct an adapted processor core for the application is illustrated. In Chapter 4, we propose a framework for building fine-grained RTOS including software adaptation and hardware adaptation. For software adaptation, we describe a method for building an application adapted system call. For hardware adaptation, the way of implementing RTOS functions as hardware and the method for building an application adapted hardware are proposed. Chapter 5 shows the effect of RTOS configuration, which includes an evaluation for hardware resource usage, software size for system calls, and system call execution time. Finally, in Chapter 6, we summarize this dissertation and discuss about future work.

Chapter 2

Related Work

2.1 Related Work

In this chapter, we review related works for this dissertation. As there are several approaches to customize microprocessors according to a requirement, we oversee studies for processor core adaptation. Then the overviews of RTOS related adaptation are summarized. First, the way of RTOS kernel adaptation in software is described. Next, several researches regarding that a part of functions of an RTOS is implemented in hardware are shown as a method for software overhead mitigation. Especially, a hardware scheduler has been studied since scheduler is a core component of an RTOS, so that we review previous studies for a hardware scheduler. Finally, we explain studies for implementing a whole RTOS system call function in hardware.

2.1.1 Processor Core Adaptation

For a processor core embedded in an FPGA, there are two ways to implement one: hard macro processors and soft macro processors. While hard-macro processors are built-in along with FPGA resources, soft processors are constructed with programmable resources in an FPGA. Cortex-A9 in Zynq-7000 in Ref.[8] is an example of a hard-macro processor. Hard processors have an advantage in performance and size while they are not flexible for changing structures. On the other hand, MicroBlaze of Xilinx in Ref.[9] and NiosII of Intel in Ref.[10] are examples of soft processors provided as software macro IP in an FPGA. Soft processors are configurable so that they can be customized according to running applications and can be extended with acceleration circuits if necessary.

To create an application specific processor, there are several approaches as follows. ASIP (Application-domain Specific Instruction-set Processor) is a technology for building a processor which is optimized for an application program. ASIP methodologies are surveyed in the literature [1],[2]. Table 2.1 summarizes a comparison of features between ASIP and the proposed method. For generating an application specific processor, ASIP needs to analyze an application program manually and create a tool set including a compiler for the processor. For creating the tool set, ASIP provides a tool called ASIP Meister, which has an interface to input the register length, the number of registers and the number of pipeline stages. Xtensa of Tensilica in the literature [11] is a commercial processor which can be customized for target applications. This processor can be optimized for an

Table 2.1: Comparison of ASIP and the proposed method.

Feature	ASIP[2]	Proposed Method
Program analysis	Manual	Automatic
Generate a processor	Manual translation to ADL[1] ADL is synthesized to HDL	Generate by a tool
Instructions	Add dedicated instructions	Use existing instructions
Registers	Add special purpose registers	Not add new registers
Development tool set	Need to extend an existing tool set for new instructions	Able to use an existing tool set

embedded application by sizing and selecting features and adding new instructions. The literature [3] proposed to change the CPU bit-width to reduce the size of memory in a processor. In these approaches, the processor needs to be tuned manually. As another approach, selecting a subset of instructions from an existing instruction set is introduced in the literature [12],[13]. In this approach, existing software development tools including compilers can be applied. The techniques for selecting instructions show that the selection according to application programs simplifies the processor's microarchitecture and reduces the amount of hardware resources and it is illustrated that about 50% of instructions among the full instruction set are actually used for several applications, and that the method is effective in reducing area and improving clock speed in the evaluation result. In their study, the analysis of program codes does not take into account dependency between instructions which determines necessity of forwarding paths. (That is, forwarding logic is always equipped even when it is unnecessary for the program.)

2.1.2 RTOS Kernel Adaptation

While an RTOS kernel software is usually provided as a library format, that is not enough for adapting to an application program. The configuration of an RTOS kernel adapted to an application system has been studied. In the literature [14], techniques for automatically reducing the memory footprint of general-purpose operating systems on embedded platforms are described. In this literature, hand-written assembly codes in the kernel are analyzed with a decompilation technique, and unused codes and duplicated codes are eliminated. However, as the target operating system of this literature is Linux, the meaning of unused codes is different from ours. In order to build an application specific RTOS kernel, OSEK in Ref.[15], which is an RTOS kernel commonly used in the automotive industry, defines OIL (OSEK Implementation Language) for description of application specific objects. This description is used as system configuration information, and a configurator generates a tailored RTOS which consists only of application specific objects and actually-used system calls. The literature [16] describes an example of OSEK-based RTOS, the main objective of which is to verify a generated RTOS, where configuration information and application codes are analyzed and an OS-application interaction graph is extracted for verification. Table 2.2 shows comparison between OSEK-based adaptation and our proposed method in terms of the level of RTOS kernel adaptation.

[1]Architecture Description Language

Table 2.2: Comparison between OSEK-based configuration and the proposed method.

Feature	OSEK-based[15][16]	Proposed Method
RTOS adaptation	Object/System call level	Code-fragments (Fine-grained) level

2.1.3 Software Overhead Mitigation

In order to mitigate software overhead in terms of resources and latency when RTOS is used, implementing some of the functions as hardware has been studied. SoCLC (SoC Lock Cache) hardware mechanism to improve the performance of lock latency is proposed in the literature [17],[18] and its framework for designing is explained in the literature [19]. A modular microkernel architecture in hardware is demonstrated in the literature [20]. In the literature [21], a configurable hardware scheduler architecture is presented. This scheduler provides three scheduling disciplines: priority-based, rate monotonic and earliest deadline first. This shows the advantage of modularity and the improvement of the performance. In the literature [22], three scheduler models are implemented: (i) SoRTS (Software Real-Time Scheduler), (ii) Co-SoRTS (Co-processor Software Real-Time Scheduler), and (iii) HaRTS (Hardware Real-Time Scheduler). It is concluded that Co-SoRTS and HaRTS present the best results for hard real-time applications, while SoRTS is suitable for soft real-time systems. The literature [23] and [24] show an ITRON based RTOS extended to support multithread processing for Responsive Multithreaded (RMT) Processor. As this implementation is dedicated to an RMT processor, the approach is different to a general-purpose processor.

2.1.4 Hardware Scheduler

A hardware scheduler with a new task-queue architecture to support various scheduling algorithms such as time sliced priority scheduling, Earliest Deadline First, and Least Slack Time is described in the literature [25]. RT-SHADOWS in the literature [26],[27] is a hardware scheduler and APIs to provide hardware multi-thread support, which is a subset of the task management APIs. In the literature [28], with SOPC (System On Programmable Chip), an architecture that co-schedules hardware and software with RTOS was discussed. The literature [29] and [30] proposed an interrupt scheduler called REMON (Real-Time Embedded Monitor). The scheduler is implemented in hardware to improve performance in executing switching of the ISR (Interrupt Service Routine). While this proposal implemented the scheduler in hardware, other functions of RTOS were not implemented in hardware. In the literature [31], earliest deadline first (EDF) scheduler is implemented in hardware. The literatures [32] and [33] show the method and evaluation result of implementing the scheduler function of TinyOS, an OS for sensor nodes. All the hardware implementations mentioned above are only for performance improvement and do not take adaptability to a target application into account.

2.1.5 Hardware RTOS

There are several studies for implementing a whole RTOS system call function in hardware. Silicon TRON in the literature [34],[35],[36],[37] provides basic functionalities of

μITRON in hardware as a peripheral chip and the performance was evaluated in the literature [38]. The literature [39] explained that a part of μITRON functions, such as task management, task dependent synchronization functions, semaphore, eventflag, and interrupt management were implemented in hardware. The literature [40] shows an idea for extracting RTOS kernel functions from UML description of an application program, whereas the detailed method was not described. In the literature [41], a real-time kernel coprocessor is implemented in an ASIC, which is called RTU. It has eight priorities and RTOS functions such as semaphore, eventflag, watchdog and priority preemptive scheduling. A real-time multithreaded operating system kernel, hthreads, is presented in the literature [42]. It has a shared memory programming model similar to POSIX Threads. In the system on CPU/FPGA chips, hardware threads and software threads can exist and they are scheduled by a hardware scheduler component, which performs first-in-first-out, round robin and priority-based scheduling algorithms. In the literature [43], general-purpose RTOS functions with API interfaces and a dedicated CISC processor are implemented in an FPGA. ARTESSO [44],[45],[46] is a hardware RTOS, which provides more than 30 system calls. The specific feature of the RTOS hardware is a Virtual Queue, which is a queue structure with a tournament circuit to select an element in the queue. They enhanced the architecture to a multi-core processor in the literature [47] and studied to implement on ARM-based SOC in the literature [48],[49]. However, these RTOSs do not have a function to adapt to an application program automatically.

2.1.6 Adaptation for Hardware RTOS Functions

Simple and Effective hardware based Real-Time Operating System (SEOS) in the literature [50] provides adaptability for hardware RTOS. SEOS adaptation consists of hardware and software processes. However, these processes need to be applied manually. In the literature [51], OSEK-based RTOS hardware, called OSEK-V processor, is implemented with an application system after analysis of the application program, but it is not flexible to updates of the application. In the literature [52], a method of generating full hardware implementation where tasks as well as RTOS are implemented in hardware is described. To synthesize tasks for hardware, there is some restriction to tasks (e.g., no mutual exclusion). In this literature, adaptation of RTOS is not described. It is described that error checking in act_tsk takes 21 cycles while only 1 cycle is needed with our method and also unused hardware resources can be deleted in our method. Table 2.3 shows comparison in terms of RTOS hardware adaptation.

While studies for implementing RTOS functions in hardware mentioned above have been conducted for several years, we have been studying to reduce runtime and resource overhead by adapting RTOS kernel functions and processor functions to an application program. We confirmed an effect of implementing a primitive RTOS kernel operation as hardware in the literature [53]. After that, we proposed a hardware RTOS implementation in a system-call level in the literature [54]. However, automatic adaptation environment is not shown in the literature. In this dissertation, we explain our method for developing an application-specific system with RTOS in detail.

Table 2.3: Comparison in terms of RTOS hardware adaptation.

Feature	SEOS[50]	OSEK-V[51]	Ref.[52]	Proposed Method
Specification	SEOS	OSEK	TOPPERS/APS3	μITRON4.0
Processor	NIOS-II	Tailoring based on Rocket RISC-V	No CPU	MIPS32
Platform	Altera Cyclone-II	Xilinx Zynq-7020	Xilinx Artix-7	Xilinx Artix-7
Software Part	Simple API for interfacing purpose	No software part	No software part	Simple API for interfacing purpose
Hardware Part	Major OS functionalities	All of OSEK system	All of RTOS system	Major OS functionalities
RTOS Adaptation	Manual	Automatic	No adaptation	Automatic

Chapter 3

Construction of Automatic Development Environment for Adaptive Processor Core

3.1 Structure of Processor Core

The techniques presented in Sections 3.1 to 3.5 are ones proposed in the author's literature [55],[56],[57]. In this study, we adopted MIPS architecture for a processor core since the architecture is well-known and is commonly used in the market. The structure of a processor core is 5-stage pipeline and the instruction set architecture is MIPS32 in Ref.[58]. The feature of the processor core we implemented is explained below.

Instruction set

We implemented all of the instructions on the core instruction set in the literature [59] except for load link instruction (`ll`) and store conditional (`sc`), which are instructions for a multi-processor core. Additionally, as the compiler (GCC) outputs several instructions which are not on the list of the literature [59], the instructions, `xor`, `xori`, `sra`, `sllv`, `srav`, `srlv`, `srav`, `srlv`, `bgez`, `bltz`, `blez`, `bgtz`, `jalr`, `lb`, and `lh`, are also implemented. To execute multiplication and division, we implemented the instructions related to these operations, `mult`, `multu`, `div`, `divu`, `mfhi` and `mflo`. `mfc0` is also implemented for a processor control. All instructions we implemented are shown in the table 3.1.

Table 3.1: Instruction set.

Category	Instruction	Description
Arithmetic operation	`add`	Add
	`addu`	Add unsigned
	`addi`	Add immediate
	`addiu`	Add immediate unsigned
	`and`	And
	`andi`	And immediate
	`sub`	Subtract
	`subu`	Subtract unsigned

continue to the next page

Category	Instruction	Description
	`lui`	Load upper immediate
Logical operation	`nor`	Nor
	`or`	Or
	`ori`	Or immediate
	`xor`	Exclusive or
	`xori`	Exclusive or immediate
Comparison	`slt`	Set on less than
	`sltu`	Set on less than unsigned
	`slti`	Set on less than immediate
	`sltiu`	Set on less than immediate unsigned
Shift	`sll`	Shift left logical
	`sra`	Shift right arithmetic
	`srl`	Shift right logical
	`sllv`	Shift left logical variable
	`srav`	Shift right arithmetic variable
	`srlv`	Shift right logical variable
Branch	`beq`	Branch on equal
	`bne`	Branch on not equal
	`bgez`	Branch on greater than or equal to zero
	`bltz`	Branch on less than zero
	`blez`	Branch on less than or equal to zero
	`bgtz`	Branch on greater than zero
Jump	`j`	Jump
	`jal`	Jump and link
	`jalr`	Jump and link register
	`jr`	Jump register
Load	`lb`	Load byte
	`lbu`	Load byte unsigned
	`lh`	Load halfword
	`lhu`	Load halfword unsigned
	`lw`	Load word
Store	`sb`	Store byte
	`sh`	Store halfword
	`sw`	Store word
Multiplication	`mult`	Multiply
	`multu`	Multiply unsigned
Division	`div`	Divide
	`divu`	Divide unsigned
Register transfer	`mfhi`	Move from high
	`mflo`	Move from low
Control	`mfc0`	Move from coprocessor

5-stage pipeline

The processor core we implemented consists of the five stage pipeline, IF (Instruction Fetch), ID (Instruction Decode), EX (Execution), MEM (Memory Access), and WB (Write Back to a register) stages. IF is the stage for reading an instruction from the instruction memory according to PC (Program Counter). With our implementation, since a program for the processor is stored in a block memory on an FPGA in advance, the instruction indicated by PC is read in synchronization with the clock. ID is a stage for decoding the instruction and setting necessary controls to execute the instruction correctly. With this implementation, a decision for whether a branch is taken or not is made in the ID stage. EX is a stage for working ALU (Arithmetic Logic Unit), the unit for arithmetical or logical operations, and the unit for multiplication or division. MEM is a stage for reading data from or writing to the data memory according to the address calculated in the previous stage. WB is a stage for writing the calculated result back to a register.

Branch decision

When a conditional branch instruction is executed, the decision of whether the control proceeds to the following instruction (not-taken) or branches (taken) is made in ID stage.

Delay Slot

A conditional/unconditional branch instruction has one delay slot, so that the following instruction of a branch instruction is always executed regardless of the branch decision. We do not need to flush or stall an instruction in the delay slot. When a jump and link instruction (jal, jalr) is executed, the return address for the instruction is the address of PC+8.

Memory Map

The processor core we developed has 32Kbytes program area which is implemented with a block memory configured as a single port ROM in the FPGA. Read-only data such as a variable with constant expression is also stored in a block memory in the FPGA and can be accessed with the addresses from 0x8000 to 0xbfff (16Kbytes). For variable data, RAM area which can be accessed with the addresses from 0xc000 to 0xefff (12Kbytes) is implemented in the logic block in the FPGA. I/O area such as the address of UART control is memory mapped to the addresses from 0xf000 to 0xffff. RTOS hardware can be accessed with the addresses from 0xf0000000 to 0xffffffff, which is explained in Section 4.1 in detail. This memory map is summarized in the Fig.3.1.

Forwarding Unit

When an instruction uses the results of the former instructions, the pipeline processing does not have to be stalled as long as the results can be obtained via the pipeline registers even if it has not been written back to general-purpose registers. The forwarding unit detects these cases.

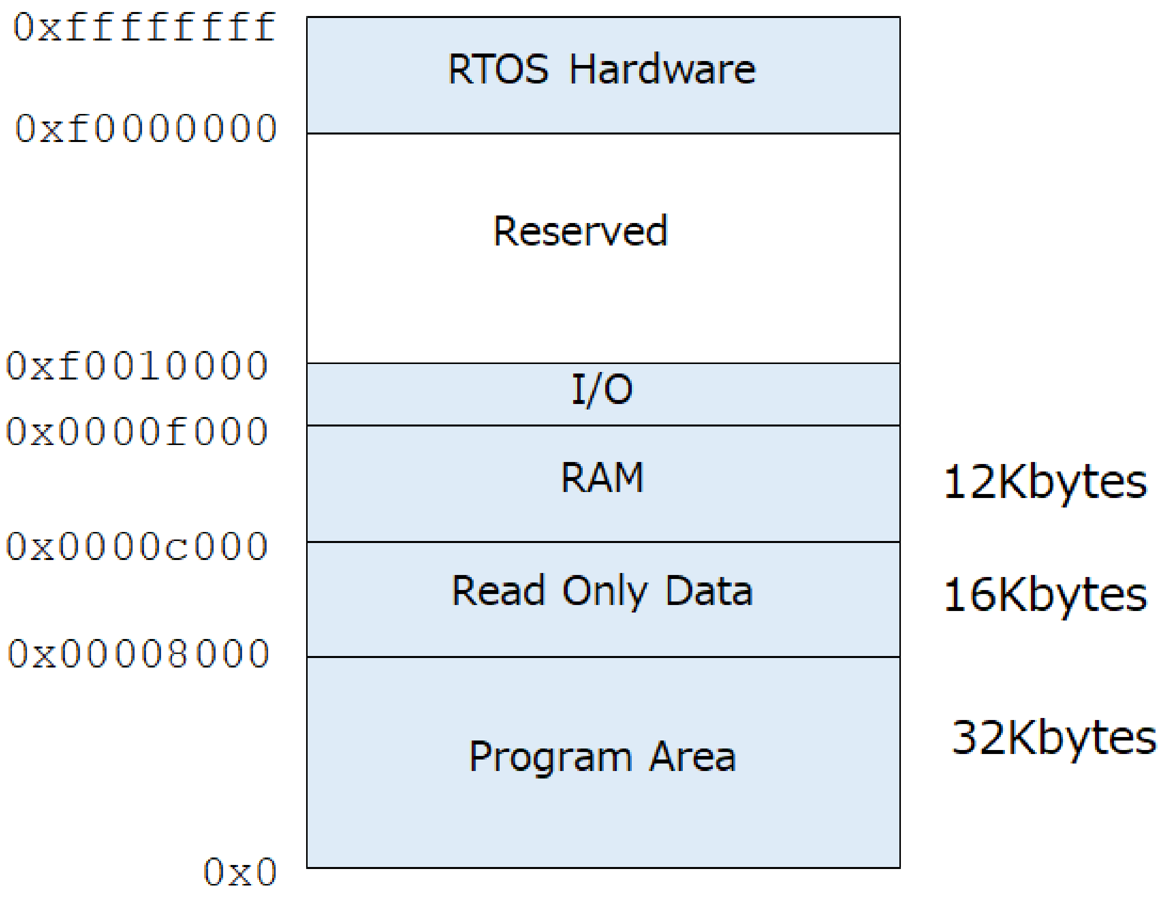


Figure 3.1: Memory map.

Detection of pipeline stall

When an instruction uses the results of the former instructions but the forwarding unit cannot supply the results immediately, the pipeline has to be stalled. The pipeline-stall-detection unit decides and controls the pipeline stall.

Clock Generation

We implemented the hardware circuit to an FPGA of Xilinx and used a clock generation IP on the FPGA to convert the clock rate. The frequency of the clock source of the FPGA is 100MHz and we used the clock generation IP to acquire a proper clock rate to execute our target program.

UART Interface

In order to output data from the processor core, we implemented UART interface through the IO pin of an FPGA board and the data can be acquired with a terminal soft on PC.

7-Segment display and LEDs

On the FPGA board we used for the evaluation, Digilent Basys3 in Ref.[60], there are 4-digit 7-segment display and 16 user LEDs. These I/O addresses are mapped to memory addresses to access these 7-segment display and LEDs, so that a write access to a designated address enables an application program to turn the 7-segment display or LEDs on.

3.2 Analyzing Application Program

In order to generate an application adaptive processor core, we proceed to the following steps. (Fig.3.2)

(Step 1) An object file for an application program is built. (Section 3.2.1)

(Step 2) An object file for the application program is analyzed. (Section 3.2.2)

(Step 3) Used instructions are extracted. (Section 3.2.3)

(Step 4) The object code is analyzed and the dependency patterns are searched. (Section 3.2.4)

(Step 5) Processor resources with used instructions and dependency patterns are selected. (Section 3.3)

These steps are also applied to our environment illustrated in Section 4.5.

3.2.1 Building Application Program

Source codes of an application program which are written in C language or MIPS assembly language are input, and C compiler is invoked to create an object file in executable format. In our implementation, GCC compiler for MIPS is used.

3.2.2 Analyzing Object Code

After an executable object file is generated, an assembly code is extracted from the object file. As we compile the source code with GCC compiler, the objdump command in GCC compiler utility is available for the disassembling.

3.2.3 Extracting Instructions

From the result of disassembling, actually used instructions are extracted and resources of a processor core to be used by the application program are determined. In disassembling the code which has been generated on the first step, the result such as Fig.3.3 is output and the instruction field is extracted to examine what instruction is used. We created a list of instructions and resources in advance and the list shows which resources are actually used for each instruction in an application program.

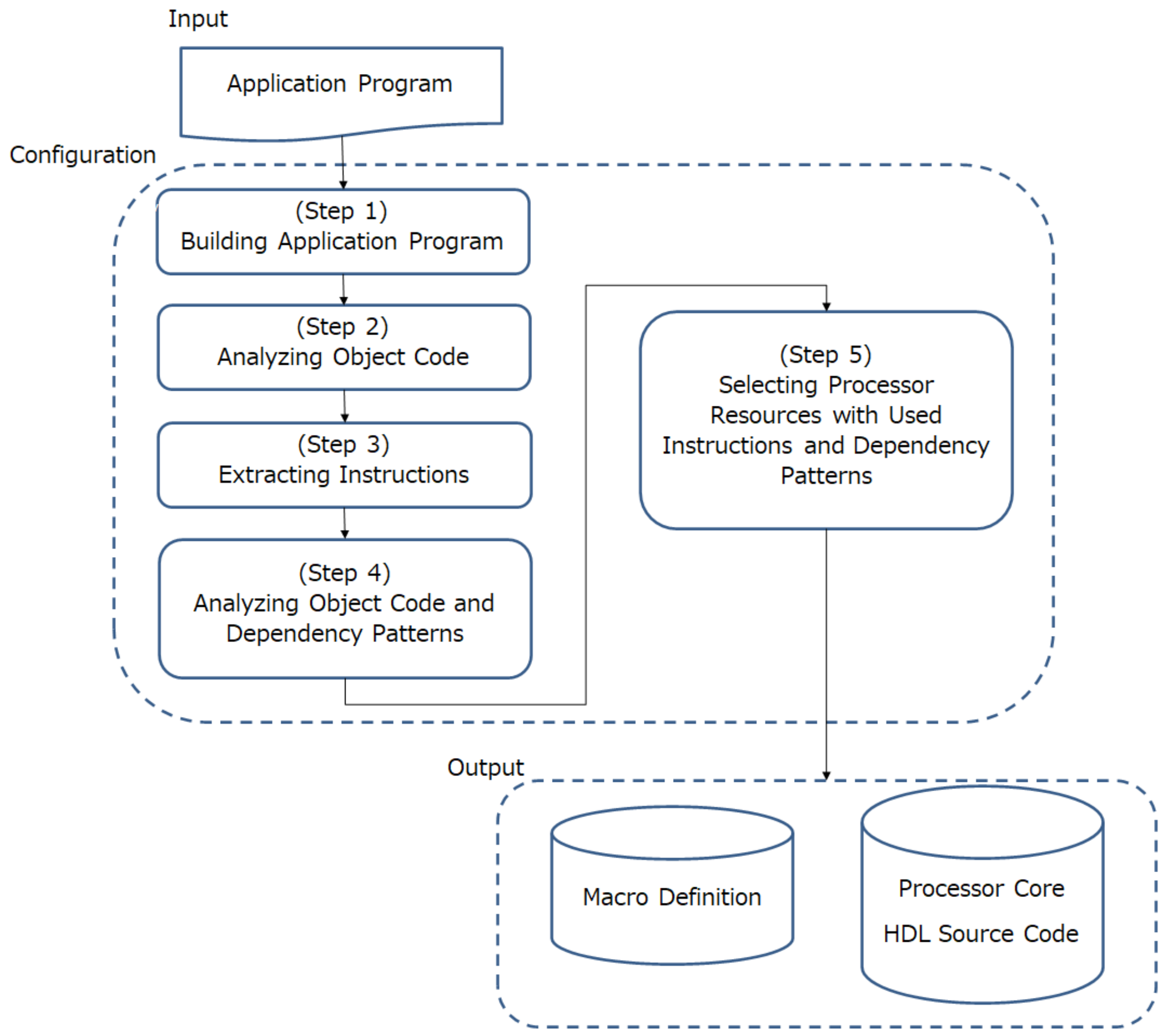

Input
Application Program
Configuration
(Step 1) Building Application Program
(Step 2) Analyzing Object Code
(Step 3) Extracting Instructions
(Step 4) Analyzing Object Code and Dependency Patterns
(Step 5) Selecting Processor Resources with Used Instructions and Dependency Patterns
Output
Macro Definition
Processor Core HDL Source Code

Figure 3.2: Analyzing application program.

3.2.4 Analyzing Object Code and Dependency Patterns

In this implementation of the processor core, to detect forwarding and hazard, the forwarding detection unit and hazard detection unit have to be implemented, whereas, if there is no code sequence causing forwarding or hazard, we can omit the circuit of the corresponding detection unit.

In this step, possibility of forwarding and pipeline stall is detected. The instruction sequence for dependency between instructions is searched, then the necessity of each forwarding or pipeline stall unit is decided. If some forwarding or pipeline stall patterns are found not to occur, the corresponding forwarding units or stall detection circuits are removed from the processor circuit. The details about the correspondence between dependency patterns and forwarding/stall detection units are described below, which is based on the literature [61].

Example of Disassemble

```
00000288 <main>:
     288: 27bdffe0  addiu $29,$29,-32
     28c: afbf001c  sw $31,28($29)
     290: afb00018  sw $16,24($29)
     294: 40806000  mtc0 $0,$12
     298: 3c020001  lui $2,0x1
     29c: ac40c034  sw $0,-16332($2)
     2a0: 3c020001  lui $2,0x1
     2a4: ac40c010  sw $0,-16368($2)
     2a8: 3c020001  lui $2,0x1
     2ac: ac40c004  sw $0,-16380($2)
     2b0: 3c020001  lui $2,0x1
     2b4: ac40c02c  sw $0,-16340($2)
     2b8: 3c020001  lui $2,0x1
     2bc: ac40c014  sw $0,-16364($2)
```

Figure 3.3: Example of the disassemble.

Dependency related to forwarding units

When an instruction uses the result of the former instructions, the forwarding unit detects the necessity of forwarding and designates the forwarding paths. If such dependencies are not found, the forwarding detection unit and the corresponding paths can be removed.

Forwarding patterns are classified into five cases in our processor core.

1. *Forwarding ALU result to the next instruction*: When the result of ALU calculation is used in the next instruction, the result is forwarded to the next instruction via a pipeline register. An example is as follows.

   ```
   add  $10, $8, $9
   sub  $12, $10, $11
   ```

   The two instructions, `add` and `sub`, have data dependency with respect to `$10`. The forwarding unit detects the dependency in the instruction sequence and controls the forwarding paths so that `sub` in EX stage immediately receives the result of `add` residing in MEM stage.

2. *Forwarding ALU result to the instruction immediately after the next instruction*:

   When the result of ALU calculation is used in the instruction immediately after the next instruction, the result is forwarded to the instruction. An example is as follows.

```
    add  $10, $8, $9
    sub  $14, $13, $12
    lw   $11, 0($10)
```

In this example, `lw` uses the result of `add` for address calculation. This type of dependency is solved by forwarding the value in WB stage to EX stage.

3. *Forwarding ALU result to conditional branch instruction*: When the result of ALU calculation is used in the conditional branch instruction just after the next instruction, forwarding from MEM stage to ID stage is performed. An example is as follows.

```
    add $10, $8, $9
    sub $14, $13, $12
    beq $10, $11, label
```

4. *Forwarding ALU result to the following store instruction*: When the result of ALU calculation is used in the next store instruction as the stored value, the value in WB stage is forwarded to MEM stage. An example is as follows.

```
    add $10, $8, $9
    sw $10, 0($11)
```

5. *Forwarding from jal,jalr*: Jump and link instructions (`jal`, `jalr`) write the return address in `$31` register. When the jump target instruction is a jump register instruction (`jr`) with `$31`, the return address being written in `$31` is forwarded to `jr`. An example is as follows.

```
    jal label
    nop
label:
    jr $31
```

Dependency related to stall detection units

When an instruction uses the result of the previous instruction but the forwarding unit cannot supply it immediately, the necessity of pipeline stall is detected by the stall detection unit. If such dependency is not found in the application program, the stall detection circuits can be removed.

There are five cases where the pipeline stalls.

1. *Result of load instruction used by the next instruction*: When the result of a load instruction is used by the next instruction, the pipeline has to be stalled. An example is as follows.

   ```
   lw $8 0($9)
   add $9, $8, $10
   ```

   In this case, after `add` is stalled in ID stage for one cycle while `lw` advances, it receives, in EX stage, the value forwarded from WB stage.

2. *Result of ALU/load instruction used by the next conditional branch*: When the result of ALU calculation or the load instruction is used in the next conditional branch instruction, pipeline has to be stalled. An example is as follows.

   ```
   add $10, $8, $9
   beq $10, $11, label
   ```

   In this case, `beq` stalls in ID stage for one cycle and is given the value from MEM stage. If the prior instruction is `lw`, it needs two-cycle stalling.

3. *Result of load instruction used by conditional branch after the next instruction*: When the result of a load instruction is used in a conditional branch instruction after the next instruction, the pipeline has to be stalled for one cycle. An example is as follows.

   ```
   lw $10, 0($9)
   nop
   beq $10, $11, label
   ```

   `beq` stalls in ID stage for one cycle and is given the value from WB stage.

4. *Result of ALU used by jr or jalr*: When the result of ALU calculation is used in the next jump register or jump and link register instruction, the pipeline has to be stalled. An example is as follows.

   ```
   add $10, $8, $9
   jr  $10
   ```

   In this case, `jr` stalls in ID stage for one cycle.

5. *Result of load instruction used by jr or jalr after the next instruction*: When the result of a load instruction is used in jr or jalr after the next instruction, the pipeline has to be stalled. An example is as follows.

   ```
   lw $10, 0($9)
   nop
   jr $10
   ```

   In this case, `jr` stalls in ID stage for one cycle.

3.3 Selecting Processor Resources with Used Instructions and Dependency Patterns

Resources/components which are required for each instruction to be executed are listed in advance. Similarly, the information about the units necessary for solving dependencies is prepared. Using this list and information for correspondence between instructions/sequences and necessary components, once the analysis in the previous section is done, all the processor resources for running the application program are fixed. The target resources for configuration are described below.

1. *multiplexer*: There exist several multiplexers in the processor core. Whether each multiplexer is used or not depends on the selected instructions. If some instructions are not actually used for the application program, the corresponding multiplexers, related data paths, and control logic are removed.

2. *Adder for PC*: When a branch is taken, an adder is used to calculate the branch target address. The current program counter value plus four (instruction size) and immediate value are added. This adder is removed if the target program does not include branch instructions, although it is not likely to happen.

3. *Comparator for equality condition*: Comparators for equality condition are used for `beq, bne` instructions. If branch instructions with equality condition are not included in the application, the comparators are removed.

4. *Comparator for positivity condition*: A comparator for positive condition is used for `bgez` and `bltz` instructions. If the application does not include branch instructions with positivity condition, this comparator is removed.

5. *Comparator for negativity condition*: A comparator for negativity condition is used for `blez` and `bgtz` instructions. If branch instructions with negativity condition do not appear, the comparator is removed.

6. *Multiplier*: A multiplier is used for multiplication. If the application does not include multiplication instructions, this component is eliminated.

7. *Divider*: A divider is used for division. Embedded applications tend to avoid integer division. Therefore, this is removed for many applications.

8. *Forwarding unit*: As described in Section 3.2.4, there are five cases for the forwarding to work. While each case involves its detection circuit and related connections, we remove them if the forwarding never happens.

9. *Stall detection unit*: There are five cases where the pipeline has to be stalled as described in Section 3.2.4. While each case needs its detection circuit and related connections, we remove them if the case is never met.

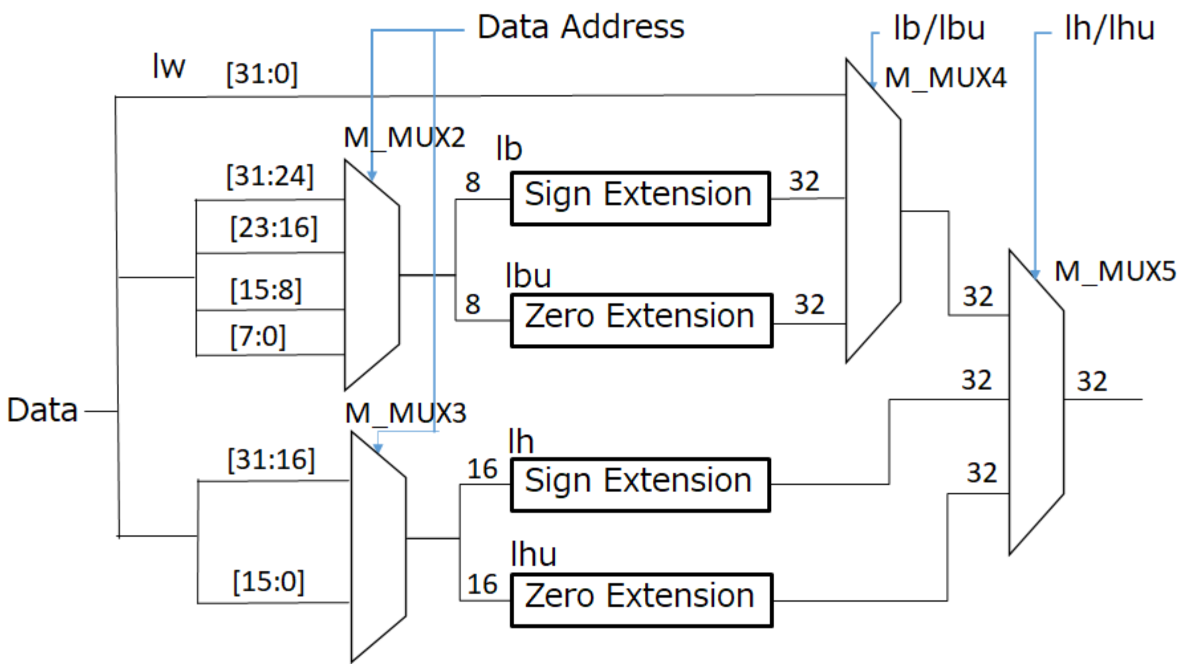


Figure 3.4: Example of selecting multiplexer.

3.4 Generating Application Adaptive Processor Core Circuit

For the resources described in the previous section, from the result of disassembling an application program, actually used resources for the application program are selected. Processor core descriptions and a macro definition file are the output in the previous step. Selected resources for the processor configured for the target application are indicated by the macro definition file. Figure 3.4 illustrates a part of the processor data path with multiplexers, which are used for a byte data selection with the load instruction. The multiplexer `M_MUX2` is not used if there is not load byte data instruction in an application program and the multiplexer `M_MUX3` is not used if there is not load half word data instruction in an application program. In addition, in the case that there is not load byte data, the selection of `M_MUX4` is not necessary. In order to make this selection possible, a directive is inserted in the source code of the processor core described in HDL (Verilog) in advance as the code of Fig.3.5.

In the case that `M_MUX3` in Fig.3.4 is not used in an application program, `memmux_3_nouse` is defined if there is not load half word instruction. On the other hand, if load half word data is used, `memmux_3_mux` is defined. These defines are an output of a configuration tool. Other resources are selected in a similar way. In the case that half word instruction is not used in an application program, the circuit block becomes as in Fig.3.6. As we can see from the figure, multiplexers, `M_MUX3` and `M_MUX5` are removed from the processor core, which contributes to reducing the size of the processor core circuit.

Macro names are expressed in the format:

```
‘define Stage_ResourceNo_Selection
```

Example of Multiplexer Selection

```
/* MUX2 MEM */
‘ifdef memmux_2_nouse
    ;
‘elsif memmux_2_mux
    MUX8_4to1 CORE_MUX8(.A(DATA_IN[31:24]), .B(DATA_IN[23:16]),
    .C(DATA_IN[15:8]), .D(DATA_IN[7:0]),
    .SEL(DATA_ADDR[1:0]), .Z(DATA_IN2_8));
‘else
/* ERROR */
Never Reached;
    assign DATA_IN2_8 = 32’hf0200bad;
‘endif
```

Figure 3.5: Example of multiplexer selection.

The `Stage` part is as follows.

ID stage	`idmux`
EX stage	`exmux`
MEM stage	`memmux`
WB stage	`wbmux`

`ResourceNo` is a sequential number of each resource. If a multiplexer is selected, `mux` or `mux3` (for 3-to-1 multiplexer) is specified for `selection`. Otherwise, `nouse` is for the unused multiplexer.

Using the results of the analysis process, the configurator makes a macro definition file as follows.

```
‘define idmux_1_mux3
‘define idmux_2_mux
‘define idmux_3_mux
‘define idmux_4_mux3
‘define idmux_5_mux3
  ...     ...
```

Synthesizing the source codes with the macro definition file, proper resources are chosen to build the application specific processor.

3.5 Experimental Result

3.5.1 Application Programs

In order to evaluate the effect of the processor adaptation, we evaluated three applications, the matrix multiplication, Rijndael and quick sort program. Features of these applications

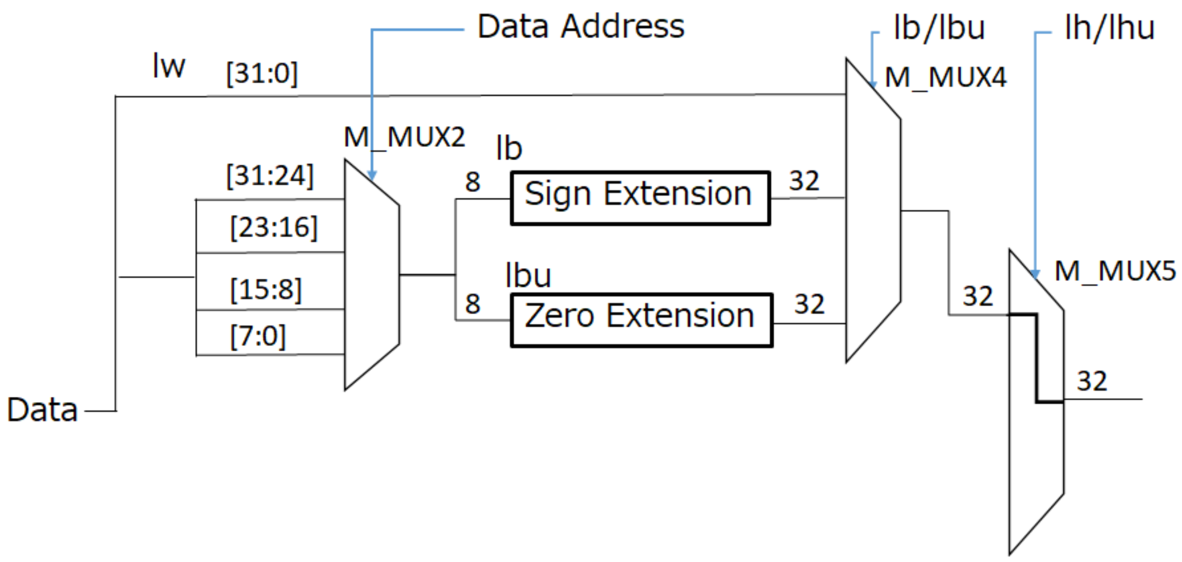


Figure 3.6: Example of selecting multiplexer (Adapted).

are as follows.

- Matrix multiplication

  In Matrix multiplication program, the size of matrices is relatively small, 4 x 4 since the instruction set is not varied if the size of matrices is increased. The initial data of the matrices are stored in Read only area, the address of which is shown in Fig.3.1 and the result of multiplication is stored in RAM for data area. This program uses the multiplier. The actually used data of the matrices are shown below.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{pmatrix} = \begin{pmatrix} 130 & 140 & 150 & 160 \\ 306 & 332 & 358 & 384 \\ 482 & 524 & 566 & 608 \\ 658 & 716 & 774 & 832 \end{pmatrix} \tag{3.1}$$

- Rijndael
  Rijndael in Ref.[62] is a cypher algorithm which is commonly used as AES. In this program, an encrypted data is stored in a Read Only Data area so that around 2.8K-byte Read Only Data area is used for a table data which is used in the algorithm. We confirmed the target data was properly decrypted with this algorithm.

- Quick sort
  Quick sort is a popular algorithm of sorting. In this evaluation, one hundred of integer data are stored in Read Only Data area and the result is stored in RAM for data area and is output with the UART interface on an evaluation board to confirm correctness of the result. The application program from Ref.[63] is used.

Table 3.2: Processor core resources.

Resource	Matrix	Rijndael[62]	Qsort[63]	Full
LUT	3954(19.01%)	3697(17.77%)	3835(18.44%)	8481(40.77%)
LUTRAM	1536(16.00%)	1536(16.00%)	1536(16.00%)	1536(16.00%)
FF	1765(4.24%)	1701(4.09%)	1695(4.07%)	1766(4.25%)
BRAM	11.50(23.00%)	11.50(23.00%)	11.50(23.00%)	11.50(23.00%)
DSP	4(4.44%)	–	–	8(8.89%)
WNS (ns)	0.333	0.298	0.345	-0.205
Fmax (MHz)	69.575	69.406	69.634	67.065
Power (W)	0.238	0.223	0.226	0.34

3.5.2 Evaluation of Processor Adaptation

We evaluated the effect of the processor adaptation. Xilinx Artix-7 FPGA (XC7A35T-ICPG236C) in Ref.[64] on the BASYS3 board of Digilent in Ref.[60] and the Xlinx development tool Vivado 2018.3 are used for the evaluation. GCC 4.3.3 is used for the compiler. For application programs described in Section 3.5.1, FPGA resources which each adaptive processor uses are shown in Table 3.2. Full is a processor core which is not adapted to an application program. The clock rate of the processor (CLK) is 68.0MHz, which is generated by the IP core from the source clock rate of 100MHz. WNS stands for Worst Negative Slack and Fmax is calculated by $Fmax = 1000/((1000/(1000/CLK) - WNS))$ in Ref.[65]. Power is Total On Chip Power, which is estimated by a Xilinx design tool, Vivado in Ref.[66].

From Table 3.2, it can be seen that the number of LUT resources are reduced by 56.4% (Rijndael) to 53.4% (Matrix) from the full set of the processor (Full) and Fmax is higher than 68.0MHz for the adapted cases while the timing is not met in the Full, not adapted case.

3.5.3 Discussion

Table 3.3 shows the resource usage of Qsort[63] to see the effect of the adaptation in detail. The column “Adapted” is cited from Table 3.2 for the case of adaptation applied. The column “w/ Multi.” is the case of “Adapted” with additional resources for multiplication. The column “w/ Div. signed” is the case of “Adapted” with additional resources for signed division as we can select the signed division and/or the unsigned division for the resources of division. The column “w/o load half word” shows the resources of the case that load half word instruction related resources are removed from “No Adaptation” as in the Fig.3.6. It can be seen that about 0.3% of LUT can be reduced by the removing load half word related resources. We can also see that 3.6% and 62.6% of LUT are increased with multiplier and signed divisor, respectively.

Table 3.3: Processor core resources in detail (Qsort[63]).

Resource	Adapted	w/ Multi.	w/ Div. signed	w/o load half word	No Adaptation
LUT	3835	3972	6235	8459	8486
LUTRAM	1536	1536	1536	1536	1536
FF	1695	1765	1767	1765	1766
BRAM	11.50	11.50	11.50	11.50	11.50
DSP	–	8	–	8	8
WNS (ns)	0.345	0.244	-0.073	-0.219	-0.402
Fmax (MHz)	69.634	69.147	67.664	67.002	66.191
Power (W)	0.226	0.235	0.292	0.342	0.34

3.6 Multicore

The techniques presented in this section are ones proposed in the author's literature [57], [67], [68] and [69]. In our study, we have developed multicore processors on an FPGA for some specific applications. We tried to adapt the processor to each application and confirmed that an eight-core processor could be implemented on a relatively small FPGA. Then, we extended the development environment to make the multicore processor adaptive to an application. This environment consists of application program analysis and creation of optimum multicore processor RTL descriptions for the application. In the following sections, we describe the functions of the development environment in detail and show the evaluation results for the circuits which were generated by the environment.

A multicore processor is constructed with a combination of the cores adapted to the application programs. Figure 3.7 shows the organization of an eight-core processor. The number of cores is given to the development environment (configurator) in advance. Each core has on-chip instruction memory (IMEM) which stores instructions (program codes) it runs and is equipped with data cache. When a multicore processor is configured, the configurator outputs Verilog-HDL descriptions for the indicated number of cores and the coretop module (Fig.3.7) which connects the cores, the instruction memories and the data caches.

3.6.1 Cache

For inter-processor communication, we implemented external memory interface with cache memory so that all processor cores can access the commonly used data. In this study, we implemented DDR2 memory interface on an evaluation board we used. Data which are commonly used in processors are accessed through the cache memory interface. Each core has a write-back cache memory, whose block size is 16 bytes (4 words). It takes one clock cycle for each core to read from or write to its cache memory. The cache memory size, the number of ways, and the width of memory addresses are selectable. One-way (direct-mapped) or two-way set associative structure can be chosen. The size is configurable and selected from 4KB, 8KB and 16KB for one-way cache, while 8KB (4KB+4KB), 16KB (8KB+8KB), and 32KB (16KB+16KB) for two-way. In addition, the width of memory

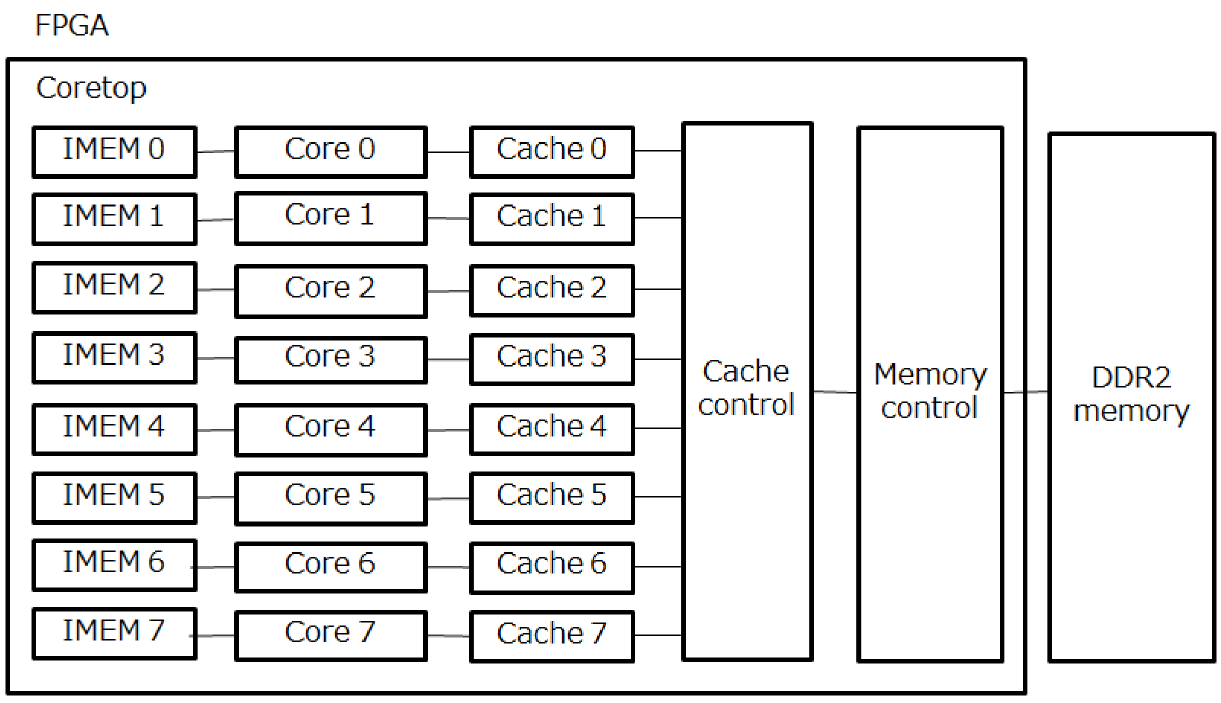


Figure 3.7: Multicore processor structure.

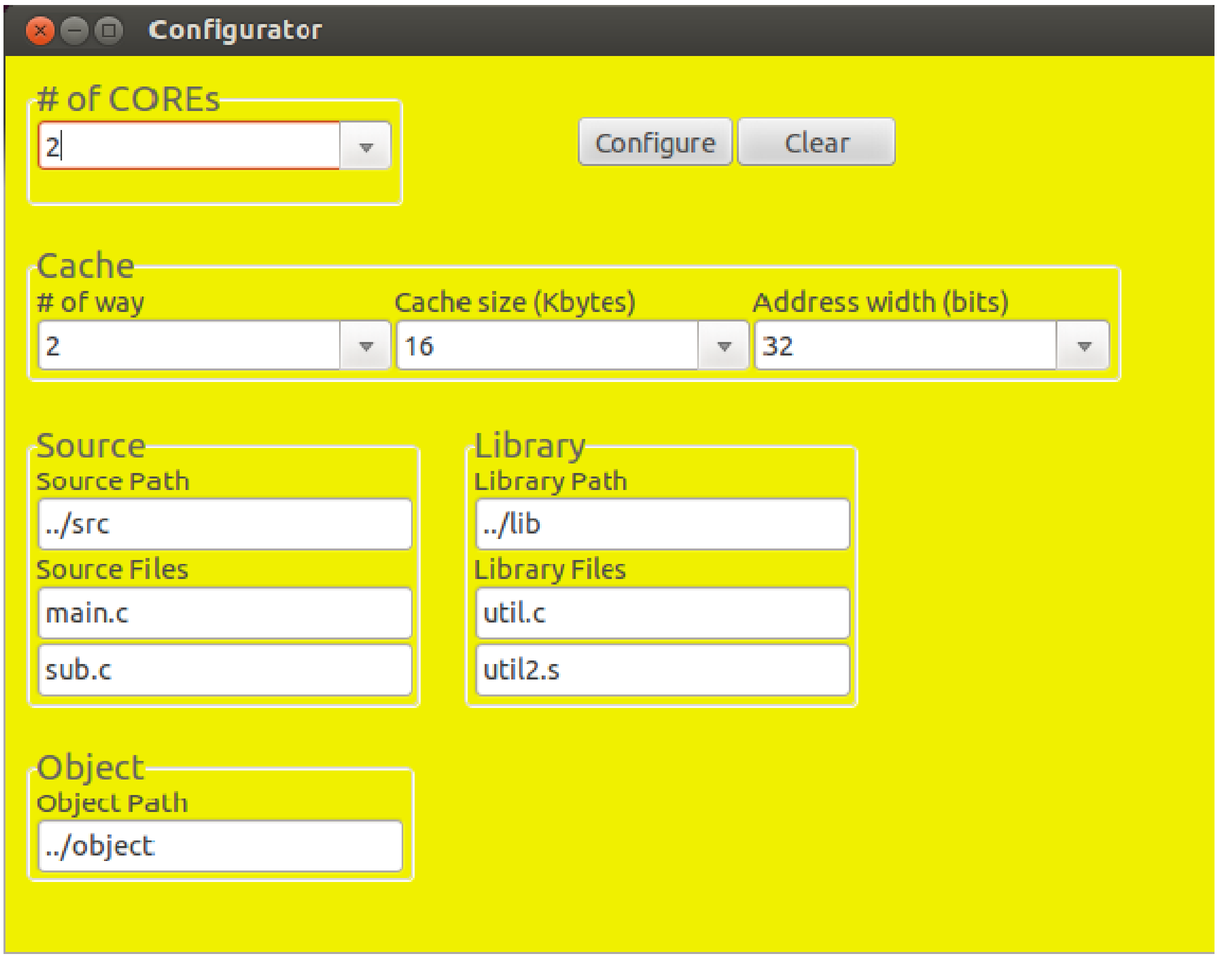
addresses can be 16 bits or 32 bits, which leads to a direct influence on the size of tag memory and the access delay/latency. When the target application processes relatively small amount of data, the address width can be reduced according to the size of data set.

To support the multicore configuration, an arbiter is embedded in the memory control module. The arbiter receives requests for accessing the external memory from the caches and decides which request is prioritized. Implementing snooping mechanisms is avoided to keep the hardware size and complexity small. Instead, data coherence is maintained explicitly by, when needed, flushing updated data by software. The configuration in terms of size, associativity, and address width has an effect on the processor size and the maximum running clock frequency. The evaluation results are shown in Section 3.6.5.

3.6.2 Multicore Configuration Environment

We have developed the configuration environment to generate the multicore processor of which each core is adapted to individual application programs.

Figure 3.8 shows the graphical user interface of the configurator, which accepts the configuration settings and application files information, and invokes the automatic generation of the application specific processor descriptions. This configurator has the following interfaces:


Configurator

# of COREs: 2

Configure | Clear

Cache
# of way: 2
Cache size (Kbytes): 16
Address width (bits): 32

Source
Source Path: ../src
Source Files: main.c, sub.c

Library
Library Path: ../lib
Library Files: util.c, util2.s

Object
Object Path: ../object

Figure 3.8: Multi core configurator GUI.

[# of COREs]	Selecting the number of cores
[Cache]	Specifying the cache structure
[Source]	Locating application's source files
[Library]	Locating library files
[Object]	Designating object path
[Configure]	Starting configuration
[Clear]	Clearing generated objects

Before starting the configuration, the number of processor cores, the cache structure, and the application program information are input in advance. The number of cores can be chosen from one to eight. For the cache information, one-way (direct-mapped) or two-way can be selected, and cache size can be selected from 4KB to 32KB. The width of memory addresses can be 16 bits or 32 bits.

Application source code information is to be input before starting the configuration. The source code path and file names of the application program are input in the configu-

Table 3.4: Multicore processor implementation results (1-8 cores) (Matrix).

# of Cores	1		2		4		8		-
	Adapted	Full	Adapted	Full	Adapted	Full	Adapted	Full	Available
Register	1,423	1,522	2,449	2,723	4,467	5,388	8,169	9,133	54,576
LUT	2,186	8,593	3,885	16,415	7,351	38,965†	13,910	78,594†	27,288
Slice	758	2,680	1,277	5,047	2,742	–	5,025	–	6,822
RAMB16WER	14	14	26	26	50	–	98	–	218
RAMB8WER	–	–	–	–	–	–	–	–	
DSP48A1	4	8	8	16	16	–	32	–	58
Min period (ns)	12.873	13.073	12.926	13.121	13.663	–	15.331	–	–
Max freq. (MHz)	77.682	76.494	77.363	76.214	73.190	–	65.227	–	–

† It exceeds the maximum LUTs in this FPGA, so it cannot be implemented.

rator as well as the library path and file names the program uses. Besides that, the object path, in which the configured Verilog-HDL files are generated, is designated.

3.6.3 Building Application Program

The configurator generates the processor circuit descriptions in HDL (Verilog). In the first process of the configuration, the configurator invokes the compiler (GCC) with the application source codes written in C language or MIPS assembly language and generates the object codes.

3.6.4 Analyzing Object Codes

The object codes of the target application are analyzed in the way described in Section 3.2.

3.6.5 Evaluation of Multi Core

We have evaluated the multicore processor generated by the configuration environment described in the previous sections with Spartan-6 xc6slx45 in Ref.[70] on Digilent Atlys board in Ref.[71]. Xilinx ISE Design Suite 14.7 in Ref.[72] was used to implement a processor circuit on the FPGA. The results of the processor implementation in terms of the usage of resources and maximum frequency are shown below.

Evaluation Result

Table 3.4 shows the implementation results of one-core to eight-core processors for the parallelized matrix multiplication program. In order to execute a program in parallel, the calculation is separated in each processor core as shown in Fig.3.9. The program calculates the product of matrixA and matrixB. matrixB is divided into the same number of the processor cores in column direction and the divided part is assigned to each processor core so that the calculation is performed in parallel. Here, N denotes the size of the matrix, Example of Matrix (Multi Core).

```
n = N;
ncols = N/N_CORE;
for (i = 0; i < n; i++) {
  for (j = (ncols * my_rank); j < (ncols * (my_rank + 1)); j++) {
    for (k = 0; k < n; k++) {
      matrixC[i][j] += matrixA[i][k]*matrixB[k][j];
    }
  }
}
```

Figure 3.9: Example of matrix (Multi core).

N_CORE is the number of the processor core and my_rank shows the processor ID for this processor core.

To compare the results in terms of adaptation, we evaluated the processors with and without configuration. "Adapted" is the results by our configuration environment and "Full" is the results without the configuration. "Available" is the maximum number of resources in the target device we used (Spartan-6 xc6slx45). In these cases, cache memories of the processors are configured as a 16KB two-way set associative cache with 32-bit addresses.

These results indicate that FPGA resources (Registers, LUTs, Slices, RAMBs and DSP48A1s) to be used increase as the number of cores increases. The full set of instructions cannot be implemented on the FPGA device in the case of more than two cores, since the total amount of the LUTs used by the cores exceeds the maximum number of LUTs in this device. On the other hand, the adapted cores for the application can be accommodated in the FPGA device even in the case of the eight-core processor.

The number of registers increases according to the number of cores. For the adapted processor, it increases by 5.7 times (from 1,423 in the single-core to 8,169 in the eight-core) while it increases by 6.0 times (from 1,522 to 9,133) for the full processor. Similarly, the number of LUTs in the adapted processor increases by 6.4 times (from 2,186 to 13,910) and that in the full processor increases by 9.1 times (from 8,593 to 78,594).

When the resources are compared between the fully implemented processor and the adapted one, the number of registers and the number of LUTs are reduced by 6.50% (from 1,522 to 1,423) and by 74.6% (from 8,593 to 2,186), respectively, for the single-core. In the case of the two-core, they are reduced by 10.1% (from 2,723 to 2,449) and by 76.3% (from 16,415 to 3,885).

As the number of cores increases from one to eight, the maximum frequency decreases by 16.0% (from 77.682 MHz to 65.227 MHz) for the adapted processor. In comparison, the maximum frequency for the adapted processor is higher by 1.55% and by 1.51% than the fully implemented one in the case of the single-core (77.682 MHz to 76.494 MHz) and two-core (77.363 MHz to 76.214 MHz), respectively.

Table 3.5: 2way 32bit address.

Cache (bytes)	32K	16K	8K
Address (bits)	32	32	32
Register	2,449	2,449	2,447
LUT	3,855	3,885	3,850
Slice	1,380	1,277	1,333
RAMB16BWER	42	26	6
RAMB8BWER	6	–	20
DSP48A1	8	8	8
Min period (ns)	13.461	12.926	12.619
Max freq.(MHz)	74.289	77.363	79.246

Table 3.6: 2way 16bit address.

Cache (bytes)	32K	16K	8K
Address (bits)	16	16	16
Register	2,350	2,351	2,349
LUT	3,739	3,798	3,806
Slice	1,380	1,296	1,224
RAMB16BWER	38	22	6
RAMB8BWER	6	4	20
DSP48A1	8	8	8
Min period (ns)	13.455	12.918	12.61
Max freq.(MHz)	74.322	77.411	79.302

Cache configuration

To evaluate the effectiveness of the cache configuration, we generated processors in changing the cache memory size, associativity, and width of memory addresses.

Table 3.5 and 3.6 show the results of two-way set associative in which the address width is 32 bits and 16 bits, respectively. Table 3.7 and 3.8 are for one-way (direct-mapped) caches with 32-bit and 16-bit addresses, respectively. In both cases, the number of processor cores is two and each core is adapted to the matrix multiplication program.

The results demonstrate the effectiveness of reducing the cache memory size and the number of bits for addresses. While the maximum frequency of the 32KB two-way cache with 32-bit addresses is 74.289 MHz, that of the 8KB two-way cache with the same address width is 79.246 MHz, which is higher by 6.67%. Similarly, when we compare the maximum frequency between the 32KB two-way cache and 8KB two-way cache with 16-bit addresses, the improvement is from 74.322 MHz to 79.302 MHz, which is 6.70%. In the case of one-way with 32 bit addresses, the maximum frequency of the 4KB cache is higher than that of the 16KB by 4.00% (80.587 MHz to 77.519 MHz). With 16 bit addresses, the 4KB cache is higher than the 16KB by 1.69% (81.679 MHz to 80.321 MHz). As a whole, these results show that the reduction in the cache size contributes to raising the maximum frequency. Shortening the address width has small effects on the running frequency. The

Table 3.7: 1way 32bit address.

Cache (bytes)	16K	8K	4K
Address (bits)	32	32	32
Register	2,419	2,415	2,411
LUT	3,537	3,525	3,515
Slice	1,232	1,235	1,247
RAMB16BWER	24	16	6
RAMB8BWER	2	–	10
DSP48A1	8	8	8
Min period (ns)	12.9	12.47	12.409
Max freq.(MHz)	77.519	80.192	80.587

Table 3.8: 1way 16bit address.

Cache (bytes)	16K	8K	4K
Address (bits)	16	16	16
Register	2,342	2,338	2,341
LUT	3,511	3,519	3,511
Slice	1,344	1,326	1,263
RAMB16BWER	22	14	6
RAMB8BWER	2	2	10
DSP48A1	8	8	8
Min period (ns)	12.45	12.307	12.243
Max freq.(MHz)	80.321	81.255	81.679

improvement in the two-way caches is 0.06% at a maximum, while that in the one-way caches is 3.61% for 16KB.

The effectiveness of the cache associativity is examined. In the case of 32-bit addresses, while the maximum frequency of the 16 KB two-way is 77.363 MHz, that of the 16KB one-way becomes 77.519 MHz which is higher by 0.202%. In the case of 16 bit addresses, while the maximum frequency of the 16KB two-way is 77.411 MHz, that of the 16KB one-way becomes 80.321 MHz which exhibits 3.76% improvement.

In terms of FPGA resources, while the number of registers or slices is almost the same for all cache sizes, the total size of block RAMs (RAMB16 and RAMB8) is reduced as the cache size shrinks. This is because the size of data arrays decreases. For example, the 32KB two-way cache requires forty-two RAMB16s and six RAMB8s while the 8KB two-way uses six RAMB16s and twenty RAM8s. Similarly, the address width influences the total size of block RAMs. With the 16-bit address width, four RAMB16s are eliminated compared to the 32-bit width for the 32KB two-way cache. This is because the reduced tag width occupies a shorter area in tag arrays. Focusing on the associativity, the same trend can be observed for block RAMs.

In summary, comparison between the two-core full set (Table 3.4) and the adapted two-core processor with the minimum cache size, 4KB one-way with 16-bit addresses (Table 3.8) results in the conclusion that the combination of the adaptation to the target instruction sequences and the configuration of cache memory improves the maximum frequency by 7.17% (from 76.214 to 81.679) and reduces 14.0% (from 2,723 to 2,341) and 78.6% (from 16,415 to 3,511) of registers and LUTs, respectively.

The evaluation results showed that our approach with resource reduction in instruction execution as well as in cache memory is effective in reducing the occupied resources and improving the maximum frequency. Although the fully implemented processor has the limit of two cores in the small FPGA device, the adapted one which the configurator generates can consist of eight cores. We can expect a sixteen-core processor to be built in the same device depending on the application.

Our approach does not require that a new compact instruction set be designed to reduce hardware sizes and improve running frequency. In addition, the configurator can be applied with the existing instruction set without new compilers so that it helps to build customized processors in a fully automatic manner.

3.7 Summary of this Chapter

In this chapter, we explained the method to generate an application adaptive processor core with analyzing an application program and implementing only necessary resources for the application program. Evaluation results with matrix multiplication, Rijndael and quick sort show the reduction of FPGA resources and improvement of running frequency in execution. Additionally, we implemented two- to eight-core processors on an FPGA and evaluated the effect of the adaptation and the sizes of cache memory. It is shown that the eight of adapted cores for the application can be accommodated in the FPGA device. In contrast, only two cores can be implemented in the FPGA device when the adaptation is not applied. The proposed method does not need to introduce a new instruction set to generate a processor core adapted to an application program so that we do not need to develop a new compiler. We showed the method that the processor core can be generated in a fully automatic manner.

Chapter 4

Framework for Building Fine-Grained RTOS

4.1 Configuration Framework

Since the RTOS kernel is overhead for an application program, it is desirable that resources for RTOS be small and the execution time be short.

This chapter describes our method to achieve the objectives listed in Chapter 1:

1. We show the structure of RTOS we developed and what unnecessary codes are in Section 4.2.
2. In Section 4.3, the method to generate fine-grained RTOS in software RTOS is explained.
3. In Section 4.4, what is implemented in hardware RTOS is described and how the hardware is adapted to the application program is proposed.
4. We illustrate the detail of the automatic development environment in Section 4.5.

The framework presented in this chapter is based on the author's literature [53],[54], [56], [73], [74], [75] and [76].

4.2 RTOS Structure

We have implemented a software-only RTOS kernel and one which utilizes RTOS hardware. The former does not use RTOS hardware so that all functions of RTOS work as software on a processor.

Both of software-only RTOS and hardware RTOS are implemented as a subset of the standard profile in the μITRON4.0 specification in Ref.[7]. Functions we implemented in both software-only RTOS and hardware RTOS are as follows: Task Management Functions (act_tsk, iact_tsk, can_act, ext_tsk, ter_tsk, chg_pri), Task Dependent Synchronization Functions (slp_tsk, wup_tsk, iwup_tsk, can_wup, rel_wai, irel_wai), Semaphores (sig_sem, isig_sem, wai_sem, pol_sem), Eventflags (set_flg, iset_flg, clr_flg, wai_flg, pol_flg), and Data Queues (snd_dtq, psnd_dtq, ipsnd_dtq, fsnd_dtq, ifsnd_dtq, rcv_dtq, prcv_dtq).

Since the other system calls in the standard profile such as Fixed-Sized Memory Pool Management and Mailboxes include similar error checking and multiple attributes, the same adaptation techniques can be applied.

Hardware RTOS has a software part, which is the interface to RTOS hardware explained in Section 4.4.2. A software part of hardware RTOS is executed when hardware RTOS is used.

A source file of each system call has codes for software-only RTOS and a software part of hardware RTOS, which can be selectable by a directive. Whether a software-only RTOS kernel is used or a hardware RTOS kernel is used is decided manually when the adaptation tool runs.

4.3 Software Adaptation

Usually, only actually used system calls are linked with an application program as an RTOS kernel is provided as a library format. Nevertheless, these system calls include unnecessary codes for the application program. As defined in Section 1.1, we use the terms "fine-grained" and "adaptive" RTOS in this dissertation for an RTOS kernel in which unnecessary codes are eliminated by removing unnecessary codes caused by fixed attributes explained in Section 4.3.1 and by the way of calling explained in Section 4.3.2. Generating fine-grained RTOS is called "adaptation" in this dissertation.

4.3.1 Removing Unnecessary Codes Caused by Fixed Attributes

The technique presented in this section and the next section are ones proposed in the author's literature [75]. Some functions included in μITRON4.0 system calls are not used according to the attributes specified through parameters in a configuration file. In this case, the corresponding unnecessary code fragments can be removed from the source codes by manipulating macro descriptions explained below.

Removing unnecessary code fragments is explained with wai_sem system call as an example below. One attribute of wai_sem system call is specified by a parameter to the static API, `CRE_SEM` described in a system configuration file. This attribute provides two options for the wait queue, fifo order and priority order which are specified by TA_TFIFO and TA_TPRI, respectively.

There are three cases:

(1) Both the priority order and fifo order are used

(2) Only the priority order is used

(3) Only the fifo order is used

Directives for the three cases are inserted in the source codes of the system call in advance as in Fig.4.1. When both the priority order and the fifo order are used in the application source codes, which is used has to be determined at runtime. In this case, the code fragments for both the usages are located in the corresponding system call. The result of a code after the attribute is analyzed are shown in Fig.4.2. When only the priority order is used in the application, the code fragment only for it is validated. The result in selecting only the priority order is shown in Fig.4.3. Similarly, only the fragment for the fifo order is selected if it is the only usage in the application. This leads to reduction of the code size and execution time. Figure 4.4 illustrates the result of Case(3), only fifo order is selected.

Example of the Directives

```
#ifdef CHK_SEM_PRI
  if ((p_sem->sematr & TA_TPRI) != FALSE) {
    /* priority queue */
    _kernel_queue_insert_tpri( ... );
  }
#endif /* CHK_SEM_PRI */
#ifdef CHK_SEM_FIFO
  if ((p_sem->sematr & TA_TPRI) == FALSE) {
    /* FIFO queue */
    _kernel_queue_insert_prev( ... );
  }
#endif /* CHK_SEM_FIFO */
```

Figure 4.1: Example of the directives.

Example of the Directives: Case(1) Both are used

```
  if ((p_sem->sematr & TA_TPRI) != FALSE) {
    /* priority queue */
    _kernel_queue_insert_tpri( ... );
  }

  if ((p_sem->sematr & TA_TPRI) == FALSE) {
    /* FIFO queue */
    _kernel_queue_insert_prev( ... );
  }
```

Figure 4.2: Example of the directives: Case(1) Both the priority order and fifo order are used.

4.3.2 Removing Unnecessary Codes Caused by the Way of Calling

Each system call in the μITRON4.0 specification includes code fragments for checking errors. Although the μITRON4.0 specification implies that error detection can be omitted for each main error class, it may fail to notice an error which has to be detected, leading to unexpected troubles.

Example of the Directives: Case(2) Only priority order

```
if ((p_sem->sematr & TA_TPRI) != FALSE) {
  /* priority queue */
  _kernel_queue_insert_tpri( ... );
}
```

Figure 4.3: Example of the directives: Case(2) Only the priority order is used.

Example of the Directives: Case(3) Only fifo order

```
if ((p_sem->sematr & TA_TPRI) == FALSE) {
  /* FIFO queue */
  _kernel_queue_insert_prev( ... );
}
```

Figure 4.4: Example of the directives: Case(3) Only the fifo order is used.

In the method we propose, checking codes for errors which never occur in the application are removed so that only necessary error checking exists in the object code. This is done by analyzing the application program.

The procedure of checking each system call is as follows. First, C language preprocessor is applied to an application source file to expand header files and macro definitions. Next, each call for system calls is checked and its parameters are extracted. From the parameters, possible errors at runtime are identified and the corresponding macro definitions are output. Here, since parameters originally expressed as symbolic constants defined in header files are translated into numeric values, the values are to be directly considered. On the other hand, if a parameter is given as a variable, the parameter needs to be checked at runtime, since the value of the variable is not decided statically. In this case, a macro definition that indicates the necessity of checking of the parameter at runtime is output. In addition, the information about the number of resources which is passed from the system configuration analysis is compared to the usage of the resources. The analyzing process above is illustrated in Fig.4.5.

The macro definition file created by the analysis of an application, a system configuration file for μITRON4.0 convention, and RTOS kernel source codes are input to a cross compiler environment for software-only RTOS kernel. For hardware RTOS kernel, the macro definition file is passed to a hardware synthesizing tool with RTOS kernel HDL source code as described in Section 4.5.

Each system call in the μITRON4.0 specification has one or more possible errors and their causes. From Table 4.1 to Table 4.4 show the case of semaphore related system calls. As for the E_ID error checking, if all semaphore IDs are confirmed to be within the proper range, omitting semaphore ID checking has no effect on the behavior of the application program, so that the code of the error checking is removed from the RTOS kernel and

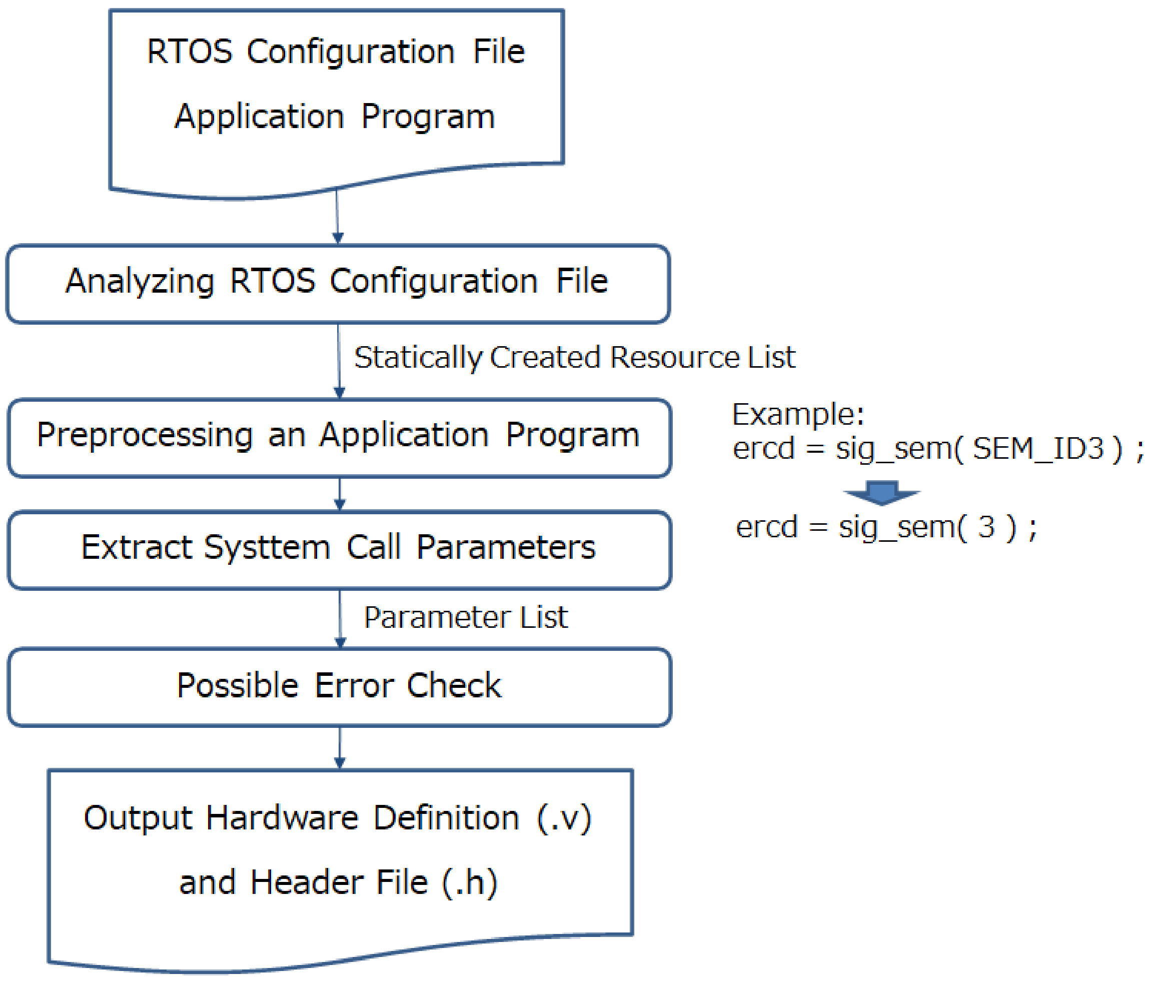

RTOS Configuration File
Application Program
Analyzing RTOS Configuration File
Statically Created Resource List
Preprocessing an Application Program
Example:
ercd = sig_sem(SEM_ID3) ;
ercd = sig_sem(3) ;
Extract System Call Parameters
Parameter List
Possible Error Check
Output Hardware Definition (.v)
and Header File (.h)

Figure 4.5: Analyzing process.

the overhead can be reduced. For the E_NOEXS error checking, the step of the system configuration file analysis recodes IDs for created resources (semaphores) in advance of this error checking procedure. The list of these IDs is delivered to this procedure, so that all the ID values used in the application are checked and, if they are all found in the list, the code fragment for checking E_NOEXS is omitted. On the other hand, E_CTX cannot be checked statically since the error condition depends on the runtime situation of the application program, so the symbol "–" is put in the fourth column of the tables. Possibility of E_RLWAI can also be statically checked.

After all errors as well as fixed attributes are checked, a hardware definition file for the hardware RTOS kernel, a file for static resource creation, and a header file are generated by the adaptation tool as shown in Fig.4.14.

Table 4.1: System call and error cause (sig_sem).

System call	Error	Description	What is checked
sig_sem	E_CTX	Context error	–
	E_ID	Invalid Semaphore ID	The range of semaphore ID
	E_NOEXS	Semaphore ID Non-existent	Whether ID is created by CRE_SEM
	E_QOVR	Queue overflow	semaphore count

Table 4.2: System call and error cause (isig_sem).

System call	Error	Description	What is checked
isig_sem	E_CTX	Context error	–
	E_ID	Invalid Semaphore ID	The range of semaphore ID
	E_NOEXS	Semaphore ID Non-existent	Whether ID is created by CRE_SEM
	E_QOVR	Queue overflow	semaphore count

4.4 Hardware Adaptation

Hardware RTOS consists of an RTOS hardware circuit and software part. To improve the performance and reduce the footprint of software, RTOS functions for static error check, task status check, dynamic error check, queue operations, getting the highest priority task and changing task status are implemented in the RTOS hardware circuit and it returns the task ID of the runnable highest priority task while task switching is implemented as software. Figure 4.6 illustrates a source code of act_tsk system call for software-only RTOS as an example. Codes indicated by blue boxes are implemented in hardware in the hardware RTOS. The software part of Hardware RTOS is described in Section 4.4.2 and shown in Fig.4.7.

Since the source code of the RTOS hardware circuit is written in HDL (Verilog), the adaptation method described in the previous sections can be applied to the RTOS hardware circuit as well as the software-only RTOS, so that the hardware resources can be reduced.

Table 4.3: System call and error cause (pol_sem).

System call	Error	Description	What is checked
pol_sem	E_CTX	Context error	–
	E_ID	Invalid Semaphore ID	The range of semaphore ID
	E_NOEXS	Semaphore ID Non-existent	Whether ID is created by CRE_SEM
	E_TMOUT	Poling failure	semaphore count

Table 4.4: System call and error cause (wai_sem).

System call	Error	Description	What is checked
wai_sem	E_CTX	Context error	–
	E_ID	Invalid Semaphore ID	The range of semaphore ID
	E_NOEXS	Semaphore ID Non-existent	Whether ID is created by CRE_SEM
	E_RLWAI	Forced release from waiting	rel_wai is called

Table 4.5: Addresses for RTOS system calls.

Address	R/W	Operation
0xffff0008	R	Read RTOS return code
0xffff0100	W	Issue RTOS system call
0xffff0104	W	Set RTOS system call 1st parameter
0xffff0108	W	Set RTOS system call 2nd parameter
0xffff010c	W	Set RTOS system call 3rd parameter
0xffff0110	W	Set RTOS system call 4th parameter
0xffff0114	W	Set RTOS system call 5th parameter
0xffff0120	R	Read RTOS return parameter

4.4.1 Structure of Hardware RTOS

Figure 4.8 roughly shows a structure of the processor core and the RTOS hardware circuit we have implemented. We designed the soft processor core, of which instruction set architecture is MIPS32 in Ref.[58]. RTOS hardware is accessed by memory mapped I/O.

Figure 4.9 depicts the RTOS hardware structure. The RTOS hardware consists of two parts, "RTOS Hardware Wrapper" and "RTOS Hardware Core". RTOS Hardware Wrapper, which is the interface between the processor core and RTOS Hardware Core, works as a state machine. When RTOS Hardware Wrapper receives an address, which indicates a command to the RTOS hardware, and data, which indicates a system call number or parameters, RTOS hardware starts to work, so that an operation such as a queue operation and input data (if any) are passed to RTOS Hardware Core.

4.4.2 Interface to RTOS Hardware

The structure of the system call software part is explained in this section. The software running in the processor core reads from or writes to the addresses in Table 4.5.

"R" in the column "R/W" indicates that a value read from the corresponding address is a return value from the hardware. On the other hand, "W" indicates that a value is written to the address so that the value such as the system call number and other parameter values is delivered to the RTOS Hardware Wrapper.

Before the software issues a system call, it writes the parameter values to the same number of addresses (starting at 0xffff0104) as arguments defined for the system call. After all the parameters are set, the software issues the system call.

A system call is issued by writing the system call number to the corresponding address

(0xffff0100). This makes the system call start by changing the state of the hardware. Then, the software reads from the address for a return code (0xffff0008) so that it checks completion of the processing and receives a task ID of the highest priority task and a return value from the system call. That is, the most significant bit of the read value indicates the completion of the RTOS hardware, and the lower bytes contain a highest-priority task ID and a return code. This is a busy-waiting procedure where, after the software writes the system call number to the address for "Issue RTOS system call" (0xffff0100), it repeatedly reads from the address for "Read RTOS return code" (0xffff0008) until it finds the most significant bit of 1. Then, it recognizes the lower bytes as a return code, and proceeds to the following processing.

Some system calls return not only a return code but the other results through call by reference. For example, wai_flg returns a flag pattern through an address which a parameter specifies. In this case, the result is obtained by reading from the address dedicated to call by reference (0xffff0120).

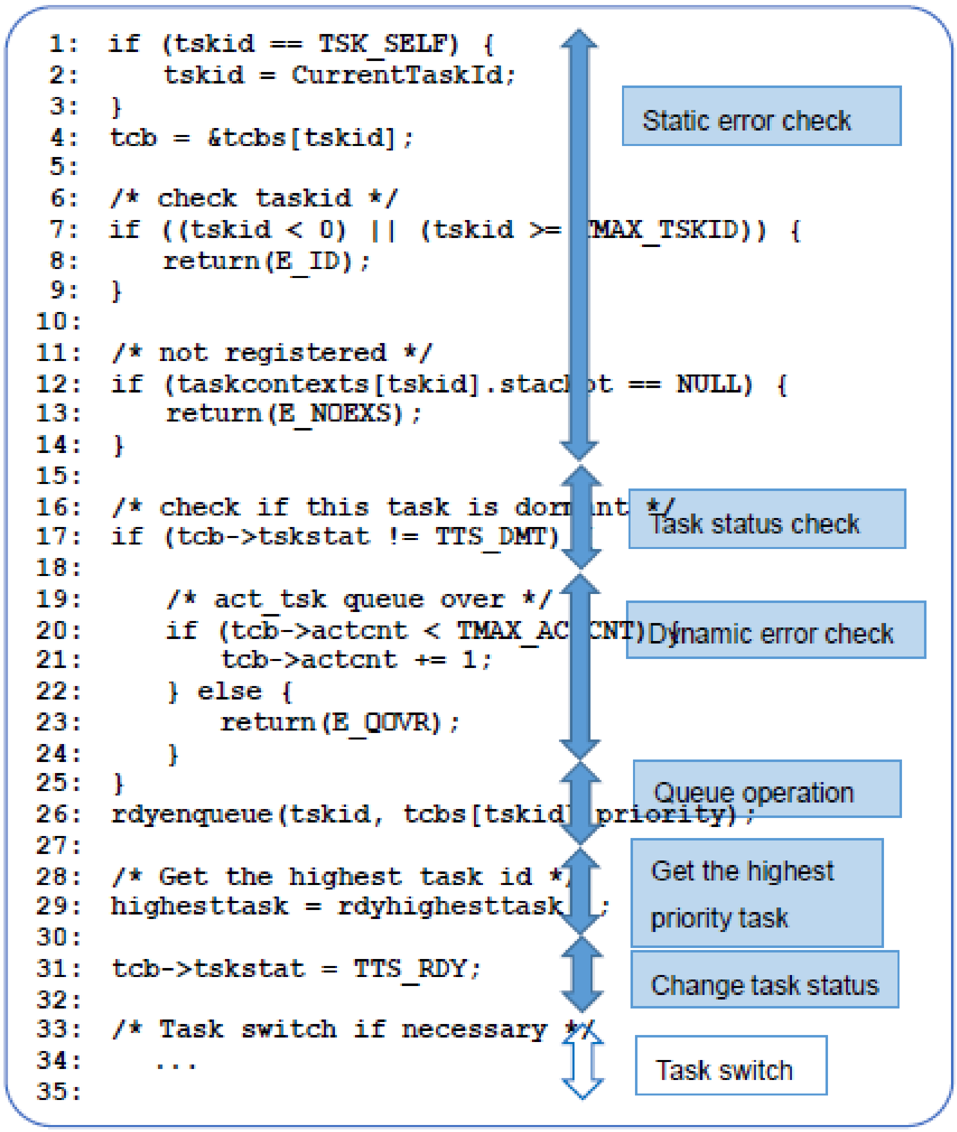
Figure 4.10 is the flow of the software part procedure for act_tsk. The software part code corresponding to Fig.4.10 is shown in Fig.4.7. Other system call functions follow a similar flow and the software part code. The software part waits for returning from hardware RTOS with polling. There is another option of using interrupt mechanisms for the completion notification. We chose polling, not interrupt, since interrupt is disabled in the system call function. In general, interrupt leads to overhead of detecting interrupt cause and context switch, while polling leads to only reading a hardware register.

4.4.3 RTOS Hardware Wrapper

RTOS Hardware Wrapper is the interface between a processor core and RTOS Hardware Core. In RTOS Hardware Wrapper, a hardware circuit which checks possible errors is implemented, so the circuit can be reduced by adaptation. In RTOS Hardware Core, TCBs (Task Control Blocks), queue headers and the circuit to control these resources are implemented, which is explained in Section 4.4.4 in detail. With our implementation, since RTOS Hardware Wrapper works as a state machine, the next state of each state is decided by the current state and input data. All of states are shown in Table 4.6. In the several states, an operation for RTOS Hardware Core described in Table 4.7 is issued. Which operation is issued is shown in the column Operation of the each state if there is an operation for RTOS Hardware Core. The state transition is shown in Fig.4.11. Initially, the state begins from INIT state after the system reset. Next, the state is in WAIT state for waiting a system call invoked from an application program. After a system call is called, the state goes to CHECK state to check possible errors. Then a state transits to several states to perform the system call. If a task switch is possible, the state goes to HIGHEST state, otherwise the state moves to END state and returns to WAIT state to wait a system call. When there is a task switch, the state transits to ENDSWITCH through END state and returns to WAIT state to wait a system call. The details are explained in Appendix A.

Table 4.6: States of RTOS hardware wrapper.

State	Operation	Summary of the state
INIT	–	Initialize RTOS hardware registers.
WAIT	–	Wait for a system call invoked.
CHECK	–	Check parameters in a system call.
SETATTR	–	Set attribute to RTOS hardware internal registers.
TASKSTATUS	TASKSTATUS	Check a task status.
SEMSTATUS	SEMHEAD	Check a semaphore status.
FLGSTATUS	FLGHEAD	Check an eventflag status.
DTQSTATUS	DTQHEAD	Check a data queue status.
CHECKTASK	–	The result of TASKSTATUS is checked.
CHECKSTATUS	–	The result of SEMSTATAUS, FLGSTATUS or DTQSTATUS are checked.
ACTCNT	–	The activation count is increased.
SEMCNT	–	The semaphore count is increased.
RDYDEQUEUE	PRIDEQUEUE	Release a task from the ready queue.
SEMDEQUEUE	SEMDEQUEUE	Release a task from the semaphore waiting queue.
CHGPRI	PRICHG	Change the task priority.
SEMENQUEUE	SEMENQUEUE	Queue the task to the semaphore waiting queue.
FLGDEQUEUE	FLGDEQUEUE	Release the task from the eventflag waiting queue.
FLGENQUEUE	FLGENQUEUE	Queue the task to the eventflag waiting queue.
DTQDEQUEUE	DTQDEQUEUE	Release the task from the data queue waiting queue.
DTQRCVENQUEUE	DTQENQUEUE	Queue the task to the data queue waiting queue in receiving data.
DTQSNDENQUEUE	DTQENQUEUE	Queue the task to the data queue waiting queue in sending data.
DTQDATA	–	Data queue data is obtained.
RDYENQUEUE	PRIENQUEUE	Queue the task to the ready queue.
HIGHEST	PRIHIGHEST	Obtain the highest task priority.
END	–	To prepare to exit the RTOS hardware.
ENDSWITCH	–	Set the task switch request.



```
if (tskid == TSK_SELF) {
    tskid = CurrentTaskId;
}
tcb = &tcbs[tskid];

/* check taskid */
if ((tskid < 0) || (tskid >= TMAX_TSKID)) {
    return(E_ID);
}

/* not registered */
if (taskcontexts[tskid].stackpt == NULL) {
    return(E_NOEXS);
}

/* check if this task is dormant */
if (tcb->tskstat != TTS_DMT) {

    /* act_tsk queue over */
    if (tcb->actcnt < TMAX_ACTCNT)
        tcb->actcnt += 1;
    } else {
        return(E_QOVR);
    }
}
rdyenqueue(tskid, tcbs[tskid].priority);

/* Get the highest task id */
highesttask = rdyhighesttask();

tcb->tskstat = TTS_RDY;

/* Task switch if necessary */
    ...

```

Figure 4.6: Hardware implemented part of RTOS system call (act_tsk).

act_tsk (excerpt)

```
RTOSPARAM1 = tskid;
RTOSSYSCALL = CODE_ACT_TSK;

/* When RTOS HW finish, bit:31 is on */
while (((rtosreturn = PRIHIGHEST)
          & 0x80000000) == 0);

highesttask = ((rtosreturn >> 16) & 0xff);
errorcode = (rtosreturn & 0xff);

if (errorcode == 0) {

  if (highesttask == 0) {

    /* No task switch */
    return(E_OK);
  } else {

    /* Task switch */
    RunTask(highesttask);
    return(E_OK);
  }
} else {

    /* Error case */
  return((ER)errorcode);
}
```

Figure 4.7: Example of system call software part (act_tsk).

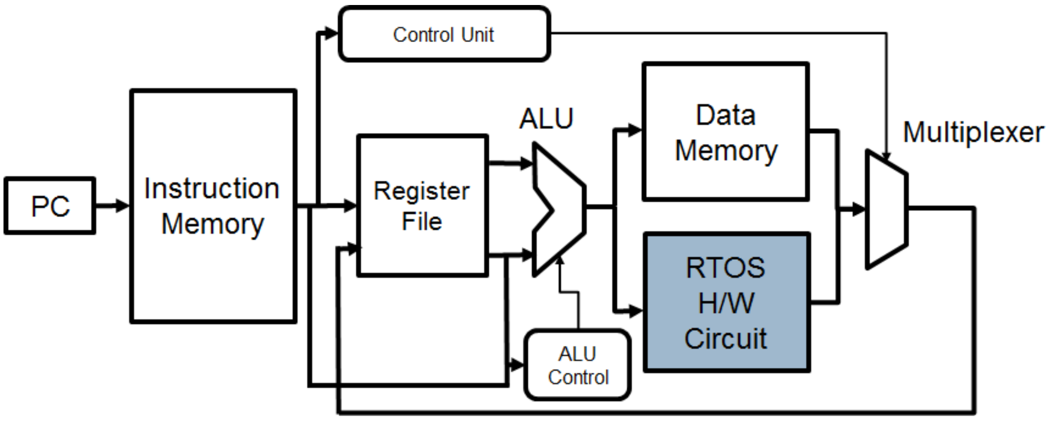


Figure 4.8: Processor structure with RTOS hardware.

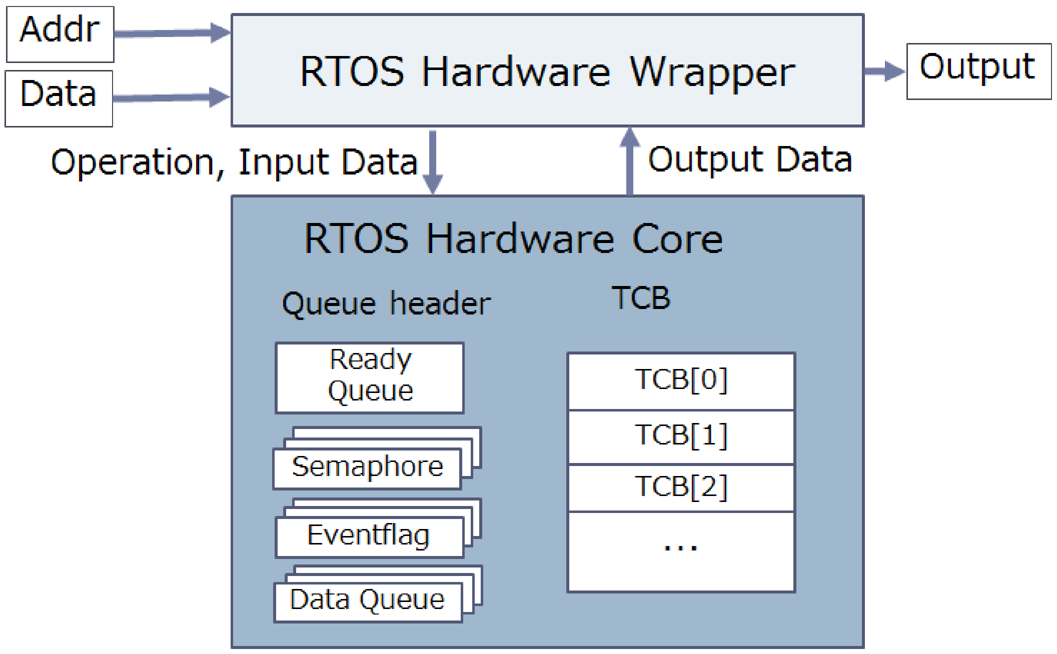


Figure 4.9: Structure of RTOS hardware.

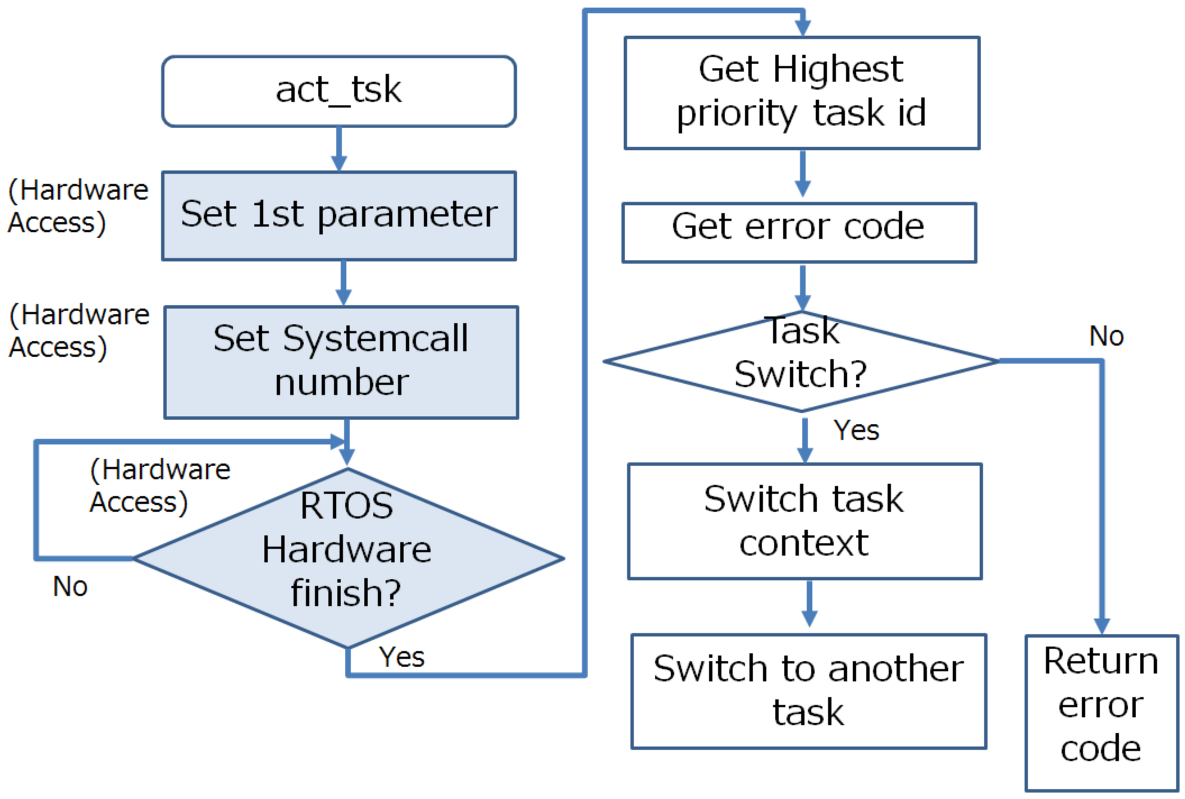


Figure 4.10: System call software flow.

Table 4.7: Operations for RTOS hardware core.

Operation	Description
READYENQUEUE	Enqueue a TCB to a ready queue
READYDEQUEUE	Dequeue a TCB from a ready queue
PRIHIGHEST	Return a task ID of the highest priority
PRICHG	Change priority of a task
TASKSTATUS	Return a task status
SEMHEAD	Return a task ID of the top of a semaphore waiting queue.
SEMENQUEUE	Enqueue a TCB to a semaphore waiting queue
SEMDEQUEUE	Dequeue a TCB from a semaphore waiting queue
FLGHEAD	Return a task ID of the top of an eventflag waiting queue
FLGENQUEUE	Enqueue a TCB to an eventflag waiting queue
FLGDEQUEUE	Dequeue a TCB from an eventflag waiting queue
DTQHEAD	Return a task ID of the top of a data queue
DTQENQUEUE	Enqueue a TCB to a data queue
DTQDEQUEUE	Dequeue a TCB from a data queue

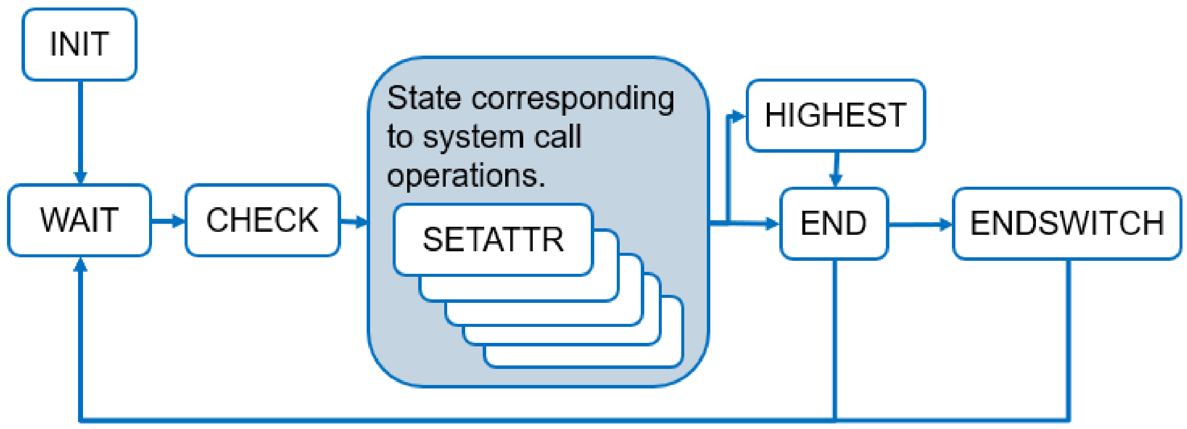


Figure 4.11: State transition.

4.4.4 RTOS Hardware Core

RTOS Hardware Core consists of TCBs and queue headers for RTOS resources which have a queue structure, that is a ready queue, semaphore waiting queue, eventflag waiting queue and data queue. RTOS Hardware Wrapper requests operations described in Table 4.7 to perform an RTOS core operation.

A module to control a task is called TCB (Task Control Block). The same number of TCBs as tasks are implemented in RTOS Hardware Core. The number of TCBs are decided when the adaptation tool works. Figure 4.12 is a TCB structure showing registers, input and output signals of a TCB with omitting the clock signal and the reset signal. The role of each signal is described below.

OPERATION_IN

This signal delivers the operation described in Table 4.7. According to this signal, each TCB sets the register values and decides output signals. For example, when READYENQUEUE signal is received, state, next_id and next_pri registers are set properly if the TCB is the target of the ready queue operation.

WE_IN

This signal delivers a write enable signal for TCB registers. Only when this signal is set, each TCB sets its own registers.

ID_IN

This signal delivers a task identifier for an operation. According to this signal, each TCB decides whether this operation would be for this TCB or not.

PRI_IN

This signal delivers a task priority for a priority queue operation. The target TCB operates the registers pri, state, next_id and next_pri according to this signal.

NEXT_ID_IN

This signal is input data for the register next_id. According to this input, the target TCB sets the register next_id.

NEXT_PRI_IN

This signal is input data for the register next_pri. According to this input, the target TCB sets the register next_pri.

NEXT_ID_OUT

This signal is output data indicating the task ID, which is used to update next_id in a relevant TCB when a queue operation is performed.

NEXT_PRI_OUT

This signal is output data indicating the task priority, which is used to update next_pri in a relevant TCB when a queue operation is performed.

Only a TCB which becomes prior to the inserted TCB or a TCB which is being deleted generates valid values for outputs of NEXT_ID_OUT and NEXT_PRI_OUT, while the other TCBs output zeros. OR gates select the valid values and transmit them to all the TCBs through NEXT_ID_IN and NEXT_PRI_IN inputs.

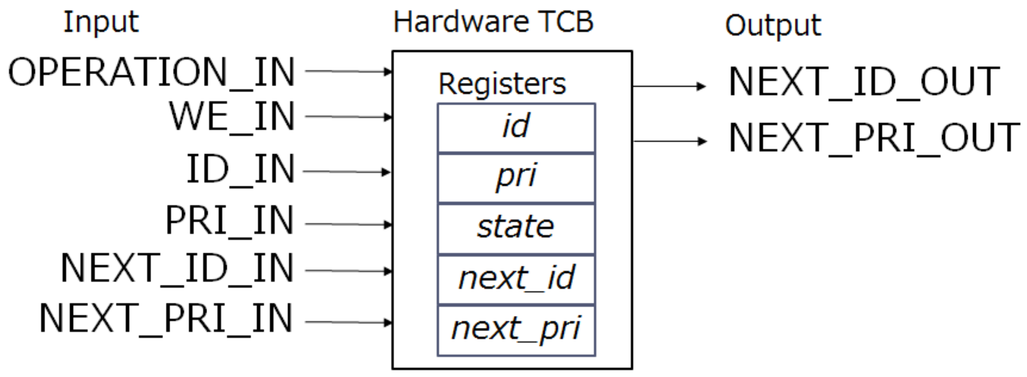


Figure 4.12: TCB structure.

What is stored in the registers in Fig.4.12 is described below.

id stores task ID of this TCB.

pri stores a priority of this TCB.

state stores a state of this TCB.

next_id has a task ID of the next task in a queue.

next_pri indicates a task priority of the next task in a queue.

How a queue operation is performed with these signals and registers is described as follows. We will show the case when tasks of *id* = 1, 2 and 4 are queued in a priority queue, and a task of *id* = 3 is designated to be enqueued in the queue. Before queuing operation, each register and signal in the TCBs are as Table 4.8. The task of *id* = 1 is the top of the queue, and the task of *id* = 2 is the next of the task of *id* = 1 since the task of *id* = 1 has *next_id* = 2. As the task of *id* = 4 is the last task in the queue, *next_id* = -1 and *next_pri* = 31, the maximum priority value in this configuration. In *state* row, priQ means this TCB is queued in a priority queue, and Not in Q means this TCB is not queued in any queue. X means this signal does not matter.

When the input signal of OPERATION_IN is READYENQUEUE, each TCB behaves as follows.

- When the input signal of PRI_IN is greater than or equal to *pri* and PRI_IN is less than *next_pri*, *next_id* and *next_pri* registers are set to the values of the input signals of ID_IN and PRI_IN, respectively. NEXT_ID_OUT and NEXT_PRI_OUT output the old values of *next_id* and *next_pri*.
- When the input signal of ID_IN is the same as *id* in the TCB, *next_id* and *next_pri* registers are set to the values of the input signals of NEXT_ID_IN and NEXT_PRI_IN, respectively, and *state* is changed to priQ.

Table 4.8: TCB registers and I/O (Before).

TCB		1	2	3	4
Registers	*id*	1	2	3	4
	pri	1	3	2	5
	state	priQ	priQ	Not in Q	priQ
	next_id	2	4	0	-1
	next_pri	3	5	0	31
Input	ID_IN	X	X	X	X
	PRI_IN	X	X	X	X
	NEXT_ID_IN	X	X	X	X
	NEXT_PRI_IN	X	X	X	X
Output	NEXT_ID_OUT	0	0	0	0
	NEXT_PRI_OUT	0	0	0	0

Table 4.9: TCB registers and I/O (On queuing).

TCB		1	2	3	4
Registers	*id*	1	2	3	4
	pri	1	3	2	5
	state	priQ	priQ	Not in Q → priQ	priQ
	next_id	2 → 3	4	0 → 2	-1
	next_pri	3 → 2	5	0 → 3	31
Input	ID_IN	3	3	3	3
	PRI_IN	2	2	2	2
	NEXT_ID_IN	2	2	2	2
	NEXT_PRI_IN	3	3	3	3
Output	NEXT_ID_OUT	2	0	0	0
	NEXT_PRI_OUT	3	0	0	0

Table 4.9 shows the register values, input and output signals during the enqueuing operation. *next_id* and *next_pri* in TCB of a task of *id* = 1 are set to 3 and 2, respectively, since PRI_IN = 2 is greater than *pri* in TCB of a task of *id* = 1 and lower than *next_pri*. As ID_IN = 3 and PRI_IN = 2 are input, according to the aforementioned behavior, *next_id* and *next_pri* in TCB of a task of *id* = 3 are set to 2 and 3, respectively.

As we can see from the explanation above, since each TCB can work in parallel, the queue search is not necessary while software-only RTOS needs the queue search.

4.5 Development Process

In this section, we describe the procedure to configure RTOS by analyzing an application and selecting functions which are actually used. The inputs of the environment are an RTOS system configuration file and application program files, and the final outputs are the source code of RTOS Hardware Core written with HDL, a file of hardware definition which is a set of directives to create the adaptive hardware RTOS, a header file written with C language to create an adaptive CPU core and a software program object to initialize the block memory on the FPGA, which is called coe file. The format of coe file is defined by Xilinx in Ref.[77]. This procedure consists of the following steps.

(Step 1) To generate an application specific RTOS kernel, static APIs in the RTOS system configuration file are analyzed. (Section 4.5.1)

(Step 2) Application programs are analyzed. (Section 4.5.2)

(Step 3) A header file and a hardware definition file are output. (Section 4.5.3)

(Step 4) Source codes for RTOS resources creation are created. (Section 4.5.4)

(Step 5) RTOS Hardware core with necessary resources is created. (Section 4.5.5)

(Step 6) Software object file is generated. (Section 4.5.6)

(Step 7) Adaptive processor core is generated. (Section 4.5.7)

(Step 8) Software program object file is created. (Section 4.5.8)

(Step 9) A bitmap file for an FPGA is generated with the hardware synthesis and implementation tool. (Section 4.5.9)

These steps in detail are explained in the following sections.

4.5.1 Parsing RTOS configuration file

The format of the configuration file follows μITRON4.0 specification in Ref.[7], which is a collection of static system call APIs. Figure 4.13 shows an example of a configuration file. In this phase, a list of IDs used in creating the specified kernel objects and the other attributes (e.g. fifo or priority order, conditions for eventflags, etc.) are extracted. This information is used both in the standard μITRON configuration process and in the proposed adaptation procedure. We explain the procedure in detail in the following steps. On the first step, the adaptation tool parses the configuration file to obtain the number of RTOS resources and the attributes such as the number of tasks, a list of task ids and task priorities to create, the number of semaphores, a list of semaphore ids to create, the number of eventflags, a list of eventflag ids to create, the number of data queues, a list of data queue ids to create, whether the waiting queue is task priority order or fifo order, and the attribute of eventflags.

4.5.2 Analyzing Application Program

On the next step, application programs are analyzed to determine which system call is used and check parameters in calling system calls. Before checking the application pragrams, firstly these programs are passed to C language preprocessor to expand macro definitions.

4.5.3 Outputting Directives

According to the number of resources obtained on the previous steps, macro definitions as directives for generating an adaptive RTOS are output to a header file for a software-only RTOS and a hardware definition file for RTOS hardware. Output definitions are as follows.

- Number of Resources
  The number of resources such as tasks or semaphores is output as a macro definition.
- Initial task priority list
  A list of initial task priorities is output to be used in task creation.
- Order of Waiting Queue
  In the result of analyzing the configuration file, the order in waiting queue, priority order of fifo order, is output.
- Attribute of Eventflag
  Whether TA_CLR attribute in creating an eventflag is used is checked since operation for clearing an eventflag can be omitted if TA_CLR attribute is not used.
- Error Check
  Macro definitions to leave only necessary error checking are output according to the result of analyzing the configuration file and application program described in the previous steps.
- Module
  Which module (semaphore, eventflag and data queue) is used is output. If a certain module is not used, all of the code fragments related to the module can be omitted.
- System call
  Information on system calls which are actually used in the application program is output as macro definitions. While necessary system calls can be linked with the linker in generating software-only RTOS, these definitions are used in order to implement the hardware RTOS in the same manner.

With this operation, an RTOS hardware core HDL (Verilog) code and hardware definitions which are used for creating the adaptive RTOS hardware are generated. In addition, C language source codes for static resource creation and a header file to generate an adaptive software-only RTOS are created.

4.5.4 Static Resource Creation

Static Resource Creation (.c) is the C language source codes in which static RTOS resources are created according to the result of analyzing the RTOS configuration file. System calls for creating resources such as tasks and semaphores which are defined in the configuration file are invoked in this file, which is generated by the tool.

4.5.5 RTOS Hardware Core

In Fig.4.14, RTOS Hardware Core (.v) is the HDL source file for RTOS hardware core circuit described in Section 4.4.4. This HDL source file is created according to the RTOS system configuration file in the same manner as the static resource creation described in the previous step. With including Hardware Definition (.v) in the HDL source file, an adaptive RTOS hardware can be generated since the only necessary resources are implemented in the RTOS hardware.

4.5.6 Generating Software Object File

The adaptation tool invokes a C compiler to generate a software program object. To build a software-only RTOS kernel, Software Object file is generated from Header File and Static Resource Creation with linking the RTOS kernel software library as well as Application Program. To build a hardware RTOS, Software Object file is generated from Application Program, the source codes of RTOS Kernel (software part), Header File and Static Resource Creation. In both cases, the C compiler generates Software Object file in the elf format.

4.5.7 Generating Adaptive Processor Core

After Software Object file explained in the previous step is generated, with the method described in Section 3.2, an adaptive processor core is generated to implement the RTOS hardware circuit. An adaptive processor core is generated with a macro definition file created in the way described in Section 4.3.

4.5.8 Creating Software Program Object

In order to initialize memory cores for Xilinx FPGA, which we use as our evaluation target device, Software Object file has to be converted to coe format in Ref.[77]. In our environment, the Software Program Object file is converted from the elf format to coe format and the file is to be used for initializing the block memory in the FPGA.

4.5.9 Implementation

In the previous steps, RTOS Hardware Core, Hardware Definition, Header File, Static Resource Creation, Software Program Object and Macro Definition for Processor Core are generated in our environment.

Finally, to make the hardware RTOS work, a bitmap file for an FPGA needs to be generated with the hardware synthesis and implementation tool. RTOS Kernel H/W Wrapper, RTOS Hardware Core, Hardware Definition, the hardware HDL codes of Processor Core with Macro Definition, and Software Program Object are used to generate the bitmap file for an FPGA as Fig.4.14 illustrates. On the other hand, RTOS Kernel H/W Wrapper, RTOS Hardware Core and Hardware Definition are not necessary for generating a software-only RTOS. That is shown in Fig.4.15.

4.6 GUI

A GUI (Graphical User Interface) tool we developed to invoke the environment described in Section 4.5 is explained in this section. The tool is worked on Linux (Ubuntu) in Ref.[78] with Python in Ref.[79]. Python is a general-purpose programing language and has been commonly used in a wide range of applications. As Python supports several platforms, the tool can be easily ported to another environment. For implementing GUI, we used wxPython library in Ref.[80] for the API of GUI and referred the code of fileHunter in Ref.[81] for file operation. That helped to create GUI interface easily. The appearance of the tool is shown in Fig.4.16. The functions are illustrated below. The item numbers below correspond to the numbers in the figure.

(1) This is the field to select a directory in which source files of an application program is located. The directory path of the source files can be written.

(2) After the directory path of the source files is written, Select button is pushed to fix the written source code path.

(3) This is the file list field to show the list of the files in the selected directory.

(4) The Object Directory text box is to set a directory for creating the object files which are produced after analyzing and compiling the selected files.

(5) The RTOS system configuration file name in the source directory is indicated in this text box. In analyzing RTOS resources, the configuration file is used to generate an application adapted hardware.

(6) The Application files field is a list of the selected application program files. These files are used to compile the source codes and analyze which system calls are used in an application program.

(7) With Select RTOS radio button, we can select which RTOS is built, software-only RTOS kernel or using hardware RTOS. If Hardware is selected, the software part of the hardware RTOS is used for compiling the application program. If Software is selected, the software-only RTOS is generated for the RTOS kernel.

(8) In selecting an RTOS configuration file in the File list window, the selected file is input to the Config file text box when Config file button is pushed.

(9) In selecting a file in the File list window, the selected file is set to the Application files window when Application button is pushed. In File List window, multiple files can be selected at one time when files are selected with pushing the control key on the PC.

(10) In selecting a file in the Application files window, the selected file disappears when Delete button is pushed. In Application files window, multiple files can be selected at on time when files are selected with pushing the control key on the PC. In this case, all selected files disappear in the Application files window.

(11) In selecting a file in the Application files window, the displayed order of the selected file moves up when Up button is pushed.

(12) In selecting a file in the Application files window, the displayed order of the selected file moves down when Down button is pushed.

(13) After the RTOS configuration file and application program files are selected, Build button is to be pushed to generate RTOS Hardware Core, Hardware Definition, Macro Definition and Software Program Object described in Section 4.4.

(14) In pushing Clean button, created object files are cleaned up.

(15) In pushing Exit button, the tool is closed.

4.7 Summary of this Chapter

In this chapter, we proposed a framework to generate an application adapted hardware RTOS and software-only RTOS. We adopted μITRON4.0 for the RTOS specification and showed the method to generate the application adapted RTOS kernel. "Removing Unnecessary Codes Caused by Fixed Attributes" and "Removing Unnecessary Codes Caused by the Way of Calling" are explained for the method for generating an application adapted RTOS. We developed the hardware RTOS to reduce the overhead of the software part in an RTOS. The hardware RTOS we implemented consists of the RTOS Hardware Wrapper and RTOS Hardware Core. The RTOS Hardware Wrapper works as a state machine and the RTOS Hardware Core manages queues and task control blocks. We showed the way to reduce hardware resources with the method of removing unnecessary codes. In order to generate the adaptive hardware and software part of RTOS, we described the development environment and GUI tool. The proposed method does not need to update the RTOS specification to generate an application adaptive RTOS in a fully automatic manner.

RTOS Configuration File

```
INCLUDE("\"main.h\"");
INCLUDE("\"rtosconfig.h\"");
INCLUDE("\"kernel_id.h\"");
INCLUDE("\"itron.h\"");
INCLUDE("\"test.h\"");

/* tskid, tskatr, exinf, task, itskpri, stksz, stk */
CRE_TSK ( TASK_ID1, { TASK_ATR1, EXINF_1, task1, TASK_PRI1, STACKSIZE1,
    &stacktask1[STACKSIZE1-1]} );
CRE_TSK ( TASK_ID2, { TASK_ATR2, EXINF_2, task2, TASK_PRI2, STACKSIZE2,
    &stacktask2[STACKSIZE2-1] } );
CRE_TSK ( TASK_ID3, { TASK_ATR3, EXINF_3, task3, TASK_PRI3, STACKSIZE3,
    &stacktask3[STACKSIZE3-1] } );
CRE_TSK ( TASK_ID4, { TASK_ATR4, EXINF_4, task4, TASK_PRI4, STACKSIZE4,
    &stacktask4[STACKSIZE4-1] } );
CRE_TSK ( TASK_ID5, { TASK_ATR5, EXINF_5, task5, TASK_PRI5, STACKSIZE5,
    &stacktask5[STACKSIZE5-1] });

/* semid, sematr, isemcnt, maxsem */
CRE_SEM ( SEM_ID1, { SEM_ATR1, SEM_ISEMCNT1, SEM_MAXSEM1 } );
CRE_SEM ( SEM_ID2, { SEM_ATR2, SEM_ISEMCNT2, SEM_MAXSEM2 } );
CRE_SEM ( SEM_ID3, { SEM_ATR3, SEM_ISEMCNT3, SEM_MAXSEM3 } );
CRE_SEM ( SEM_ID4, { SEM_ATR4, SEM_ISEMCNT4, SEM_MAXSEM4 } );

/* flgid, flgatr, iflgptn */
CRE_FLG ( FLG_ID1, { FLG_ATR1, FLG_PTN1 } );
CRE_FLG ( FLG_ID2, { FLG_ATR2, FLG_PTN2 } );
CRE_FLG ( FLG_ID3, { FLG_ATR3, FLG_PTN3 } );

/* dtqid, dtqatr, dtqcnt, dtqaddr */
CRE_DTQ ( DTQ_ID1, { DTQ_ATR1, DTQ_CNT1, DTQ_ADDR1 } );
CRE_DTQ ( DTQ_ID2, { DTQ_ATR2, DTQ_CNT2, DTQ_ADDR2 } );
CRE_DTQ ( DTQ_ID3, { DTQ_ATR3, DTQ_CNT3, DTQ_ADDR3 } );
```

Figure 4.13: Example of a configuration file.

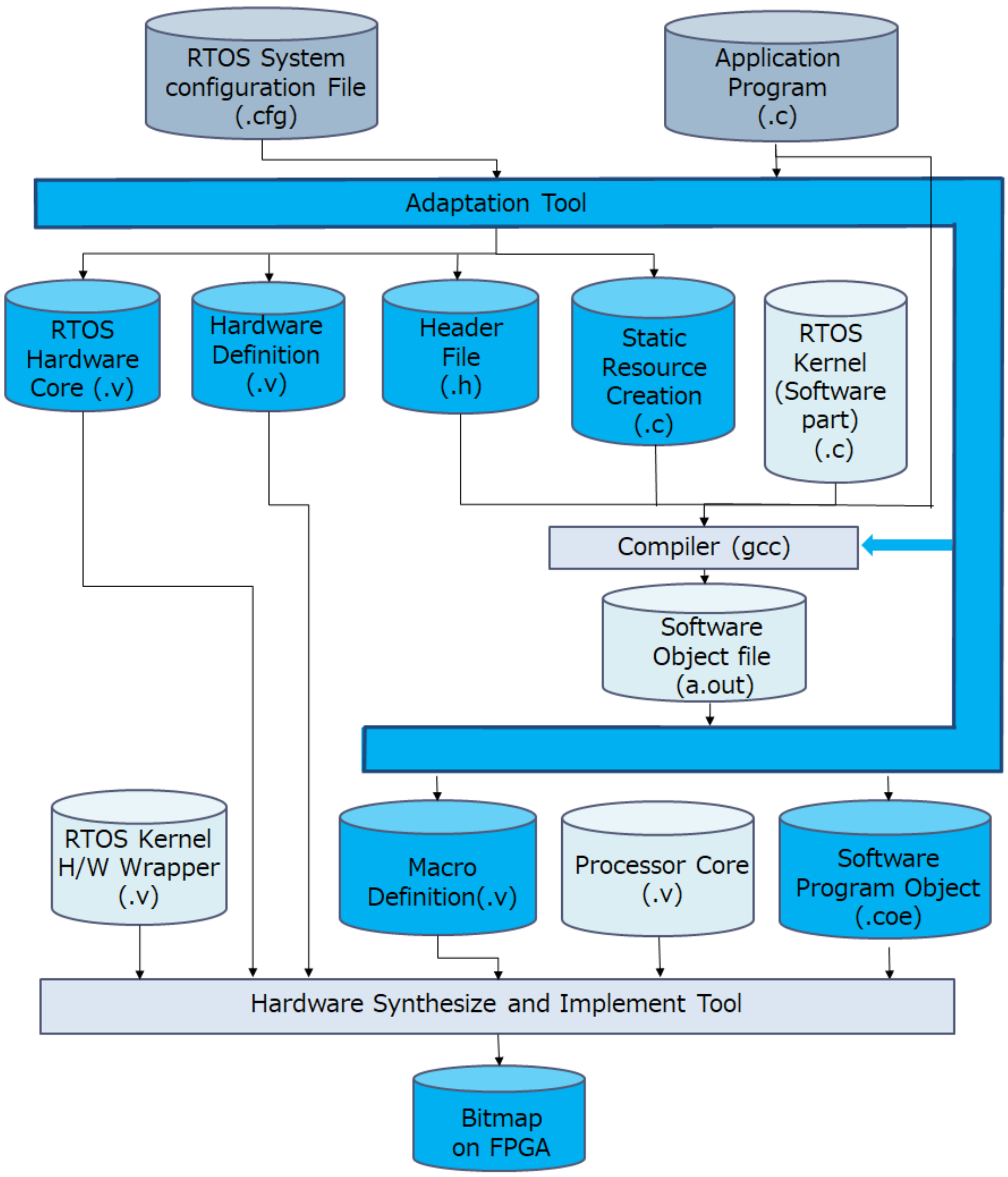
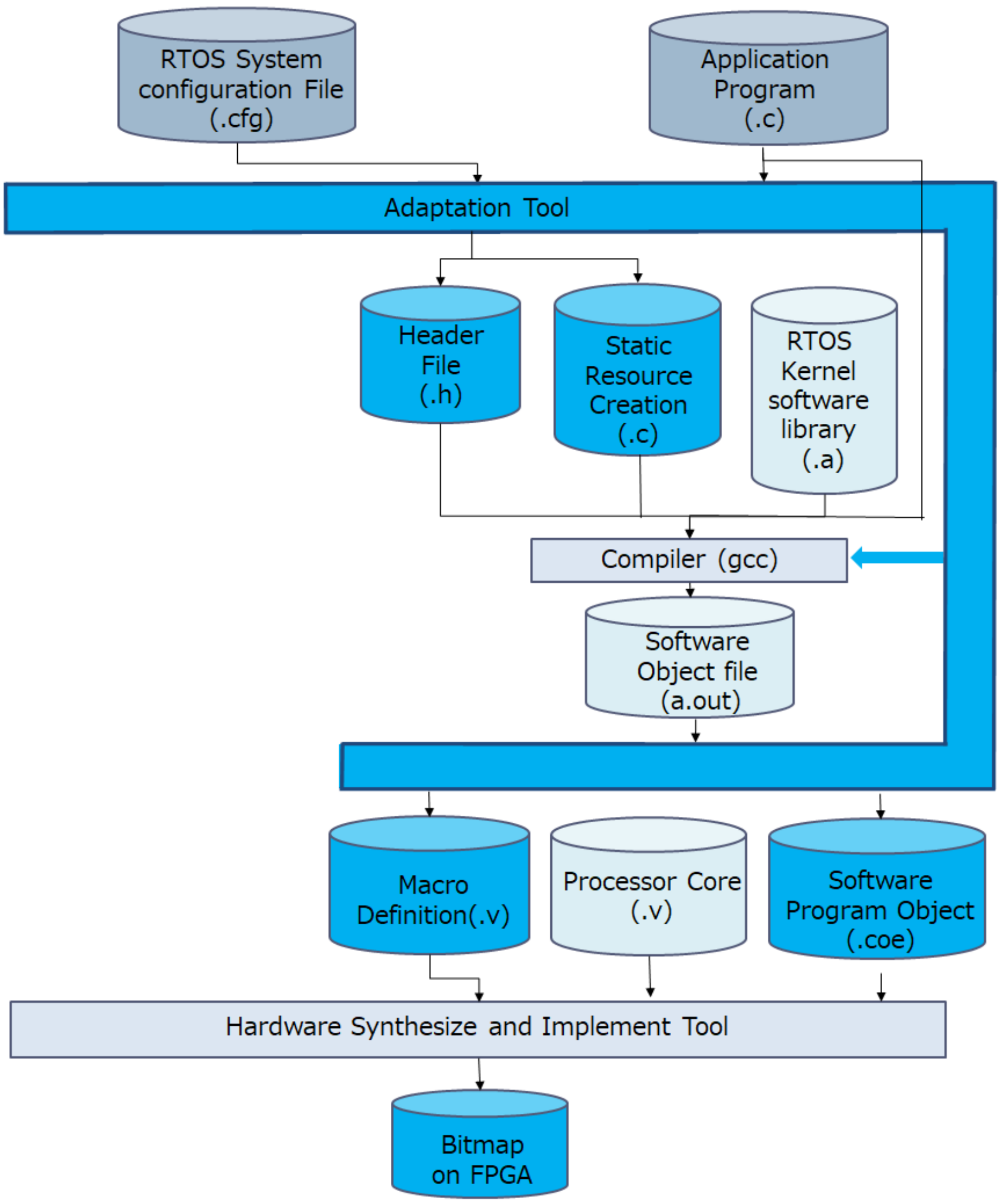
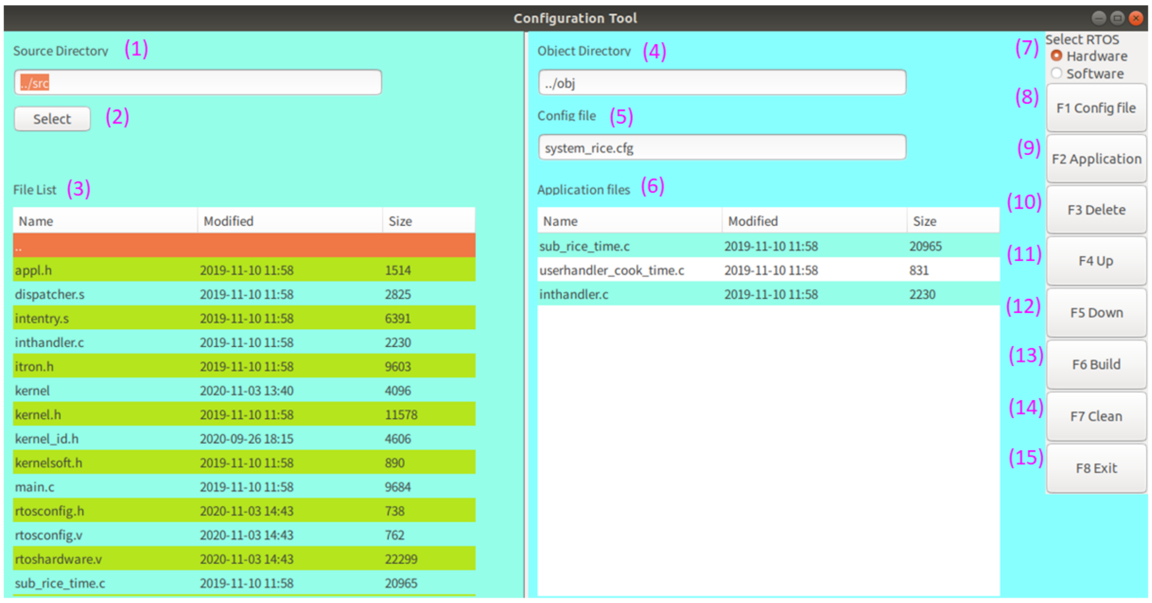

RTOS System configuration File (.cfg)
Application Program (.c)
Adaptation Tool
RTOS Hardware Core (.v)
Hardware Definition (.v)
Header File (.h)
Static Resource Creation (.c)
RTOS Kernel (Software part) (.c)
Compiler (gcc)
Software Object file (a.out)
RTOS Kernel H/W Wrapper (.v)
Macro Definition(.v)
Processor Core (.v)
Software Program Object (.coe)
Hardware Synthesize and Implement Tool
Bitmap on FPGA

Figure 4.14: Development environment.



RTOS System configuration File (.cfg)
Application Program (.c)
Adaptation Tool
Header File (.h)
Static Resource Creation (.c)
RTOS Kernel software library (.a)
Compiler (gcc)
Software Object file (a.out)
Macro Definition(.v)
Processor Core (.v)
Software Program Object (.coe)
Hardware Synthesize and Implement Tool
Bitmap on FPGA

Figure 4.15: Development environment for software RTOS.



Source Directory: ../src

Object Directory: ../obj

Config file: system_rice.cfg

Select RTOS: [✓ Hardware] Software

File List

Name	Modified	Size
..		
appl.h	2019-11-10 11:58	1514
dispatcher.s	2019-11-10 11:58	2825
intentry.s	2019-11-10 11:58	6391
inthandler.c	2019-11-10 11:58	2230
itron.h	2019-11-10 11:58	9603
kernel	2020-11-03 13:40	4096
kernel.h	2019-11-10 11:58	11578
kernel_id.h	2020-09-26 18:15	4606
kernelsoft.h	2019-11-10 11:58	890
main.c	2019-11-10 11:58	9684
rtosconfig.h	2020-11-03 14:43	738
rtosconfig.v	2020-11-03 14:43	762
rtoshardware.v	2020-11-03 14:43	22299
sub_rice_time.c	2019-11-10 11:58	20965

Application files

Name	Modified	Size
sub_rice_time.c	2019-11-10 11:58	20965
userhandler_cook_time.c	2019-11-10 11:58	831
inthandler.c	2019-11-10 11:58	2230

Figure 4.16: GUI tool.

Chapter 5

Evaluation

5.1 Overview

This chapter presents the effect of the method explained in Chapter 3 and 4. To evaluate the proposal, we implemented the processor core and the hardware RTOS in an FPGA so that the improvement of execution time of system calls and reduction of software resources and hardware resources against software-only RTOS without adaptation can be seen with several application programs.

The architecture of the processor core is the one described in Section 3.1 and RTOS hardware is described in Section 4.4, which are implemented in an FPGA, Xilinx Artix-7-FPGA (XC7A35T-ICPG236C) on the evaluation board of Digilent Basys 3 FPGA Board with the Xilinx development tool, Vivado® Design Suite 2018.3. The Basys3 board is shown in Fig.5.1.

The processor core we implemented runs at 68.0MHz and executes MIPS32 instruction set in Ref.[58]. GCC 4.3.3 is used for the compiler. The RTOS kernel is configured to be either a software-only kernel described in Section 4.2 or a kernel with RTOS hardware in Section 4.4. Which RTOS kernel is used is selectable when the system is configured. We evaluated the effects of application adaptation described in Section 4.3 for the software-only RTOS kernel and RTOS hardware implementation described in Section 4.4 comparing with the software-only RTOS kernel without application adaptation.

5.2 FPGA Resources

Table 5.1, 5.2, 5.3, 5.4 and 5.5 illustrate the number of FPGA resources occupied by the processor core and RTOS hardware, and worst negative slack (WNS), Fmax and Power. Percentages in the parentheses indicate the rates to the whole resources of the devices we used for the evaluation.

LUT, LUTRAM, FF, BRAM and DSP are configurable resources in an FPGA and the maximum number available is shown in the column of "Available" in Table 5.5. LUT is a six-input look-up table implemented in Artix-7-FPGA, Xilinx FPGA in Ref.[82]. LUTRAM indicates the number of used resources of Distributed RAM. FF shows the number of used flip-flops. BRAM is Block RAM, which is used to store instructions and read only data for the processor core described in Section 3.1. DSP is used when the

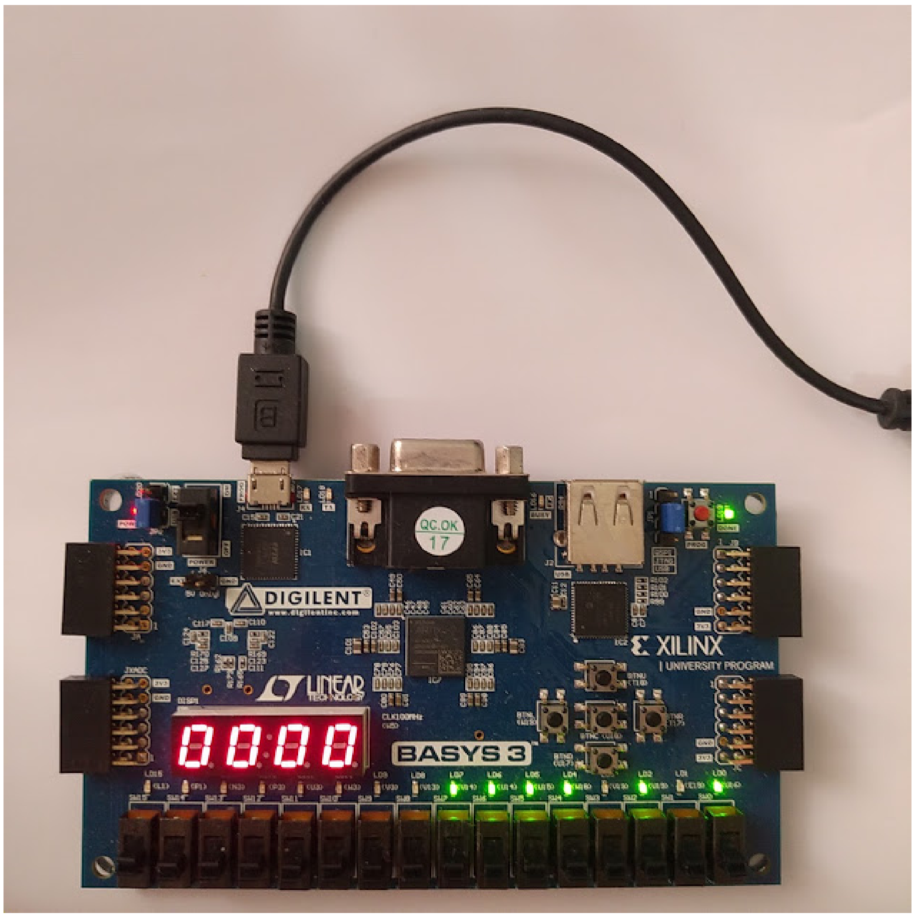


Figure 5.1: The Basys3 board.

processor core needs multipliers. For the detailed feature of FPGA resources, the user guide in Ref.[82] can be referred.

The clock rate of the processor (CLK) is 68.0MHz, WNS stands for Worst Negative Slack, the value of which is shown after implementation and Fmax is calculated by $Fmax = 1000/((1000/(1000/68.0) - WNS))$ in Ref.[65].

We evaluated programs as follows: "sem02" (for semaphore test) and "flg02" (for eventflag test) are from the μITRON4.0 TOPPERS kernel test suites in Ref.[83], "dtq" (for data queue test) is our original program, "semflgdtq" is a combination of sem02, flg02 and dtq, "semflg" is a combination of sem02 and flg02, and two programs, "Cooker" and "Pot", are from the literature [84], which are RTOS application programs for a rice cooker and an electric pot, respectively. (Since the two test suites programs include all error cases, intentional error checking codes among them are removed from the programs to evaluate the effect of adaptation.)

In the row of CPU Adaptive in Table 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, "Yes" shows CPU circuit is adapted with the method described in Section 3.4 and in the row of RTOS Adaptive, "Yes" shows RTOS circuit is adapted with the method described in Section 4.4. Table 5.1 is the result of a program of semflgdtq, from which the effect of CPU adaptation and RTOS adaptation can be seen. Table 5.2 shows the result of applying CPU and RTOS adaptation to semflg and sem02. Table 5.3 shows the result of applying

Table 5.1: FPGA resources (semflgdtq).

CPU Adaptive RTOS Adaptive	No No	Yes No	No Yes	Yes Yes
# of Task	5	5	5	5
# of Semaphore	4	4	4	4
# of Eventflag	3	3	3	3
# of Data Queue	3	3	3	3
LUT	11,546 (55.51%)	6,706 (32.24%)	11,226 (53.97%)	6,430 (30.91%)
LUTRAM	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)
FF	3,528 (8.48%)	3,460 (8.32%)	3,517 (8.45%)	3,448 (8.29%)
BRAM	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)
DSP	8 (8.89%)	–	8 (8.89%)	–
WNS (ns)	-0.507	0.044	-0.222	0.002
Fmax (MHz)	65.734	68.204	66.989	68.009
Power (W)	0.372	0.247	0.358	0.236

CPU and RTOS adaptation to flg02 and dtq. Table 5.4 shows the result of applying CPU and RTOS adaptation to Cooker and Pot. Table 5.5 shows the result of w/o RTOS Hardware and availability of the FPGA resources for comparison with other cases. The column of w/o RTOS Hardware is the resource usage of only a processor core adapted to semflgdtq. Effects of fine-grained configuration for RTOS kernel can be confirmed even with simple programs such as the ones mentioned above.

From Table 5.1, we can see that the adaptation achieves a reduction in LUT and FF by 44.3% (from 11,546 to 6,430) and 2.27% (from 3,528 to 3,448), respectively, comparing adaptive and no adaptive configurations for both CPU and RTOS in semflgdtq. In addition, we can see that DSP resources can be deleted when the CPU is adapted. From Table 5.2, since semflg does not include a data queue system call, it can be seen that LUT and FF are reduced by 16.9% (from 6,430 to 5,343) and 22.9% (from 3,448 to 2,658), respectively, compared to semflgdtq with adaptation, due to the elimination of data queue resources. The resources of sem02 show that we can further decrease LUT and FF by 23.7% (from 6,430 to 4,905) and 32.4% (from 3,448 to 2,330), respectively, since it does not have the resources of eventflag. Similarly, we can see the results that the resources of flg02 and dtq are reduced due to the reduction of unused system call resources in Table 5.3. For flg02, LUT and FF can be reduced by 14.1% (from 6,430 to 5,523) and 16.6% (from 3,448 to 2,875), respectively, compared to semflgdtq. For dtq, LUT and FF can be reduced by 10.8% (from 6,430 to 5,733) and 12.4% (from 3,448 to 3,021), respectively, compared to semflgdtq. In Table 5.4, since Cooker employs only the necessary number of resources, LUT and FF can be reduced by 4.9% (from 6,430 to 6,115) and 5.8% (from 3,448 to 3,249), respectively, compared to semflgdtq which uses more semaphores, eventflags and data queues. The same trend can be seen in the results for Pot.

Table 5.2: FPGA resources (semflg, sem02).

Name	semflg	semflg No Adaptation	sem02 [83] w/o error	sem02 [83] w/o error No Adaptation
CPU Adaptive	Yes	No	Yes	No
RTOS Adaptive	Yes	No	Yes	No
# of Task	5	5	5	5
# of Semaphore	4	4	4	4
# of Eventflag	3	3	0	0
# of Data Queue	0	0	0	0
LUT	5,343 (25.69%)	11,475 (55.17%)	4,905 (23.58%)	11,369 (54.66%)
LUTRAM	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)
FF	2,658 (6.39%)	3,496 (8.40%)	2,330 (5.60%)	3,464 (8.33%)
BRAM	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)
DSP	–	8 (8.89%)	–	8 (8.89%)
WNS (ns)	0.322	-0.132	0.008	-0.029
Fmax (MHz)	69.522	67.395	68.037	67.866
Power (W)	0.231	0.37	0.229	0.369

5.3 Execution Time

Table 5.7 illustrates comparison of each system call execution time among the cases of hardware RTOS (w/ Hardware), adaptive software-only RTOS (Software w/ Adaptive), and software-only RTOS without adaptation (Software w/o Adaptive).

Since execution of system calls can involve task switching, the table includes execution times in both cases with task switching and without it. In the column of Task Switch, “No” indicates that the system call is executed and completed without task switching. Meanwhile, “Yes” corresponds to the situation where the system call execution includes task switching.

The number of clock cycles taken for executing a system call is counted and the execution time is obtained by converting the number of clock cycles to the duration of the system call execution (μsec), considering the processor core’s running clock frequency of 68.0MHz.

The column of w/ Hardware shows the system call execution time (μsec) of adaptive RTOS hardware with the ratio to Software w/o Adaptive case. The column of Software w/ Adaptive shows that the system call execution time (μsec) of adaptive software-only RTOS. In this case, each RTOS system call is executed on the processor without using RTOS hardware, and the ratio of Software w/ Adaptive to Software w/o Adaptive is shown. The column of Software w/o Adaptive illustrates the system call execution time (μsec) when RTOS kernel is not adapted to the application program.

From Table 5.7, it can be seen that, on average, w/ Hardware can reduce the execution time to 71.8% and 40.2% for w/o Task Switch and w/ Task Switch, respectively, comparing to software-only RTOS without adaptation. That means 1.39 times faster execution in the case of w/o Task Switch and 2.49 times faster execution in the case of w/ Task Switch. For the case of Software w/ Adaptive, it is reduced to 82.9% and 95.0%, respectively.

Table 5.3: FPGA resources (flg02, dtq).

Name	flg02 [83] w/o error	flg02 [83] w/o error No Adaptation	dtq	dtq No Adaptation
CPU Adaptive	Yes	No	Yes	No
RTOS Adaptive	Yes	No	Yes	No
# of Task	4	4	3	3
# of Semaphore	0	0	0	0
# of Eventflag	3	3	0	0
# of Data Queue	0	0	3	3
LUT	5,523 (26.55%)	11,390 (54.76%)	5,733 (27.56%)	11,540 (55.48%)
LUTRAM	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)
FF	2,875 (6.91%)	3,496 (8.40%)	3,021 (7.26%)	3,496 (8.40%)
BRAM	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)
DSP	–	8 (8.89%)	–	8 (8.89%)
WNS (ns)	0.321	-0.494	0.163	-0.135
Fmax (MHz)	69.517	65.790	68.762	67.381
Power (W)	0.232	0.37	0.241	0.37

That means 1.21 times faster execution in the case of w/o Task Switch and 1.05 times faster execution in the case of w/ Task Switch. In the case of sem02 pol_sem and wai_sem without task switching, the execution time of w/ Hardware is longer than that of Software w/ Adaptive, the reason of which is that the semaphore count is simply implemented in this operation and there is no queue operation or priority search involved. pol_flg and wai_flg without task switching have the same reason.

These results show that, except for a few cases without task switching, w/ Hardware makes the system call execution time faster than Software w/ Adaptive. Especially, the system call execution time with task switching is much reduced due to the reduction of queue operation time with hardware. On the other hand, for Software w/ Adaptive, while we can see the effect of removing unnecessary codes caused by fixed attributes and the way of calling, the rate of the reduction of the execution time is lower in the case of w/ Task Switch since it does not accelerate queue operations.

Since the adaptation method explained in this dissertation never increases the execution time of system calls in any case, it does not have a negative effect for real-time performance compared to the software-only RTOS without adaptation. In addition, queue operation time does not fluctuate in case of hardware RTOS, which contributes to reducing jitters and making the execution time constant. Table 5.8 shows execution time of each system call taking the maximum frequency (Fmax) into consideration. The result reveals that the performance deterioration which sometimes occurs with non-adapted RTOS hardware is reduced when the RTOS is adapted to an application even if RTOS hardware is implemented so that sufficient performance improvement can be obtained.

Table 5.4: FPGA resources (Cooker, Pot).

Name	Cooker [84]	Cooker [84] No Adaptation	Pot [84]	Pot [84] No Adaptation
CPU Adaptive	Yes	No	Yes	No
RTOS Adaptive	Yes	No	Yes	No
# of Task	4	4	3	3
# of Semaphore	0	0	0	0
# of Eventflag	1	1	2	2
# of Data Queue	1	1	1	1
LUT	6,115 (29.40%)	11,311 (54.38%)	5,934 (28.53%)	11,229 (53.99%)
LUTRAM	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)	1,536 (16.00%)
FF	3,249 (7.81%)	3,460 (8.32%)	3,173 (7.63%)	3,424 (8.23%)
BRAM	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)	7.50 (15.00%)
DSP	4 (4.44%)	8 (8.89%)	4 (4.44%)	8 (8.89%)
WNS (ns)	0.280	-0.074	0.343	0.054
Fmax (MHz)	69.320	67.660	69.624	68.251
Power (W)	0.239	0.365	0.237	0.365

5.4 RTOS Kernel Size

Table 5.9 shows the RTOS kernel sizes (bytes) of a software part of w/ Hardware, the software-only RTOS with adaptation (Software w/ Adaptive), and the software-only RTOS without adaptation (Software w/o Adaptative). For each program, the sizes of representative system calls, utility functions used in system calls (Kernel Soft), and other kernel parts (Others) are shown. Kernel Soft includes functions for queue management. Others consist of kernel initialization codes and other system calls. Percentages in parentheses indicate the rates to Software w/o Adaptive. For example, the size of the software part for sig_sem in sem02 is 59.4% of the size of Software w/o Adaptive and the total size of sem02 is reduced to 69.4%.

Kernel Soft size is 0 for w/ Hardware since the corresponding operation is implemented in hardware. In the case of dtq, Cooker and Pot, Kernel Soft size is not 100% in Software w/ Adaptive, the reason of which is the programs use only TA_TFIFO for the attribute of data queue while both of TA_TPRI and TA_TFIFO are implemented in the original (Software w/o Adaptive) Kernel Soft, where the code for the attribute of TA_TPRI is removed by adaptation. In w/ Hardware, the size of a software part of each system call is reduced to 65.6% at worst (pol_flg in flg02) and to 41.7% (fsnd_dtq in dtq, Cooker and Pot) at best. Additionally, total size of each program is reduced. The maximum reduction is 68.7% in Cooker. This large amount of reduction is achieved since most code fragments except for task switching are eliminated by adaption and Kernel Soft is fully reduced. In Software w/ Adaptive, we can see that the total size of RTOS kernel is reduced to 88.6% in the case of sem02, and the size of each system call is reduced to 89.8% at worst (prcv_dtq in dtq) and to 63.2% (pol_sem in sem02) at best.

Table 5.5: FPGA resources (w/o RTOS, Availability).

Name	w/o RTOS Hardware		Available
CPU Adaptive	Yes		–
RTOS Adaptive	–		–
# of Task	–		–
# of Semaphore	–		–
# of Eventflag	–		–
# of Data Queue	–		–
LUT	3,829	(18.41%)	20,800
LUTRAM	1,536	(16.00%)	9,600
FF	1,701	(4.09%)	41,600
BRAM	7.50	(15.00%)	50
DSP	–		90
WNS (ns)		0.268	–
Fmax (MHz)		69.262	–
Power (W)		0.228	–

5.5 Discussion

Table 5.6 shows the FPGA resource reduction of removing unnecessary codes caused by fixed attributes and the way of calling. The column "sem02 w/o error Priority only" is the case that all of attributes of semaphores are TA_TPRI, which specifies the semaphore's wait queue is the task priority order. Since the original sem02 program includes the fifo order, the configuration file is modified to the priority order. The column "sem02 w/o error" is referred from Table 5.2 for the case of the adapted RTOS hardware. The difference of LUT resources shows 1.1% reduction by the caused by fixed attributes. The last column is the case that all static errors are checked. As the result, LUT resources increase from 4,905 to 4,970, which means 1.3% of LUT resources can be reduced by the effect of adaptation for error checking. For Software RTOS (w/o Hardware), Table 5.10 shows the reduction of the software codes. The column "Software w/ Adaptive Priority only" is the case that all of the semaphores' wait queues are the task priority order. In this case, the reduction of Kernel Soft reflects the effect of the adaptation. We can see that 4.8% of Kernel Soft size is reduced. The columns, "Software w/ Adaptive" and "Software w/o Adaptive", are cited from Table 5.9. The difference of resources of system calls is derived from the way of calling. For example, 21.9% (from 512 to 400) of RTOS kernel size can be reduced for sig_sem system call.

While we can make this adaptation automatically, we show the performance of the automatic adaptation environment comparing to manual adaptation. Table 5.11 shows the comparison between automatic and manual adaptation for three application programs, semflgdtq, sem02 and Cooker, of which the details are explained in Section 5.2. The time of Manual is the duration of manual adaptation, which was performed by a person who saw this RTOS source code at the first time and had experience of the μITRON4.0 specification so that the person can read and understand the RTOS source code. The

Table 5.6: FPGA resources (sem02).

Name	sem02 [83] w/o error Priority only		sem02 [83] w/o error		sem02 [83] w/o error w/ error check	
CPU Adaptive	Yes		Yes		Yes	
RTOS Adaptive	Yes		Yes		Error check	
# of Task	5		5		5	
# of Semaphore	4		4		4	
# of Eventflag	0		0		0	
# of Data Queue	0		0		0	
LUT	4,850	(23.32%)	4,905	(23.58%)	4,970	(23.89%)
LUTRAM	1,536	(16.00%)	1,536	(16.00%)	1,536	(16.00%)
FF	2,318	(5.57%)	2,330	(5.60%)	2,354	(5.66%)
BRAM	7.50	(15.00%)	7.50	(15.00%)	7.50	(15.00%)
DSP	–		–		–	
WNS (ns)		0.178		0.008		0.008
Fmax (MHz)		68.833		68.037		68.037
Power (W)		0.228		0.229		0.227

time of Automatic is the executing time from the beginning of the adaptation process to the output of the tool, which works with Core i5-3230M 2.60GHz CPU on Ubuntu OS. The unit of time is seconds. This result reveals that the automatic generation is from 3,209 times (semflgdtq) to 2,458 times (Cooker) faster than manual generation.

5.6 Summary of this Chapter

We showed the effect of the proposed method in this chapter. It can be seen that FPGA resources, RTOS kernel execution time, and RTOS kernel size are reduced by the effect of the processor and RTOS adaptation. For example, 44.3% of LUT resources is reduced in the case that the processor and RTOS are adapted to application with respect to the case that both are not adapted for an application program of semflgdtq. For system call execution time, hardware RTOS makes 2.49 times faster in including task switch than software-only RTOS without adaptation, and 1.39 times faster without task switch. That is because the execution time of a queue operation and priority search can be reduced by the hardware RTOS. For the case of software-only RTOS, adaptation makes 1.21 times faster without task switch and 1.05 times faster in including task switch for the average execution time. The RTOS kernel size of a software part can be also reduced with hardware RTOS since most of software part in software-only RTOS can be removed due to hardware RTOS. In the case of adaptation of software-only RTOS, the RTOS kernel size is reduced owning to elimination of the unused code. The system call size of software part is also reduced to 41.7% at best with hardware RTOS and 63.2% at best with software-only RTOS with adaptation. The result of this chapter reveals that hardware sizes are reduced, the running frequency is improved, RTOS sizes are reduced, and system call execution times are improved, without updating the RTOS specification, in a fully automatic manner with the proposed method in this dissertation.

Table 5.7: RTOS kernel execution time (μsec).

Program	System Call	Task Switch	w/ Hardware	Software w/ Adaptive	Software w/o Adaptive
sem02	sig_sem	No	1.44 (66.2%)	1.78 (81.8%)	2.18
	sig_sem	Yes	3.15 (46.6%)	6.28 (92.8%)	6.76
	pol_sem	No	1.04 (77.7%)	0.94 (70.3%)	1.34
	wai_sem	No	1.26 (87.4%)	0.94 (65.3%)	1.44
	wai_sem	Yes	3.04 (37.4%)	7.62 (93.8%)	8.12
flg02	set_flg	No	1.57 (66.7%)	1.97 (83.8%)	2.35
	set_flg	Yes	3.35 (40.0%)	8.00 (95.4%)	8.38
	pol_flg	No	1.69 (85.8%)	1.59 (80.6%)	1.97
	wai_flg	No	1.87 (84.8%)	1.82 (82.7%)	2.21
	wai_flg	Yes	3.47 (36.3%)	9.18 (96.0%)	9.56
dtq	fsnd_dtq	No	1.54 (56.0%)	2.38 (86.6%)	2.75
	fsnd_dtq	Yes	3.21 (43.8%)	6.96 (95.0%)	7.32
	psnd_dtq	No	1.54 (59.5%)	2.32 (89.8%)	2.59
	psnd_dtq	Yes	3.19 (43.9%)	7.00 (96.4%)	7.26
	rcv_dtq	No	1.57 (76.8%)	1.78 (87.1%)	2.04
	rcv_dtq	Yes	3.04 (40.9%)	7.04 (94.7%)	7.44
	prcv_dtq	No	1.57 (72.6%)	1.90 (87.8%)	2.16
Cooker	iset_flg	Yes	4.13 (44.3%)	8.94 (95.9%)	9.32
	wai_flg	Yes	3.49 (36.3%)	9.10 (94.6%)	9.62
	fsnd_dtq	Yes	3.09 (44.1%)	6.63 (94.7%)	7.00
	rcv_dtq	Yes	2.94 (39.6%)	7.03 (94.7%)	7.43
Pot	set_flg	Yes	3.29 (36.6%)	8.59 (95.6%)	8.99
	wai_flg	Yes	3.47 (36.2%)	9.06 (94.6%)	9.57
	fsnd_dtq	Yes	3.10 (42.0%)	7.01 (95.0%)	7.38
	rcv_dtq	Yes	2.96 (39.8%)	7.04 (94.7%)	7.44
Average	w/o Task Switch		1.51 (71.8%)	1.74 (82.9%)	2.10
	w/ Task Switch		3.26 (40.2%)	7.70 (95.0%)	8.11

Table 5.8: RTOS kernel execution time in the maximum frequency.

Program	System Call	Task Switch	w/ RTOS Hardware Adaptive		w/ RTOS Hardware w/o Adaptive		Software w/ Adaptive		Software w/o Adaptive	
			Exec. Time (μsec)	Fmax (MHz)	Exec. Time (μsec)	Fmax (MHz)	Exec. Time (μsec)	Fmax (MHz)	Exec. Time (μsec)	Fmax (MHz)
sem02	sig_sem	No	1.44	68.037	1.44	67.866	1.75	69.262	2.14	69.262
	sig_sem	Yes	3.15		3.16		6.17		6.64	
	pol_sem	No	1.04		1.04		0.92		1.32	
	wai_sem	No	1.26		1.26		0.92		1.41	
	wai_sem	Yes	3.04		3.05		7.48		7.97	
flg02	set_flg	No	1.54	69.517	1.62	65.790	1.93	69.262	2.31	69.262
	set_flg	Yes	3.28		3.46		7.85		8.23	
	pol_flg	No	1.65		1.75		1.56		1.93	
	wai_flg	No	1.83		1.93		1.79		2.17	
	wai_flg	Yes	3.39		3.59		9.01		9.39	
dtq	fsnd_dtq	No	1.52	68.762	1.55	67.381	2.34	69.262	2.70	69.262
	fsnd_dtq	Yes	3.17		3.24		6.83		7.19	
	psnd_dtq	No	1.52		1.55		2.28		2.54	
	psnd_dtq	Yes	3.15		3.22		6.87		7.13	
	rcv_dtq	No	1.55		1.58		1.75		2.00	
	rcv_dtq	Yes	3.01		3.07		6.91		7.30	
	prcv_dtq	No	1.55		1.58		1.87		2.12	
Cooker	iset_flg	Yes	4.05	69.320	4.15	67.660	8.78	69.262	9.15	69.262
	wai_flg	Yes	3.42		3.51		8.93		9.44	
	fsnd_dtq	Yes	3.03		3.11		6.51		6.87	
	rcv_dtq	Yes	2.88		2.95		6.90		7.29	
Pot	set_flg	Yes	3.21	69.624	3.28	68.251	8.43	69.262	8.83	69.262
	wai_flg	Yes	3.39		3.46		8.89		9.40	
	fsnd_dtq	Yes	3.03		3.09		6.88		7.25	
	rcv_dtq	Yes	2.89		2.95		6.91		7.30	
Average	w/o Task Switch		1.49	68.771	1.53	67.049	1.71	69.262	2.06	69.262
	w/ Task Switch		3.21	69.145	3.28	67.540	7.56	69.262	7.96	69.262

Table 5.9: RTOS kernel size (bytes).

Program	System Call	w/ Hardware	Software w/ Adaptive	Software w/o Adaptive
sem02	sig_sem	304 (59.4%)	400 (78.1%)	512
	pol_sem	176 (57.9%)	192 (63.2%)	304
	wai_sem	336 (48.8%)	512 (74.4%)	688
	Kernel Soft	0 (0%)	1,680 (100%)	1,680
	Others	9,104 (69.4%)	9,872 (88.9%)	11,104
	Total	9,920 (69.4%)	12,656 (88.6%)	14,288
flg02	set_flg	320 (44.4%)	608 (84.4%)	720
	pol_flg	336 (65.6%)	400 (78.1%)	512
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	Kernel Soft	0 (0%)	1,392 (100%)	1,392
	Others	10,912 (88.5%)	11,488 (93.1%)	12,336
	Total	12,032 (75.2%)	14,816 (92.6%)	16,000
dtq	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	prcv_dtq	336 (42.9%)	704 (89.8%)	848
	psnd_dtq	320 (51.3%)	544 (87.2%)	624
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,280 (96.4%)	1,328
	Others	9,584 (85.4%)	10,512 (93.7%)	11,216
	Total	10,928 (71.0%)	14,288 (92.8%)	15,392
Cooker	iset_flg	352 (43.1%)	624 (76.5%)	816
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,824 (95.0%)	1,920
	Others	10,592 (85.5%)	11,504 (92.9%)	12,384
	Total	12,096 (68.7%)	16,128 (91.6%)	17,600
Pot	set_flg	320 (44.4%)	608 (84.4%)	720
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,824 (95.0%)	1,920
	Others	11,968 (88.1%)	12,768 (94.0%)	13,584
	Total	13,072 (72.5%)	16,784 (93.1%)	18,032

Table 5.10: RTOS kernel size (sem02) (bytes).

Program	System Call	Software w/ Adaptive Priority only	Software w/ Adaptive	Software w/o Adaptive
sem02	sig_sem	400 (78.1%)	400 (78.1%)	512
	pol_sem	192 (63.2%)	192 (63.2%)	304
	wai_sem	512 (74.4%)	512 (74.4%)	688
	Kernel Soft	1,600 (95.2%)	1,680 (100%)	1,680
	Others	9,872 (88.9%)	9,872 (88.9%)	11,104
	Total	12,576 (88.0%)	12,656 (88.6%)	14,288

Table 5.11: Comparison between automatic and manual adaptation.

Program	Automatic (sec)	Manual (sec)
semflgdtq	2.150	6,900
sem02	1.796	4,740
Cooker	1.953	4,800

Chapter 6

Conclusion

6.1 Summary of the Dissertation

Firstly, we focused on the increase of embedded microprocessors in various appliances and the importance of decreasing the size of a device. As RTOSs are commonly used for the embedded systems, we discussed advantages and disadvantages of using an RTOS. To mitigate the disadvantages, the objectives of this study are described.

As the background of this dissertation, we introduced previous studies, which are related to processor core adaptation, OS kernel adaptation, software overhead mitigation, hardware scheduler, hardware RTOS, and adaptation for RTOS functions in Chapter 2.

In Chapter 3, an architecture of the processor core we implemented was explained. Instruction set of the processor core is MIPS32 instruction set architecture. The microarchitecture is based on 5-stage pipe line consisting of IF, ID, EX, MEM and WB. To build an application adapted processor core, we illustrated the method for analyzing an application program and generating the application adaptive processor core circuit and we evaluated the number of resources and clock rate with three application programs. In addition, we implemented two- to eight-core multi core processor on an FPGA and showed eight-core processor can be implemented on a relatively small FPGA device with application adaptive processor cores while not-adaptive processor cores exceeded the number of the FPGA resources.

In Chapter 4, we proposed a framework to generate an application adapted hardware RTOS and software-only RTOS. First, RTOS features are explained, and then the methods for generating an application adapted RTOS are shown. One of the methods is "Removing Unnecessary Codes Caused by Fixed Attributes" and the other is "Removing Unnecessary Codes Caused by the Way of Calling". After a code of an application program is analyzed, RTOS hardware or software-only RTOS is adapted to the application program. We explained the structure of the hardware RTOS, which consists of RTOS Hardware Wrapper and RTOS Hardware Core. Since RTOS Hardware Wrapper works with a mode transition, the modes and the next states are illustrated. On the other hand, RTOS Hardware Core consists of TCBs and queue headers for RTOS resources. How RTOS Hardware Core works the queue operation is described. In addition, we introduced a GUI tool for generating adapted RTOS hardware and software-only RTOS.

We showed the effect of the proposal in this dissertation with several application programs in Chapter 5. With the development environment, adaptive hardware RTOS for an application program and adaptive software-only RTOS are evaluated. As the result, the number of FPGA resources, RTOS kernel execution time and the size of the software parts are shown. It can be seen that LUT resources in the FPGA are reduced by 44.3% with the same application program (semflgdtq) comparing both of the CPU and RTOS adaptive configurations and no adaptive configurations. For average system call execution time, using hardware RTOS is 2.49 times faster in the case with task switch than the case of software-only RTOS without adaptation. The RTOS kernel size of a software part can be reduced with RTOS hardware or software-only RTOS with adaptation. For example, the application program sem02 can reduce the size of a software part to 69.4% in the case of RTOS hardware and to 88.6% in the case of software with adaptation.

6.2 Conclusion

In summary, we proposed a method for developing an application adaptive processor core and generating a hardware RTOS or software-only RTOS kernel in a fully automatic manner with illustrating the effectiveness of the techniques by showing several experimental results.

Our approach does not need to:

- introduce new compact instruction set,
- develop new compiler, or
- update RTOS specification,

in order to reduce hardware sizes, improve running frequency, reduce RTOS sizes and improve execution times. All the proposed adaptations are achieved in a fully automatic manner. While the method we proposed is based on FPGA, the method can be applied to not only FPGA but also ASIC development.

As shown in Chapter 5, hardware resources can be reduced by around 40% by adaptation to an application. In general, the device cost decreases in proportion to the usage of resources so that the device cost can be reduced by that amount. We conclude this dissertation by stating that the proposed environment makes it possible to produce low cost and effective devices in a short period of time.

6.3 Future Work

While we adopted MIPS32 architecture for the processor architecture and μITRON4.0 for the RTOS specification, which is one of the most popular RTOS specification for embedded RTOS kernel, the proposed method is expected to be applied to another processor architecture and RTOS specification. As shown in the literature [85], we are studying another processor architecture such as RISC-V processor in Ref.[86] so that we can enhance the framework to another processor core. RISC-V has been gaining popularity for IoT devices due to the open architecture and suitable for IoT devices. Regarding an RTOS specification, for example, a specification of μT-Kernel in Ref.[87] has a similar attribute described in Section 4.3.1 and error checking described in Section 4.3.2 so that it would be possible to achieve adaptation described in this dissertation. We will expand the proposed method in other platforms and study generating more fine-grained RTOSs.

Necessity of application specific hardware and software is increasing due to expansion of IoT devices as described in the introduction. Since embedded processors and RTOS are fundamental technology, which are commonly used in lots of appliances, application specific processors and reduction of RTOS overhead are desirable. While application specific processors are not flexible to be used with another application, that is advantage in the security point of view as unauthorized software may not work. Functions of embedded processors and RTOS are the key technology of IoT devices so that the study in this dissertation will lead to the next study for processor architecture and RTOS which are suitable for IoT devices.

Appendix A

This appendix explains the state transition described in Section 4.4.3 in detail.

A.1 INIT

The state of RTOS hardware begins from the INIT state after the reset. INIT is a state for initializing RTOS hardware internal registers. The state transition is shown in Table A.1.

Table A.1: INIT.

System call	Next state	Condition
All system calls	WAIT	All system calls transit to WAIT at the next clock.

A.2 WAIT

In the WAIT state, RTOS hardware waits for an issue of a system call. If a system call is not called, the RTOS Hardware Wrapper stays in the WAIT state. The state transition is shown in Table A.2.

Table A.2: WAIT.

System call	Next state	Condition
All system calls	CHECK	When a system call is issued, the state transits to CHECK.

A.3 CHECK

In the CHECK state, RTOS hardware checks system call parameter errors. When a parameter error is found, the RTOS Hardware Wrapper transits to the END state, otherwise RTOS Hardware Wrapper transits to a proper state to proceed to a system call operation. The state transition is shown in Table A.3.

Table A.3: CHECK.

System call	Next state	Condition
All system calls	END	When a parameter error is found, the state transits to the END state.
act_tsk	ACTCNT	When the task is not DORMANT.
	RDYENQUEUE	When the task is DORMANT.
ext_tsk	RDYDEQUEUE	When the activation request count is 1 or more.
	END	When the activation request count is 0.
chg_pri	TASKSTATUS	When there is no error.
ter_tsk	RDYDEQUEUE	When there is no error.
wup_tsk	RDYENQUEUE	When there is no error.
can_wup	END	When there is no error.
slp_tsk	RDYDEQUEUE	When the wakeup request count is 1 or more.
	END	When the wakeup request count is 0.
rel_wai	TASKSTATUS	When there is no error.
can_act	END	When there is no error.
sig_sem	SEMSTATUS	When there is no error.
wai_sem	SEMSTATUS	When there is no error.
pol_sem	SEMSTATUS	When there is no error.
set_flg	FLGSTATUS	When there is no error.
wai_flg	FLGSTATUS	When there is no error.
pol_flg	FLGSTATUS	When there is no error.
clr_flg	SETATTR	When there is no error.
psnd_dtq	DTQSTATUS	When there is no error.
fsnd_dtq	DTQDEQUEUE	When there is a waiting task.
	DTQSTATUS	When the data queue area is not 0 and there is no waiting task.
	END	When the data queue area is 0 and there is no waiting task.
rcv_dtq	DTQSTATUS	When there is no error.
prcv_dtq	DTQSTATUS	When there is no error.
snd_dtq	DTQSTATUS	When there is no error.

A.4 SETATTR

In the SETATTR state, an internal attribute of a system call resource is set. The state transition is shown in Table A.4.

Table A.4: SETATTR.

System call	Next state	Condition
clr_flg	END	The state transits to the END state at the next clock.

A.5 TASKSTATUS

In the TASKSTATUS state, an operation TASKSTATUS in Table 4.7 is issued to the RTOS hardware. The state transition is shown in Table A.5.

Table A.5: TASKSTATUS.

System call	Next state	Condition
chg_pri	CHECKTASK	The state transits to the CHECKTASK state at the next clock.
rel_wai	CHECKTASK	The state transits to the CHECKTASK state at the next clock.

A.6 SEMSTATUS

In the SEMSTATUS state, an operation SEMHEAD in Table 4.7 is issued to the RTOS hardware. The state transition is shown in Table A.6.

Table A.6: SEMSTATUS.

System call	Next state	Condition
sig_sem	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
wai_sem	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
pol_sem	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.

A.7 FLGSTATUS

In the FLGSTATUS state, an operation FLGHEAD in Table 4.7 is issued to the RTOS hardware. The state transition is shown in Table A.7.

Table A.7: FLGSTATUS.

System call	Next state	Condition
set_flg	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
wai_flg	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
pol_flg	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.

A.8 DTQSTATUS

In the DTQSTATUS state, an operation DTQHEAD in Table 4.7 is issued to the RTOS hardware. The state transition is shown in Table A.8.

Table A.8: DTQSTATUS.

System call	Next state	Condition
psnd_dtq	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
fsnd_dtq	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
rcv_dtq	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
prcv_dtq	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.
snd_dtq	CHECKSTATUS	The state transits to the CHECKSTATUS state at the next clock.

A.9 CHECKTASK

In the CHECKTASK state, the result of the previous state, TASKSTATUS, is checked and the next state is decided by the result. The state transition is shown in Table A.9.

Table A.9: CHECKTASK.

System call	Next state	Condition
chg_pri	RDYDEQUEUE	When the target task is in a ready queue.
	SEMDEQUEUE	When the target task is in a semaphore waiting queue.
	CHGPRI	When the target task is in in an eventflag waiting queue or data queue waiting queue.
rel_wai	SEMDEQUEUE	When the target task is in a semaphore waiting queue.
	FLGDEQUEUE	When the target task is in an eventflag waiting queue.
	DTQDEQUEUE	When the target task is in a data queue waiting queue.
	END	When the target task is not waiting.

A.10 CHECKSTATUS

In the CHECKSTATUS state, the result of the previous state is checked and the next state is decided to proceed to the system call operation. The state transition is shown in Table A.10.

Table A.10: CHECKSTATUS.

System call	Next state	Condition
sig_sem	SEMCNT	When there is a waiting task.
	SEMDEQUEUE	When there is no waiting task.
wai_sem	SEMCNT	When the semaphore is taken.
	RDYDEQUEUE	When the semaphore is not taken.
pol_sem	SEMCNT	When the semaphore is taken.
	END	When the semaphore is not taken.
set_flg	FLGDEQUEUE	When the flag condition is satisfied.
	END	When there is no waiting task or the flag condition is not satisfied.
wai_flg	RDYDEQUEUE	When the flag condition is not satisfied.
	END	When the flag condition is satisfied.
pol_flg	END	When the state transits to the END state at the next clock.
psnd_dtq	DTQDEQUEUE	When there is a waiting task.
	DTQDATA	When there is no waiting task and there is a buffer to write the data in the data queue.
	END	When there is no waiting task and no buffer in the data queue.
fsnd_dtq	DTQDEQUEUE	When there is a waiting task to receive data.
	DTQDATA	When there is no waiting task to receive data.
rcv_dtq	DTQDEQUEUE	When there is a waiting task to send data.
	RDYDEQUEUE	When there is no data in the buffer.
	DTQDATA	When there is data to receive in the buffer.
prcv_dtq	DTQDEQUEUE	When there is a waiting task to send data.
	DTQGETDATA	When there is data in the buffer.
	END	When there is no data in the buffer.
snd_dtq	DTQDEQUEUE	When there is a waiting task to receive data.
	RDYDEQUEUE	When there is no buffer to send data.

A.11 ACTCNT

In the ACTCNT state, the activation count is increased. The state transition is shown in Table A.11.

Table A.11: ACTCNT.

System call	Next state	Condition
act_tsk	END	The state transits to the END state at the next clock.

A.12 SEMCNT

In the SEMCNT state, the semaphore count is increased. The state transition is shown in Table A.12.

Table A.12: SEMCNT.

System call	Next state	Condition
sig_sem	END	The state transits to the END state at the next clock.
wai_sem	END	The state transits to the END state at the next clock.
pol_sem	END	The state transits to the END state at the next clock.

A.13 RDYDEQUEUE

In the RDYDEQUEUE state, an operation READYENQUEUE in Table 4.7 is issued to release a task from the ready queue. The state transition is shown in Table A.13.

Table A.13: RDYDEQUEUE.

System call	Next state	Condition
ext_tsk	HIGHEST	The state transits to HIGHEST at the next clock.
chg_pri	CHGPRI	The state transits to CHGPRI at the next clock.
ter_tsk	RDYDEQUEUE	The state transits to RDYDEQUEUE at the next clock.
slp_tsk	HIGHEST	The state transits to HIGHEST at the next clock.
wai_sem	SEMENQUEUE	The state transits to SEMENQUEUE at the next clock.
wai_flg	FLGENQUEUE	The state transits to FLGENQUEUE at the next clock.
rcv_dtq	DTQENQUEUE	The state transits to DTQENQUEUE at the next clock.
snd_dtq	DTQENQUEUE	The state transits to DTQENQUEUE at the next clock.

A.14 SEMDEQUEUE

In the SEMDEQUEUE state, an operation SEMDEQUEUE in Table 4.7 is issued to release a task from the semaphore waiting queue. The state transition is shown in Table A.14.

Table A.14: SEMDEQUEUE.

System call	Next state	Condition
chg_pri	HIGHEST	The state transits to HIGHEST at the next clock.
rel_wai	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
sig_sem	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.

A.15 CHGPRI

In the CHGPRI state, an operation PRICHG in Table 4.7 is issued to change the priority of a task. The state transition is shown in Table A.15.

Table A.15: CHGPRI.

System call	Next state	Condition
chg_pri	RDYENQUEUE	Target task is queued in the ready queue.
	SEMENQUEUE	Target task is queued in the semaphore queue.
	END	Other cases.

A.16 SEMENQUEUE

In the SEMENQUEUE state, an operation SEMENQUEUE in Table 4.7 is issued to queue a task to the semaphore waiting queue. The state transition is shown in Table A.16.

Table A.16: SEMENQUEUE.

System call	Next state	Condition
chg_pri	HIGHEST	The state transits to HIGHEST at the next clock.
wai_sem	HIGHEST	The state transits to HIGHEST at the next clock.
	END	The state transits to END at the next clock.

A.17 FLGDEQUEUE

In the FLGDEQUEUE state, an operation FLGDEQUEUE in Table 4.7 is issued to release a task from the eventflag waiting queue. The state transition is shown in Table A.17.

Table A.17: FLGDEQUEUE.

System call	Next state	Condition
rel_wai	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
set_flg	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.

A.18 FLGENQUEUE

In the FLGENQUEUE state, an operation FLGENQUEUE in Table 4.7 is issued to queue a task to the eventflag waiting queue. The mode transition is shown in Table A.18.

Table A.18: FLGENQUEUE.

System call	Next state	Condition
wai_flg	HIGHEST	The state transits to HIGHEST at the next clock.

A.19 DTQDEQUEUE

In the DTQDEQUEUE state, an operation DTQDEQUEUE in Table 4.7 is issued to release a task from the data queue waiting queue. The state transition is shown in Table A.19.

Table A.19: DTQDEQUEUE.

System call	Next state	Condition
rel_wai	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
psnd_dtq	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
fsnd_dtq	RDYENQUEUE	When there is a waiting task to receive data.
	DTQENQUEUE	When there is no waiting task to receive data.
rcv_dtq	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
prcv_dtq	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.
snd_dtq	RDYENQUEUE	The state transits to RDYENQUEUE at the next clock.

A.20 DTQRCVENQUEUE

In the DTQRCVENQUEUE state, an operation DTQENQUEUE in Table 4.7 is issued to queue a task to the data queue waiting queue. This state is entered when rcv_dtq is invoked and there is no data in the data queue. The state transition is shown in Table A.20.

Table A.20: DTQRCVENQUEUE.

System call	Next state	Condition
rcv_dtq	HIGHEST	The state transits to HIGHEST at the next clock.

A.21 DTQSNDENQUEUE

In the DTQSNDENQUEUE state, an operation DTQENQUEUE in Table 4.7 is issued to queue a task to the data queue waiting queue. This state is transited when snd_dtq or fsnd_dtq is invoked and there is no room in the data queue area. The state transition is shown in Table A.21.

Table A.21: DTQSNDENQUEUE.

System call	Next state	Condition
fsnd_dtq	HIGHEST	The state transits to HIGHEST at the next clock.
snd_dtq	HIGHEST	The state transits to HIGHEST at the next clock.

A.22 DTQDATA

In the DTQDATA state, a data element is placed to the data queue. The state transition is shown in Table A.22.

Table A.22: DTQDATA.

System call	Next state	Condition
psnd_dtq	END	The state transits to END at the next clock.
fsnd_dtq	END	The state transits to END at the next clock.
rcv_dtq	END	The state transits to END at the next clock.
prcv_dtq	END	The state transits to END at the next clock.

A.23 RDYENQUEUE

In the RDYENQUEUE state, an operation READYENQUEUE in Table 4.7 is issued to queue a task to the ready queue. The mode transition is shown in Table A.23.

Table A.23: RDYENQUEUE.

System call	Next state	Condition
act_tsk	HIGHEST	The state transits to HIGHEST at the next clock.
chg_pri	HIGHEST	The state transits to HIGHEST at the next clock.
wup_tsk	HIGHEST	When a task switch occurs.
	END	When there is no task switch.
rel_wai	HIGHEST	The state transits to HIGHEST at the next clock.
sig_sem	HIGHEST	The state transits to HIGHEST at the next clock.
set_flg	HIGHEST	The state transits to HIGHEST at the next clock.
psnd_dtq	HIGHEST	The state transits to HIGHEST at the next clock.
fsnd_dtq	HIGHEST	The state transits to HIGHEST at the next clock.
rcv_dtq	HIGHEST	The state transits to HIGHEST at the next clock.
prcv_dtq	HIGHEST	The state transits to HIGHEST at the next clock.
snd_dtq	HIGHEST	The state transits to HIGHEST at the next clock.

A.24 HIGHEST

In the HIGHEST state, an operation PRIHIGHEST in Table 4.7 is issued to get the task with the highest precedence. The state transition is shown in Table A.24.

Table A.24: HIGHEST.

System call	Next state	Condition
act_tsk	END	The state transits to END at the next clock.
ext_tsk	END	The state transits to END at the next clock.
chg_pri	END	The state transits to END at the next clock.
ter_tsk	END	The state transits to END at the next clock.
wup_tsk	END	The state transits to END at the next clock.
slp_tsk	END	The state transits to END at the next clock.
rel_wai	END	The state transits to END at the next clock.
sig_sem	END	The state transits to END at the next clock.
wai_sem	END	The state transits to END at the next clock.
set_flg	END	The state transits to END at the next clock.
wai_flg	END	The state transits to END at the next clock.
psnd_dtq	END	The state transits to END at the next clock.
fsnd_dtq	END	The state transits to END at the next clock.
rcv_dtq	END	The state transits to END at the next clock.
prcv_dtq	END	The state transits to END at the next clock.
snd_dtq	END	The state transits to END at the next clock.

A.25 END

In the END state, exiting from RTOS hardware is prepared. The state transition is shown in Table A.25.

Table A.25: END.

System call	Next state	Condition
act_tsk	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
ext_tsk	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
chg_pri	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
wup_tsk	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
slp_tsk	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
rel_wai	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
sig_sem	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
wai_sem	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
set_flg	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
wai_flg	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
psnd_dtq	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
fsnd_dtq	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
rcv_dtq	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
prcv_dtq	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
snd_dtq	ENDSWITCH	The state transits to ENDSWITCH at the next clock.
All system calls except for the above	WAIT	The state transits to WAIT at the next clock.

A.26 ENDSWITCH

As described in section 4.4.2, when the task switch is necessary in invoking a system call, a task ID of the highest priority task is returned to the software part. In the ENDSWITCH state, the task ID is set to the return code. The state transition is shown in Table A.26.

Table A.26: ENDSWITCH.

System call	Next state	Condition
act_tsk	WAIT	The state transits to WAIT at the next clock.
ext_tsk	WAIT	The state transits to WAIT at the next clock.
chg_pri	WAIT	The state transits to WAIT at the next clock.
wup_tsk	WAIT	The state transits to WAIT at the next clock.
slp_tsk	WAIT	The state transits to WAIT at the next clock.
rel_wai	WAIT	The state transits to WAIT at the next clock.
sig_sem	WAIT	The state transits to WAIT at the next clock.
wai_sem	WAIT	The state transits to WAIT at the next clock.
set_flg	WAIT	The state transits to WAIT at the next clock.
wai_flg	WAIT	The state transits to WAIT at the next clock.
psnd_dtq	WAIT	The state transits to WAIT at the next clock.
fsnd_dtq	WAIT	The state transits to WAIT at the next clock.
rcv_dtq	WAIT	The state transits to WAIT at the next clock.
prcv_dtq	WAIT	The state transits to WAIT at the next clock.
snd_dtq	WAIT	The state transits to WAIT at the next clock.

Bibliography

[1] Jain, M.K., Balakrishnan, M. and Kumar, A.: ASIP Design Methodologies: Survey and Issues, *Proc. 14th Intl. Conf. on VLSI Design*, pp.76–81 (2001).

[2] Imai, M., Takeuchi, Y., Sakanushi, K. and Ishiura, N.: Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP), *IPSJ Transactions on System LSI Design Methodology*, Vol.3, pp.161–178 (2010).

[3] Shackleford, B., Yasuda, M., Okushi, E., Koizumi, H., Tomiyama, H. and Yasuura, H.: Memory-CPU Size Optimization for Embedded System Designs, *Proc. the 34th annual Design Automation, DAC'97*, pp.246–251 (1997).

[4] Takada, H. and Sakamura, K.: μITRON for Small-Scale Embedded Systems, *IEEE Micro*, Vol.15, No.6, pp.46–54 (1995).

[5] Anh, T.N.B. and Tan, S.-L.: Real-Time Operating Systems for Small Microcontrollers, *IEEE Micro*, Vol.29, No.5, pp.30–45 (2009).

[6] Renesas RL78 Family Microcontrollers [Online] Available: `https://www.renesas.com/us/en/products/microcontrollers-microprocessors/rl78.html`

[7] μITRON4.0 Specification Ver.4.01.00, ITRON Committee, TRON ASSOCIATION.

[8] Zynq-7000 SoC, Technical Reference Manual, UG585 (v1.12.2), Xilinx, July 1 (2018).

[9] Seely, J., Erusalagandi, S. and Bethurem, J: The MicroBlaze Soft Processor:Flexibility and Performance for Cost-Sensitive Embedded Designs, Xilinx, WP501 (v1.0) April 13 (2017).

[10] Nios®II Processor Reference Guide, NII-PRG, 2020.10.22 (2020).

[11] Gonzalez, R.E.: XTENSA:A Configurable and Extensible Processor, *IEEE Micro*, Vol.20, No.2, pp.60–70 (2000).

[12] Yiannacouras, P., Rose, J., Steffan, J.G.: The Microarchitecture of FPGABased Soft Processors, *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded systems*, pp.202–212 (2005).

[13] Yiannacouras, P., Steffan, J.G., Rose, J.: Application-Specific Customization of Soft Processor Microarchitecture, *Proc. ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pp.201–210 (2006).

[14] He, H., Trimble, J., Perianayagam, S., Debray, S. and Andrews, G.: Code Compaction of an Operating System Kernel, *Proc. the International Symposium on Code Generation and Optimization*, pp.283–298 (2007).

[15] OSEK group, OSEK/VDX Operating System Specification 2.2.3, February 17th (2005).

[16] Deifel, H.P., Göttlinger, M., Milius, S. and Schröder, L., Dietrich, G. and Lohmann, D.: Automatic Verification of Application-Tailored OSEK Kernels, *Proc. Formal Methods in Computer Aided Design*, pp.196–203 (2017).

[17] Akgul, B.S., Lee, J. and Mooney, V.J.: A System-on-a-Chip Lock Cache with Task Preemption Support, *Proc. the 2001 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01)*, pp.149–157 (2001).

[18] Saglam, B.E. and Mooney III, V.J.: System-on-a-Chip Processor Synchronization Support in Hardware, *Proc. Design, Automation and Test in Europe (DATE 01), IEEE CS Press*, pp.633–639 (2001).

[19] Mooney III, V.J. and Blough, D.M.: A Hardware-Software Real-Time Operating System Framework for SoCs, *IEEE Design & Test*, Vol.19, No.6, pp.44–51 (2002).

[20] Nordstrom, S., Lindh, L., Johansson, L. and Skoglund, T.: Application Specific Real-Time Microkernel in Hardware, *Proc. IEEE-NPSS Real Time Conference (RTC)*, pp.79–82 (2005).

[21] Kuacharoen, P., Shalan, M.A. and Mooney III, V.J.: A Configurable Hardware Scheduler for Real-Time Systems, *Proc. the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp.96–101 (2003).

[22] Vetromille, M., Ost, L., Marcon, C.A.M., Reif, C. and Hessel, F.: RTOS Scheduler Implementation in Hardware and Software for Real Time Applications, *Proc. Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, pp.163–168 (2006).

[23] Ueda, R, Fujii, K., Chishiro, H., Matsutani, M. and Yamasaki, N.: Extension of ITRON Specification OS for Multithreaded Processors, *IEICE Technical Report*, Vol.111, No.397, pp.43-48 (2012). (In Japanese)

[24] Ueda, R, Fujii, K., Chishiro, H., Matsutani, M. and Yamasaki, N.: Implementation of ITRON Specification OS for RMT Processor, *IPSJ Journal*, Vol54, No.7, pp.1835–1848 (2013). (In Japanese)

[25] Tang, Y., Bergmann, N.W.: A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems, *IEEE Transactions on Computers*, Vol.64, No.5, pp.1254–1267 (2015).

[26] Gomes, T., Pinto, S., Garcia, P. and Tavares, A.: RT-SHADOWS:Real-Time System Hardware for Agnostic and Deterministic OSes Within Softcore, *Proc. the IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp.1–4 (2015).

[27] Gomes, T., Garcia, P., Pinto, S., Monteiro, J. and Tavares, A.: Bringing Hardware Multithreading to the Real-Time Domain, *IEEE Embedded Systems Letters*, Vol.8, No.1, pp.2–5 (2016).

[28] Deng, Q, Wei, S., Xu, H., Han, Y. and Yu, G.: A Reconfigurable RTOS with HW/SW Co-scheduling for SOPC, *Proc. the Second International Conference on Embedded Software and Systems (ICESS'05)*, 6pages (2005).

[29] Fujimoto, K., Takiguchi, H., Nakamura, S., Nankaku, S. and Noborito, H: Proposed high-speed technique using hardware implementation of the interrupt scheduler, *Proc. 2014 Annual Conference on Electronics, Information and Systems, IEEJ*, pp.550–555 (2014). (In Japanese)

[30] Fujimoto, K., Takiguchi, H., Nakamura, S., Watanabe, K., Nankaku, S. and Noborito, H: Proposal and Evaluation High-speed Technique using Hardware Implementation of the Interrupt Scheduler, *IEEJ Transactions on Electronics, Information and Systems*, Vol.135, No.11, pp.1427-1438 (2015). (In Japanese)

[31] Kim, B.K. and Shin, K.G.: Hardware Earliest-Deadline-First Scheduler for ATM Switching Networks, *Proc. the Real-time Systems Symposium*, pp.210–218 (1997).

[32] Kobayashi, S. and Mitsui, H.: Implementing embedded OS for sensor node using Hardware, *Proc.73th National Convention of IPSJ*, Vol.1, pp.159–160 (2011). (In Japanese)

[33] Kobayashi, S., Kobayashi, K. and Mitsui, H.: Hardware Implementation of Embedded OS for Sensor Node Using FPGA, *Proc. 2011 Annual Conference on Electronics, Information and Systems, IEEJ* pp.692–697 (2011). (In Japanese)

[34] Utama, A., Itabashi, M., Nakano, T., Shiomi, A. and Imai, M: The Evaluation and Implementation of Silicon TRON, *IEICE Technical Report*, VLD94-40, pp.9–16 (1994). (In Japanese)

[35] Nakano, T., Utama, A., Itabashi, M., Shiomi, A. and Imai, M.: Hardware Implementation of a Real-time Operating System, *Proc. the 12th TRON Project International Symposium, IEEE*, pp.34–42 (1995).

[36] Nakano, T., Utama, A., Itabashi, M., Shiomi, A. and Imai, M.: VLSI Implementation and Evaluation of a Real-Time Operating System, *IEICE Trans. Inf.&Syst.* Vol.J78-D1, No.8, pp.679–686 (1995). (In Japanese)

[37] Nakano, T., Komatsudaira, Y., Shiomi, A., Imai, M.: Performance Evaluation of STRON: A Hardware Implementation of a Real-Time OS, *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, Vol.E82-A, No.11, pp.2375-2382 (1999).

[38] Itabashi, M., Utama, A., Nakano, T., Shiomi, A. and Imai, M.: Implement Real-Time OS by Hardware and Evaluation, *IPSJ SIG Technical Reports*, 1993-ARC-103, pp.183-190 (1993). (In Japanese)

[39] Mori, H., Sakamaki, K. and Shigematsu, H.: Hardware Implementation of a real-time operating system for embedded control systems, *Tokyo Metropolitan Industrial Technology Bulletin of Study* No.8, pp55–58 (2005). (In Japanese)

[40] Murakoshi, H., Takeda, Y. and Katagiri, H.: Extraction of RTOS Kernel Functions from UML and Implementation by FPGA, *Bulletin of Advanced Institute of Industrial Technology* No.1, pp.127–130 (2007). (In Japanese)

[41] Adomat, J., Furunäs J., Lindh, L. and Stärner, J.: Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-performance Real-Time Systems, *Proc. EURWRTS '96*, pp.164–168 (1996).

[42] Andrews, D., Peck, W., Agron, J., Preston, K., Komp, E., Finley, M. and Sass,R.: hthreads: A hardware/software co-designed multithreaded RTOS kernel, *Proc. 2005 IEEE Conference on Emerging Technologies and Factory Automation*, pp.331–338 (2005).

[43] Lange, A.B., Andersen, K.H., Schultz, U.P. and Sorensen, A.S.: HartOS – a Hardware Implemented RTOS for Hard Real-Time Applications, *Proc. 11th IFAC, IEEE International Conference on Programmable Devices and Embedded Systems*, Vol.45, No.7, pp.207–213 (2012).

[44] Maruyama, N., Ishihara, T. and Yasuura, H.: Exploiting Virtual Queue for Implementing a High Performance RTOS in Hardware, *Proc. 9th Forum on Information Technology*, Vol.1, pp.115–120 (2010). (In Japanese)

[45] Maruyama, N., Ishihara, T. and Yasuura, H.: An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing, *Proc. IEEE 8th Symposium on Application Specific Processors (SASP)*, pp.58–63 (2010).

[46] Maruyama, N., Ishihara, T. and Yasuura, H.: An Energy Efficient Software-Based TCP/IP Processing Method Using an RTOS in Hardware, *IEICE Transactions on Information and Systems, A*, Vol.J94-A, No.9, pp.692–701 (2011). (In Japanese)

[47] Maruyama, N., Ichiba, T., Honda, S. and Takada, H.: A Hardware RTOS for Multicore Systems, *IEICE Transactions on Information and Systems, D*, Vol.J96-D, No.10, pp.2150–2162 (2013). (In Japanese)

[48] Maruyama, N., Ishikawa, T., Honda, S., Takada, H., Suzuki, K.: ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems, *Proc. The 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp.9–16 (2014).

[49] Maruyama, N., Ishikawa, T., Honda, S., Takada, H., Suzuki, K.: A SoC with Loosely Coupled Type Hardware RTOS for Industrial Network Systems, *IEICE Transactions on Information and Systems, D*, Vol.J98-D, No.4, pp.661–673 (2015).

[50] Ong, S.F., Lee, S.C., Ali, N.B.Z., Hussin, F.A.B.: SEOS:Hardware Implementation of Real-Time Operating System for Adaptability, *Proc. 2013 First International Symposium on Computing and Networking*, pp.612–616, IEEE (2013).

[51] Dietrich, C. and Lohmann, D.: OSEK-V:Application-Specific RTOS Instantiation in Hardware, *Proc. The 18th Annual ACM SIGPLAN / SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp.111–120 (2017).

[52] Oosako, Y., Ishiura, N., Tomiyama, H. and Kanbara, H.: Synthesis of Full Hardware Implementation of RTOS-Based Systems, *Proc. 2018 International Symposium on Rapid System Prototyping*, pp.1–7, IEEE (2018).

[53] Miyauchi, T. and Tanaka, K.: Building a Framework for an Application-Adaptive Processor System on FPGA-based SoC, *Proc. The 21st Workshop on Synthesis And System Integration of Mixed Information technologies*, pp.359–364 (2018).

[54] Miyauchi, T. and Tanaka, K.: An Adaptive Approach for Implementing RTOS in Hardware, *Proc. Embedded Systems Symposium*, pp.44–50 (2018).

[55] Miyauchi, T. and Tanaka, K.: A Configurator to Optimize Soft Processors for FPGA, *Proc. 77th National Convention of IPSJ*, pp.23–24 (2015). (In Japanese)

[56] Miyauchi, T. and Tanaka, K.: A Framework for Automatic Generation of Application-Specific FPGA-based SoC, *Proc. The 20th Workshop on Synthesis And System Integration of Mixed Information technologies* pp.305–310 (2016).

[57] Miyauchi, T. and Tanaka, K.: Building Automatic Optimizing Environment for Multicore Processors, *Proc. Embedded Systems Symposium 2015*, pp.99–104 (2015). (In Japanese)

[58] MIPS® Architecture For Programmers, Volume II-A: The MIPS32® Instruction Set (2013).

[59] Patterson, D.A. and Hennessy, J.: "Computer Organization & Design, 4th edition, Japanese edition", ISBN:978-4-8222-8479-4 (2013).

[60] Basys 3® FPGA Board Reference Manual, Revised April8, 2016 (2016).

[61] Miyauchi, T.: Building a Configurator to Optimize Soft Processors for FPGA, Master thesis, Japan Advanced Institute of Science and Technology (2015).

[62] Rijndael [Online] Available `https://embeddedsw.net/Cipher_Reference_Home.html#AES`

[63] The 2nd ARC/CPSY/RECONF High-Performance Computer System Design Contest [Online] Available `http://www.is.utsunomiya-u.ac.jp/pearlab/contest/`

[64] Artix-7 [Online] Available `https://japan.xilinx.com/products/silicon-devices/fpga/artix-7.html`

[65] Xilinx answer record:AR# 57304 [Online] Available `https://japan.xilinx.com/products/silicon-devices/fpga/artix-7.html`

[66] Xilinx Vivado [Online] Available `https://www.xilinx.com/products/design-tools/vivado.html`

[67] Miyauchi, T. and Tanaka, K.: Configuration Technique for Adaptability of Multicore Processors on FPGA, *Proc. The 27th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2 pages (2016).

[68] Miyauchi, T. and Tanaka, K.: An Adaptive Approach to Cache Memory for Soft Processors on FPGA, *Proc. 78th National Convention of IPSJ*, pp.21–22 (2016). (In Japanese)

[69] Miyauchi, T. and Tanaka, K.: Implementation of Multicore Processors by Automatic Optimization for FPGA, *Proc. Joint conference of Hokuriku chapters of Electrical and information Societies 2015*, 1 page, CD-ROM (2015). (In Japanese)

[70] Spartan6 [Online] Available `http://www.xilinx.com/products/silicondevices/fpga/spartan-6.html`

[71] Atlys [Online] Available `https://reference.digilentinc.com/reference/programmable-logic/atlys/start`

[72] ISE [Online] Available `https://www.xilinx.com/products/design-tools/ise-design-suite.html`

[73] Miyauchi, T. and Tanaka, K.: Building Fine-Grained Configurable ITRON Based RTOS, *Journal of Information Processing*, Vol.28, pp.395–405 (2020).

[74] Miyauchi, T. and Tanaka, K.: Overview of An Adaptive Approach for Implementing RTOS in Hardware, *Proc. Asia Pacific Conference on Robot IoT System Development and Platform*, 2pages (2018).

[75] Miyauchi, T. and Tanaka, K.: Fine-Grained Configuration of RTOS Adapted to Applications, *Proc. Embedded Systems Symposium 2016*, pp.73–81 (2016). (In Japanese)

[76] Miyauchi, T. and Tanaka, K.: A Study of Hardware Acceleration of RTOS using FPGA, *Proc. 79th National Convention of IPSJ*, pp.9–10 (2017). (In Japanese)

[77] Xilinx COE File Syntax [Online] Available `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgn_r_coe_file_syntax.htm`

[78] Ubuntu [Online] Available `https://ubuntu.com/`

[79] Python [Online] Available `https://www.python.org/`

[80] wxPython [Online] Available `https://docs.wxpython.org/`

[81] fileHunter [Online] Available `https://github.com/dsbmac/ProfessionalDevelopment/blob/master/Python/wxPython/fileHunter.py`

[82] 7 Series FPGAs Configurable Logic Block User Guide, Xilinx, UG474 (v1.8) September 27 (2016).

[83] TOPPERS Kernel Test Suites [Online] Available `https://www.toppers.jp/testsuites.html`

[84] Tamura, K.: A Study on a development environment of general user interface for embedded systems, Master thesis, Japan Advanced Institute of Science and Technology (2014).

[85] Miyauchi, T. and Tanaka, K.: A Proposal of Application Specific Approach with RISC-V Processor on FPGA, *Proc. The 22nd Workshop on Synthesis And System Integration of Mixed Information Technologies*, 4pages (2019).

[86] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design, RISC-V Edition”, Morgan Kaufmann Publishers (2018).

[87] μT-Kernel 3.0 (2019). [Online] Available `https://www.tron.org/specifications/`

Publications

Journal

[1] Miyauchi, T. and Tanaka, K.: Building Fine-Grained Configurable ITRON Based RTOS, *Journal of Information Processing*, Vol.28, pp.395–405 (2020). (with review)

International Conference

[2] Miyauchi, T. and Tanaka, K.: A Proposal of Application Specific Approach with RISC-V Processor on FPGA, *Proc. The 22nd Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp.270–273 (2019). (with review)

[3] Miyauchi, T. and Tanaka, K.: Overview of An Adaptive Approach for Implementing RTOS in Hardware, *Proc. Asia Pacific Conference on Robot IoT System Development and Platform*, 2pages (2018). (with review)

[4] Miyauchi, T. and Tanaka, K.: Building a Framework for an Application-Adaptive Processor System on FPGA-based SoC, *Proc. The 21st Workshop on Synthesis And System Integration of Mixed Information technologies*, pp.359–364 (2018). (with review)

[5] Miyauchi, T. and Tanaka, K.: A Framework for Automatic Generation of Application-Specific FPGA-based SoC, *Proc. The 20th Workshop on Synthesis And System Integration of Mixed Information technologies* pp.305–310 (2016). (with review)

[6] Miyauchi, T. and Tanaka, K.: Configuration Technique for Adaptability of Multicore Processors on FPGA, *Proc. The 27th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2 pages (2016). (with review)

Domestic Conference

[7] Miyauchi, T. and Tanaka, K.: An Adaptive Approach for Implementing RTOS in Hardware, *Proc. Embedded Systems Symposium 2018*, pp.44–50 (2018). (with review)

[8] Miyauchi, T. and Tanaka, K.: Fine-Grained Configuration of RTOS Adapted to Applications, *Proc. Embedded Systems Symposium 2016*, pp.73–81 (2016). (In Japanese) (with review)

[9] Miyauchi, T. and Tanaka, K.: Building Automatic Optimizing Environment for Multicore Processors, *Proc. Embedded Systems Symposium 2015*, pp.99–104 (2015). (In Japanese) (with review)

[10] Miyauchi, T. and Tanaka, K.: A Study of Hardware Acceleration of RTOS using FPGA, *Proc. 79th National Convention of IPSJ*, pp.9–10 (2017). (In Japanese)

[11] Miyauchi, T. and Tanaka, K.: An Adaptive Approach to Cache Memory for Soft Processors on FPGA, *Proc. 78th National Convention of IPSJ*, pp.21–22 (2016). (In Japanese)

[12] Miyauchi, T. and Tanaka, K.: Implementation of Multicore Processors by Automatic Optimization for FPGA, *Proc. Joint conference of Hokuriku chapters of Electrical and information Societies 2015*, 1 page, CD-ROM (2015). (In Japanese)

[13] Miyauchi, T. and Tanaka, K.: A Configurator to Optimize Soft Processors for FPGA, *Proc. 77th National Convention of IPSJ*, pp.23–24 (2015). (In Japanese)

Other Research Publications

[14] Miyauchi, T. and Tanaka, K.: Solving Slitherlink with FPGA and SMT Solver, *Journal of Information Processing*, Vol.28, pp.959–969 (2020). (with review)

[15] Miyauchi, T. and Tanaka, K.: A Solution to Slitherlink Puzzles Using FPGA, *Proc. The 32nd International Conference on Computers and Their Application*, pp.77–83 (2017). (with review)

[16] Miyauchi, T. and Tanaka, K.: A Solution to the Slitherlink Puzzle Using SMT Solver, *Proc. The 22nd Game Programming Workshop 2017*, pp.111–118 (2017). (In Japanese) (with review)