Paper: rb17-2-2285; Feb 28, 2005

Responsive Multithreaded Processor for Distributed Real-Time Systems

Nobuyuki Yamasaki

Department of Information and Computer Science, Faculty of Science and Technology, Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama 223-8522, Japan
E-mail: yamasaki@ics.keio.ac.jp
[Received October 24, 2004; accepted January 6, 2005]

The Responsive MultiThreaded (RMT) Processor is a system LSI that integrates almost all functions for parallel/distributed real-time systems including robots, intelligent rooms/buildings, ubiquitous computing systems, and amusement systems. Concretely, the RMT Processor integrates real-time processing (RMT Processing Unit), real-time communication (Responsive Link II), computer I/O peripherals (DDR SDRAM I/Fs, DMAC, PCI-X, USB2.0, IEEE1394, etc.), and control I/O peripherals (PWM generators, pulse counters, etc.). The RMT Processor, with a design rule of 0.13μm CMOS Cu 1P8M and a die size 10.0mm square, was fabricated by TSMC. The RMT Processing Unit (RMT PU) executes eight prioritized threads simultaneously using fine-grained multithreading based on priority, called the RMT architecture. Priority of real-time systems is introduced into all functional units, including cache, fetch, and execution, so the RMT PU guarantees real-time execution of prioritized threads. If resource conflicts occur at functional units, higher priority threads overtake lower priority threads. Flexible powerful vector operation units for multimedia processing are also designed. System designers use on-chip functions easily by connecting required I/Os to this chip and the designers realize distributed control by connecting several RMT Processors with their own functions via Responsive Link II.

Keywords: real-time, distributed control, Responsive Link, SoC, RMT

1. Introduction

Parallel/distributed processing is required to control high-performance and/or multi-functional robots in real-time, especially in designing very large-scale systems not controllable by a single processor, or control systems in which many sensors and actuators are spread widely. Distributed control is less widely used with embedded control than in the field of information processing.

We designed the Responsive MultiThreaded (RMT) Processor, which is a system LSI, to realize distributed real-time systems easily. The RMT Processor integrates real-time processing (RMT Processing Unit), real-time communication (Responsive Link II), computer peripherals (PCI-X, USB2.0, IEEE1394, etc.), and control peripherals (Pulse Width Modulation (PWM) generators, pulse counters, etc.).

Because the RMT Processor is a very large-scale system LSI (exceeding 14Mgates), we focus on RMT Processing Unit (RMT PU) architecture. The RMT Processor design concept, the real-time processing concept by hardware, a bus, and power management are described in [1].

A register set to execute a thread is called a (hardware) context. A context consists of general-purpose (GP) registers, floating point (FP) registers, status registers, and a program counter (PC). The RMT PU has eight hardware contexts, so eight threads with 256-level priority run simultaneously. Software such as a real-time scheduler sets the priority for each thread.

The fetch mechanism fetches eight instructions of the higher priority thread per clock cycle from the i-cache. If it cannot fetch the highest priority thread because of a cache miss, a branch prediction miss, etc., it fetches instructions from the next higher priority threads in priority order. Whenever higher priority threads finish execution, lower priority threads begin execution. Execution mechanisms execute threads based on priority out of order using prioritized renaming buffers, prioritized reservation stations, prioritized reorder buffers, etc.

When the number of threads is less than or equal to eight and a static real-time scheduling algorithm including Rate Monotonic (RM) is used, the RMT PU executes these threads in real-time by hardware alone and no software scheduler is needed. To execute more threads in real-time by hardware and to cope with a dynamic real-time scheduling algorithm including EDF, the on-chip context cache that can save 32 thread contexts including GP registers, FP registers, status registers, and PCs, is designed and implemented. Only four clock cycles are needed to switch (swap) thread contexts between a context and the context cache, greatly reducing context switching overhead.

Since current real-time applications require high computing performance for multimedia processing, including image processing and voice processing, flexible powerful

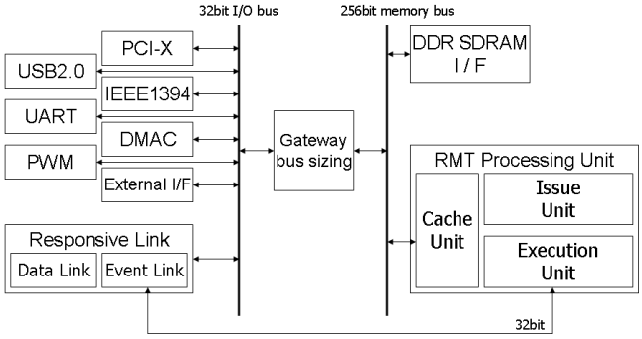

Fig. 1. Block diagram of RMT Processor.

vector operation units are designed for multimedia processing. As multiple threads are executed in parallel on the RMT PU, some require simultaneous vector operations, so vector registers are shared by multiple threads efficiently by reserving the size required for the executing vector operation. A FP vector unit executes four 64-bit IEEE754 FP operations and an integer vector unit executes eight 32-bit integer operations per clock cycle. Two FP vector units that share 512-entry 64-bit FP vector registers and two integer vector units that share the 512-entry 32-bit integer vector registers are implemented. Each vector unit is used independently by different threads simultaneously. A single thread can also use all vector units at the same time.

The RMT PU executes real-time tasks prioritized by a real-time scheduler, making the time granularity of real-time processing finer.

2. RMT Processor Overview

The Responsive Processor, an earlier version of the RMT Processor, has real-time communication mechanism [2] but not real-time processing by hardware, but the RMT Processor has both.

The RMT Processor (**Fig.1**) integrates the following onto one chip (system-on-a-chip) so that it can be widely used in embedded systems.

- Real-time processing: RMT PU
- Real-time communication: Responsive Link II
- Computer peripherals: PCI-X, USB2.0, IEEE1394, DDR SDRAM I/Fs, DMACs, RS-232C, etc.
- Control peripherals: PWM Generators, Pulse Counters, etc.

System designers can use on-chip various functions easily by connecting required I/Os to this chip, and the designers can realize distributed control by connecting several RMT Processors with their own functions via Responsive Link II.

While an internal vector unit executes image processing, for example, captured by a digital camera connected to USB2.0 or IEEE1394 in real-time, pulse counters and PWM generators are controlled by the processing result, as are corresponding actuators.

3. Responsive Link

The RMT Processor integrates Responsive Link II as real-time communication. Responsive Link II is based on the real-time communication standard Responsive Link [2]. The real-time communication architecture of Responsive Link II is designed for distributed real-time systems as follows:

- Separation of data transmission for soft real-time and event transmission for hard real-time
- Fixed packet size: 64-byte data and 16-byte event
- Point-to-point serial link
- Independent routing of the data link and the event link
- Priority-based packet overtaking. The packet with higher priority overtakes packets with lower priority at each node.
- Packet acceleration/deceleration using priority replacement. Packet priority can be replaced by a new priority at each node to accelerate/decelerate packets under distributed control.
- Prioritized routing. When multiple packets with different priority levels are sent to the same destination, a different route can be set to realize exclusive communication links or detours.
- Variable link speed (800, 400, 200, 100, 50, 25, 12.5Mbaud)
- Plug & Play
- Topology-free
- Low latency

Responsive Link II realizes flexible real-time communication based on these features. Maximum one-way communication speed is 800Mbaud. Communication speed can be changed dynamically to reduce power consumption. Five sets of Responsive Link II are implemented on the RMT Processor, one set connected directly to the RMT PU and four sets arranged on the chip. Since a set of Responsive Link II consists of an event link and a data link, each link is full-duplex, and there are five sets of Responsive Link II on the chip, the total switching speed of Responsive Link II per chip is 16Gbaud/chip.

Responsive Link II is connected to the RMT PU two ways, – with Responsive Link II treated as a normal 32-bit I/O peripheral, and with Responsive Link II treated as a RMT PU internal register file. By using a DMAC and dual port memory inside Responsive Link II, packets are

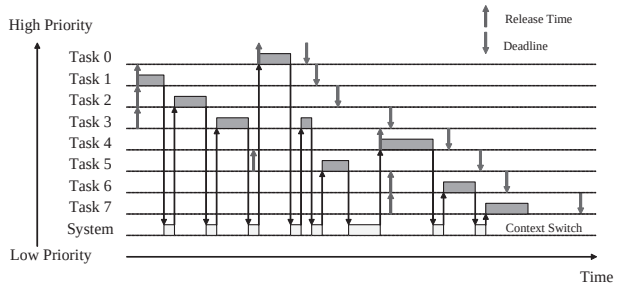

Fig. 2. Real-time execution on a single processor.

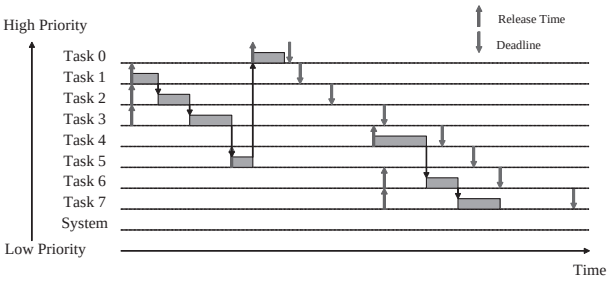

Fig. 3. Real-time execution on a single pipeline multithreaded processor with priority.

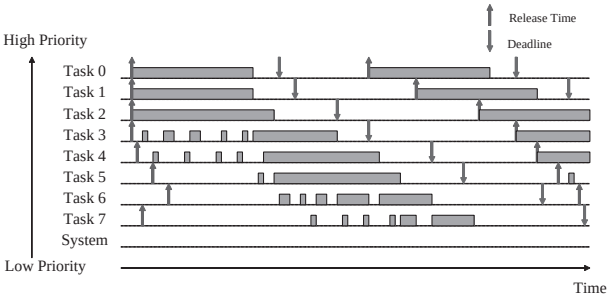

Fig. 4. Real-time execution on RMT PU, a multiple pipeline multithreaded processor with priority.

automatically transmitted and received by hardware; or with event links required to be low latency connected to the internal processor bus of the RMT PU directly. Since the Responsive Link II event link is treated as a special internal register file, the RMT PU transmits and receives event packets by writing and reading the internal register file, and realizing very low-latency communication and inter-processor shared registers easily.

Responsive Link has been standardized as the IPSJ Trial Standard (IPSJ-TS 0006:2003), the domestic standard in Japan [3]. Responsive Link is also being standardized by ISO/IEC JTC1 SC25. Once this international standardization is realized, real-time communication among different systems will be realized, together with large-scale distributed control systems.

4. Design Policy Realizing Real-Time Execution

Figure 2 shows real-time execution scheduled by the EDF scheduler on a single processor. When an operating system switches an execution task, it saves the hardware context of the task in memory, then restores the new context of the next executing task to the processor. Such frequent context switching causes a serious problem in real-time systems.

To solve context switch overhead and bound time required to switch a context, we apply a multithreaded mechanism with priority to real-time processing.

First, multiple hardware contexts are implemented on a chip as in a single pipeline multithreaded processor. Tasks prioritized by a real-time scheduler are stored in the hardware contexts and executed by a multithreaded mechanism with priority. Higher priority tasks are fetched, issued, and executed at each pipe stage based on priority. If enough contexts are implemented on the chip and the number of tasks is fewer than the number of implemented contexts, the tasks are executed in real-time by the multithreaded mechanism with priority without context switching. **Fig.3** shows a real-time execution on the single pipeline multithreaded processor with priority. Tasks are executed in priority order. Lower priority tasks are kept waiting until higher tasks have finished execution even if lower tasks are ready to run, the same as conventional by using a software scheduler and a context switch (**Fig.2**). In brief, each software context switch is converted to prioritized multithreading by hardware. Since no software context switch is needed, real-time task schedulability is improved.

Then, we apply priority to a multiple pipeline multithreaded processor such as a Simultaneous MultiThreading (SMT) [4, 5] processor. If priority is introduced to the SMT processor, prioritized tasks are executed in priority order similar to the case of the single pipeline multithreaded processor with priority, so real-time execution is also realized. In brief, all software context switches are converted to prioritized SMT by hardware, i.e., Responsive MultiThreaded (RMT) execution, with both processor utilisation and task schedulability improved because of RMT execution.

We designed the RMT PU so that it could execute prioritized threads in real-time (**Fig.4**), e.g., some higher threads run simultaneously in priority order. We also call this architecture Responsive MultiThreaded (RMT) architecture.

5. Related Work

A real-time scheduling policy is applied to select decoded instructions in a Komodo multithreaded microcontroller [6] to reduce context switching overhead. Improving processor utilisation improves total system performance, but thread performance is low, because the Komodo micro-controller pipeline consists of a single four-stage pipeline. Soft real-time applications including im-

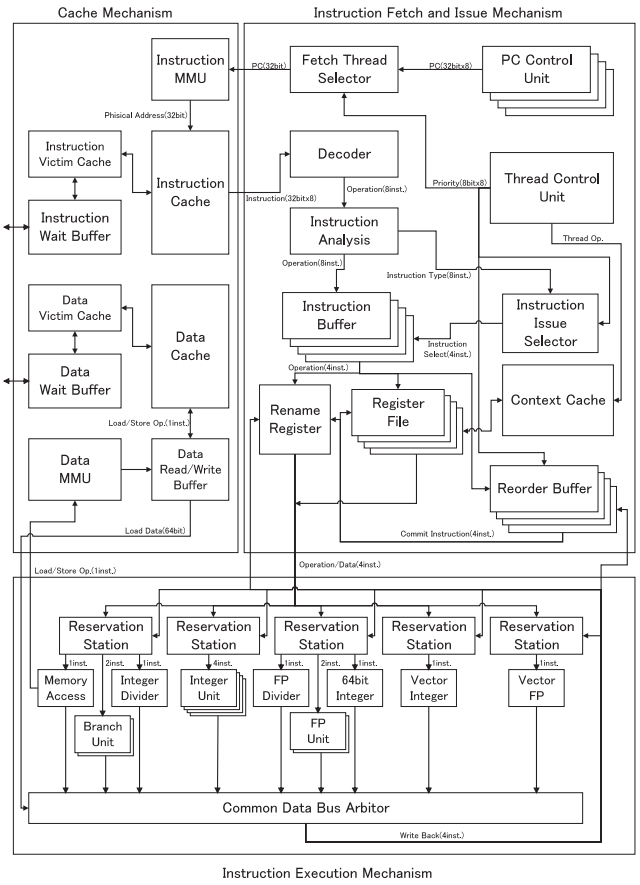

Fig. 5. Block diagram of RMT PU.

age processing require high performance computing, so the performance of a single thread must be high.

Priority is introduced in the selection of fetch threads in a SMT processor, so foreground thread performance while other threads are running is not lower than for single thread execution [7], thereby improving whole-system performance.

Real-time scheduling by software on an SMT processor is detailed in [8]. This research focuses on soft real-time scheduling for simultaneous execution threads and their shared resources on the SMT processor.

6. RMT PU Design

6.1. Pipeline

The RMT PU is designed to execute prioritized threads in real-time without software context switching. When multiple threads run in parallel, resource conflicts including functional units and caches among these threads may occur. The RMT PU gives conflicted resources to higher priority instructions. Priority is assigned by a real-time operating system.

Figure 5 shows the RMT PU block diagram. **Fig.6** shows pipeline stages. Each pipe stage is processed as follows:

FS	Fetch Thread Select
IF1	Instruction Fetch 1
IF2	Instruction Fetch 2
IF3	Instruction Fetch 3
IA	Instruction Analysis
IS	Issue Instruction Select
REG	Register Renaming and Read
RS1	Reservation Station 1
RS2	Reservation Station 2
EXE	Execution
WB	Write Back
CS	Commit instruction Select
COM	Instruction Commit

Fig. 6. Pipeline stages.

1. The FS stage selects which thread is fetched in the instruction unit based on priority.
2. The IF1 stage translates the address using the i-MMU and reads tag information.
3. The IF2 stage selects the objective way by comparing tags and updates access information.
4. The IF3 stage accesses the i-cache and fetches eight instructions.
5. The IA stage decodes four instructions in parallel, analyzes branches, and stores the instructions in the instruction buffer.
6. The IS stage selects which instructions are issued in the instruction buffer. Unselected low priority instructions are kept waiting in the instruction buffer until all higher priority instructions are issued.
7. The REG stage renames destination registers in rename buffers (GP rename buffer and FP rename buffer), accesses source registers, and assigns reorder buffer entries to corresponding instructions.
8. The RS1 stage stores the instructions in corresponding reservation stations.
9. The RS2 stage tests whether instructions are ready and selects which instructions are to be executed among ready instructions based on priority.
10. The EXE stage executes operations.
11. The WB stage writes finished instructions back to rename and reorder buffers, and writes results back to the rename buffer.
12. The CS stage selects instructions among instructions ready to be committed based on priority.

Table 1. Instruction fetch and issue overview.

Contexts (register sets)	8
Instruction fetch	8
Instruction decode	8
Instruction issue	4
32-bit integer registers	32 entries/thread
64-bit FP registers	8 entries/thread
Integer rename registers	32 entries
FP rename registers	32 entries
Reorder buffers	16 entries/thread

13. The COM stage commits the instructions. When an instruction writes data to a register file, this stage writes the value of the rename register to the register file including the GP register file and the FP register file.

As noted above, each functional unit at which resource conflicts occur is designed so as to be arbitrated by priority. In this design, hardware is more complex than in [7], but the highest priority thread always gets resources at each unit and continues running to prevent priority inversion. If higher priority threads do not occupy all resources, lower priority threads run using remaining resources. The RMT architecture accurately guarantees the execution time of the prioritized threads. Software scheduling such as in [8] may improve execution time precision of lower priority threads.

6.2. Instruction Issue Mechanism

The instruction issue mechanism issues multiple threads to the instruction execution mechanism. Parameters as shown in **Table 1** are decided by a tradeoff between required hardware resources and silicon area (the number of gates), because the RMT Processor is implemented on an actual VLSI chip, TSMC 0.13μm CMOS 1P8M 10.0mm square chip, by our project as part of a large national project.

This mechanism writes results sent by instruction execution mechanisms to integer rename registers or FP rename registers. If an instruction is committed, the result is only written to the register of the corresponding thread. The result is not written to the reorder buffers.

6.3. Thread Control Unit

Eight contexts are designed and implemented on the RMT PU. If the number of real-time threads is less than or equal to eight, the RMT PU can execute these threads without context switching. If the number of threads exceeds eight, a context switch is required. A software context switch normally takes over 1,000 clock cycles. To reduce context switching overhead, the on-chip context cache which can save 32 thread contexts is designed and implemented on the RMT PU. The RMT Processor designed for embedded control handles 40 threads at a time

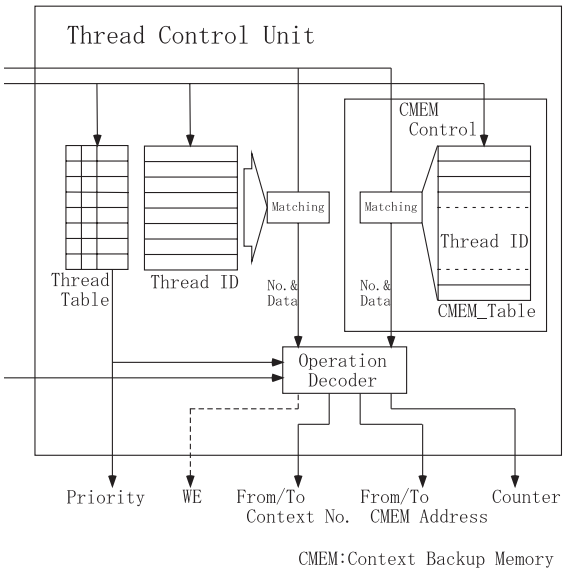

Fig. 7. Thread control unit.

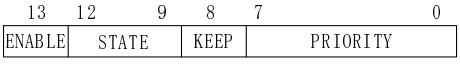

Fig. 8. A line of thread table.

by hardware. The context cache is connected to each register file via wide exclusive buses (256-bit for GP registers and 128-bit for FP registers). The context switch between on-chip contexts and the context cache is realized by hardware to dramatically reduce overhead.

A thread saved in the context cache is called a cached thread, while a thread in one of eight hardware contexts and ready to run is called an active thread.

The thread control unit realizes thread control including context switching (**Fig.7**). Active threads are managed by the thread table in the thread control unit, which generates all thread control signals to the whole processor.

Figure 8 shows a line of the thread table. The ENABLE field indicates whether the active thread is valid. The STATUS field indicates the status of the active thread, such as executing, stopped, saving to the context cache, or restoring from the context cache. The KEEP field indicates whether the active thread is kept in the hardware context. The PRIORITY field indicates the priority level of the thread. As 256-level priority is enough for rate monotonic scheduling [9], 256-level (8-bit) priority is used in the RMT PU.

Special instructions control active threads to access the context cache in the RMT PU. Thread control instructions control each thread using 32-bit unique ID.

Special control instructions for active threads, including create, remove, execute, stop, and set priority, write the thread table to control active threads. Special control instructions for the context cache such as saving an active thread to the context cache, restoring a cached thread from the context cache to the processor, and swapping an active

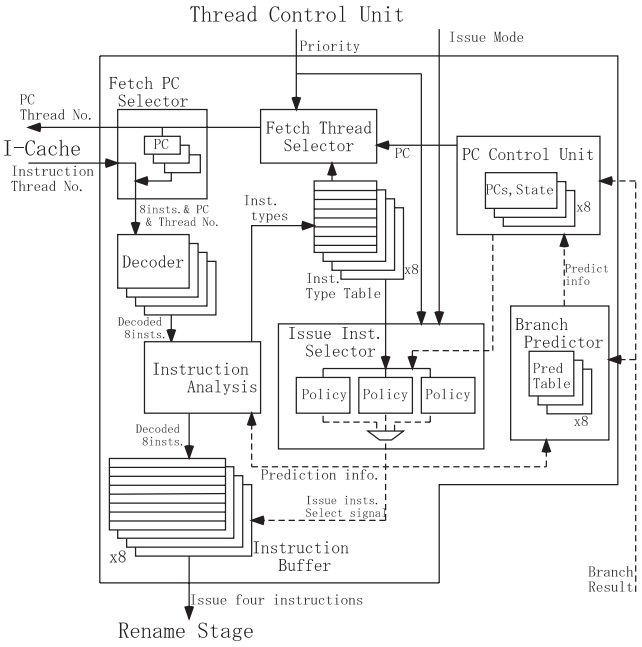

Fig. 9. Instruction issue unit.

thread and a cached thread, are sent to the thread control unit, where the thread control unit searches the cached thread table (CMEM_Table in **Fig.7**), generates the entry number of the context cache, and accesses the context cache.

These control instructions are not speculative and are processed only when the instruction is committed. When a thread control instruction, such as saving another active thread to the context cache or swapping another active thread and a cached thread, is committed, the instruction fetch and issue of the target thread that will be saved or swapped immediately stop. Just executing instructions in each pipe stage only proceed. After executing instructions are all committed, the thread control instruction is executed; the target thread that should be saved is actually saved to the context cache.

6.4. Instruction Issue Unit

Figure 9 diagrams the instruction issue unit.

6.4.1. Instruction Fetch

If an instruction fetch unit is designed to fetch multiple threads at the same time, the throughput of the whole system is improved, but the number of cache ports increases, increasing the size of the i-cache. For example, the number of gates of dual port SRAM is two times larger than single port SRAM, so the RMT PU is designed to fetch eight instructions of a thread per clock cycle based on priority at the FS stage (**Fig.6**) with single port SRAM used as the i-cache.

Generally, the virtual address cache is faster than the physical address cache, but the special physical cache with priority is designed and implemented in the RMT PU for two main reasons:

The RMT Processor is used for distributed real-time control and has many I/O peripherals. Each I/O peripheral is controlled independently by its control thread. These I/O control threads run cyclicly and simultaneously, and account for a large percent of all threads in real-time control systems. In this case, the virtual address cache is not effective.

If a synonym problem could occur, anti-aliasing mechanism should be implemented by hardware. Since 8-way set associativity is desirable at least because of 8-way multithreading, anti-aliasing mechanism increases hardware.

The physical address cache with priority is thus designed and implemented in the RMT PU.

The instruction MMU is located between the processing unit and the instruction cache, so one additional clock cycle is required for an instruction fetch. Thread selection requires a pipe stage (FS stage), and MMU translation requires a pipe stage (IF1 stage). To hide the latency, a speculative fetch unit is designed while the IF3 stage gets instructions from the i-cache. Fetch addresses are predicted by the branch target buffer (BTB).

Instruction fetch processes are shown in **Fig.9** as follows:

- Fetch Thread Selector
  selects threads to be fetched based on priority. If these thread priorities are the same, it fetches the threads by round robin.
- Fetch PC Selector
  keeps fetched PCs, generating the prefetched PCs using the BTB. It also invalidates the fetched instructions when the prefetched PCs become invalid.
- Decoder
  decodes instructions at the first half of the IA stage.
- Instruction Analysis unit
  analyzes the relationship among four of eight fetched instructions, deciding the next fetching address and validation of each fetched instruction using both the decoded results and branch prediction. The decoded results are stored in the instruction type table used by the instruction issue selections.
- PC Control Unit
  keeps the next PCs.

6.4.2. Instruction Issue

To avoid blocking the execution of high priority threads when instruction issues of low priority threads are congested, instruction buffers keeping decoded instructions and instruction type tables keeping decoded results are designed. Eight sets of the instruction buffer that keeps 16 instructions per context (thread) are designed. Issuing instructions are selected in the instruction buffer at the IS stage.

The instruction issue selector also selects real-time threads based on priority. Some applications may require

soft real-time processing and high performance by parallel processing, so instruction issue policies for total system performance are designed. These instruction issue policies are selected by software.

One is that instructions of higher priority threads should be issued as much as possible at the instruction issue policy that regards real-time performance as important, called real-time policy. The other is that instructions of different multiple threads should be issued as much as possible at the instruction issue policy that regards total system performance as important, called performance policy. These policies are implemented in the Issue Instruction Selector (**Fig.9**).

Two real-time policies are designed. One is that instructions of the highest priority thread are issued as much as possible. Remaining issue slots are used gradually by the next priority threads. This policy keeps real-time performance and improves system throughput. The other is that one or a few issue slot(s) is(are) reserved for a real-time thread. Software (RT-OS) reserves resources for real-time threads under this policy.

Two performance policies are designed. One is that four threads whose instructions can be issued are selected, and four sets of an instruction of each thread are issued. Under this policy, three performance policies are designed:

1. Four threads are selected based on priority.
2. The priority of the thread that the number of executing instructions are large is lowered.
3. The priority of the thread executed speculatively by branch prediction is lowered.

The other is that issue slots are selected by using round robin without priority.

Instruction issue is shown in **Fig.9** as follows:

- Instruction Type Table
  keeps analyzed information on decoded instructions such as whether the instruction is valid, whether the instruction is fetched based on branch prediction, the current depth of prediction, whether the instruction is special. Whenever an instruction is issued, its table entry is invalidated. If the branch result is decided, the branch information is stored in the table.
- Instruction Buffers
  keep decoded instructions.
- Issue Instruction Selector unit
  selects instructions based on the instruction issue policy.

6.4.3. Interrupt Unit

The RMT Processor integrates many I/O peripherals. External events are notified by interrupts in embedded systems. It is very important for real-time systems to shorten and fix the response time against external events.

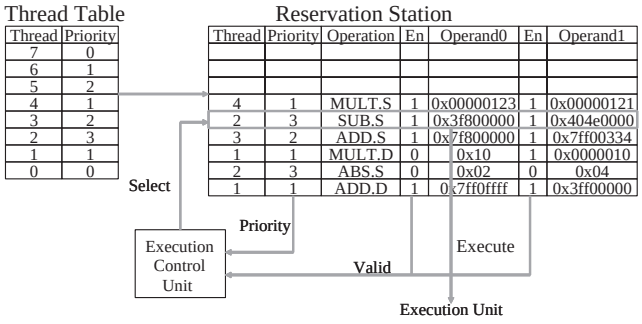

Fig. 10. Reservation station with priority.

Therefore a special interrupt unit is designed so as to reduce the interrupt overhead and process interrupts effectively. Each IRL (Interrupt Level), which is 32-level, can be assigned to a specific thread that processes the interrupt. This IRL assignment is set to the status register of each thread. An active thread waked up by an interrupt is waiting for the interrupt on a context. When an interrupt occurs, the corresponding active thread begins to run immediately. This dramatically reduces the response time.

6.5. Instruction Execution

6.5.1. Reservation Station

Each execution unit has a reservation station that keeps operations, operands, thread IDs, and priority (**Fig.10**). The priority of an instruction is updated by the thread control table at each clock cycle. An execution control unit tests which instruction in the reservation station can be executed. If multiple instructions can be executed, the highest priority instruction is selected.

6.5.2. Soft Real-Time Processing

Instruction execution units achieve the high computing performance required for soft real-time processing such as image processing. Soft real-time processing data has enough data parallelism and is processed by the same operations, so the Vector Integer unit (VINT) and Vector FP unit (VFP) are designed as shown in **Fig.1**. The latency of vector operations is hidden effectively by multithreading in the RMT PU.

6.6. Vector Processing Unit

Figure 11 diagrams a vector processing unit. A vector control unit calculates the address for accessing to a vector register, reserving and releasing vector registers, and processing compound operation instructions. To execute vector operations of multiple threads in parallel, a vector processing unit has two operation pipelines; two vector execution units share a vector register file. The operation pipeline executes many vector elements at a clock cycle by multiple operation units. A vector integer unit processes eight 32-bit elements at a clock cycle. A vector floating-point unit also processes four 64-bit elements at a

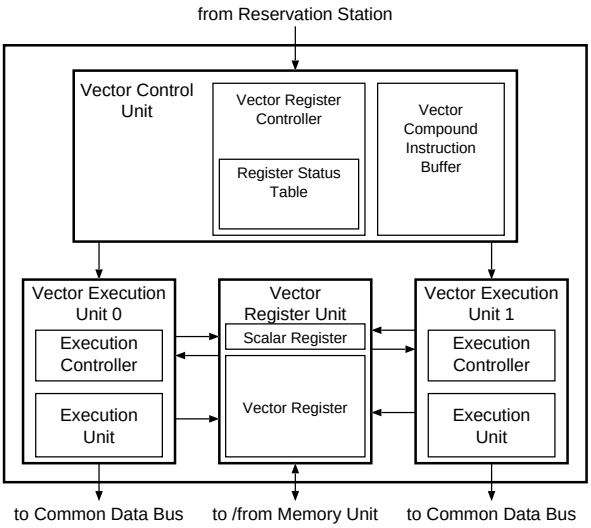

Fig. 11. Vector processing unit.

6	5:4	3:0
Busy	Address	Mode

Fig. 12. Format of register status table.

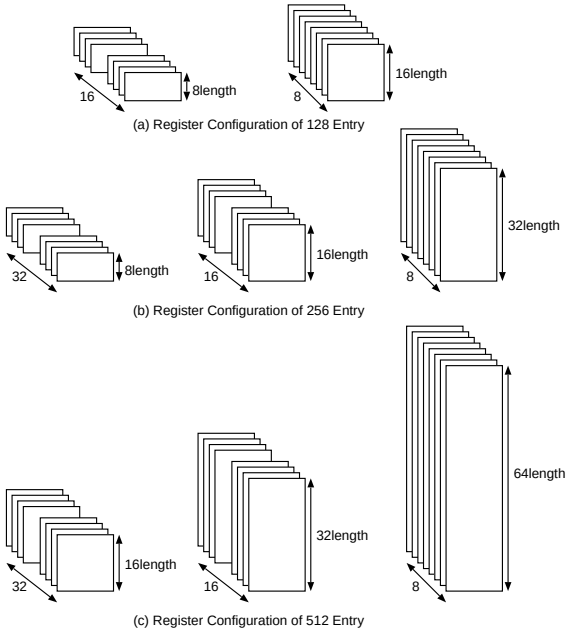

Fig. 13. Vector register file configuration.

clock cycle. As two integer vector units and two FP vector units are implemented on RMT PU, sixteen 32-bit integer elements and eight 64-bit FP elements can be executed simultaneously at a clock cycle.

6.6.1. Reserving and Releasing Vector Registers

Multiple threads may use a vector register file at a time, because the RMT PU is a multithreaded processor, so the shared vector register files among multiple threads that execute vector operations are implemented on the RMT PU.

Each thread reserves a part of the vector register file by a vector register reserve instruction before it executes the vector operation. The reserved part of the vector register file is released by a vector register release instruction after the thread finishes vector operations so that other threads can reserve the vector register file efficiently and execute vector operations.

The configuration of a vector register file, such as vector length and the number of registers, depends on applications. A thread (application) must assign the suitable size of vector registers to share a vector register file efficiently.

When each thread reserves a part of the vector register file, it specifies the required size and the vector length. If the specified size of the vector register file can be reserved, the reserve operation is accepted and part of the vector register file is assigned to the thread. If the vector register file remaining is not enough, the reserve operation fails. The configuration of the vector register file, which contains the assigned area, the vector length, and the register size, is saved in a register status table (**Fig.12**).

When a vector operation is executed, the vector control unit calculates the effective address of the vector register by using the decoded register ID and the configuration information in this table.

A vector integer unit has 512-entry 32-bit elements in a vector integer register file. A vector FP unit has 512-entry 64-bit elements in a vector FP register file. Each thread shares these 512 elements and executes vector operations.

When a thread reserves part of the vector register file, it specifies the required register size and vector length. If variable size is specifiable, transistors used for allocating logic increase, and fragmentation occurs if many threads reserve and release vector registers repeatedly. Allocating units for vector registers are designed so that each thread must choose from fixed configurations (**Fig.13**).

The RMT PU prepares two instructions to execute reserving and releasing of vector registers. The VRES instruction executes reserving part of a vector register file, and specifies the configuration described previously as an operand. The VREL instruction executes releasing the reserved part of the vector register file.

6.6.2. Compound Operation

Many soft real-time applications repeat the same operations, including the multiply-add operation. The issue rate of lower priority threads that executes soft real-time applications may become lower. A compound operation is designed so that programmers can define a series of vector operations performed repeatedly. A compound operation, which is a long latency instruction, executes multiple operations at a time. The number of scalar instructions of the program is reduced to control vector units and to execute vector instructions, also increasing vector unit utilisation.

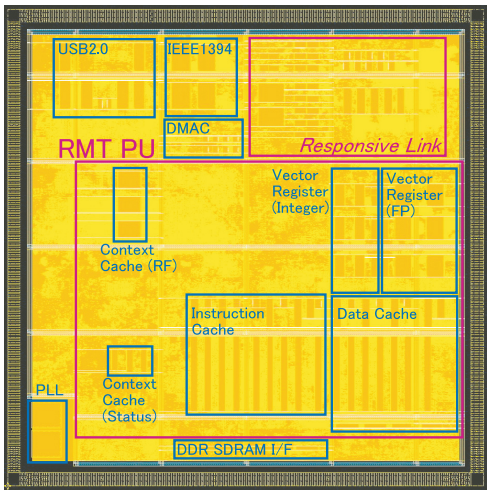

Fig. 14. RMT Processor layout.

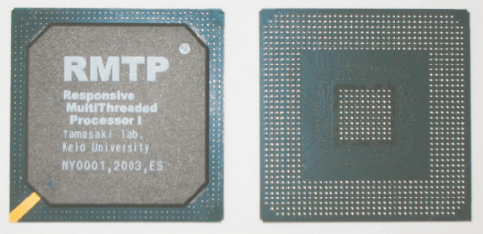

Fig. 15. RMT Processor package.

The priority of many soft real-time threads that use vector operations is low, because its deadline or cycle time is longer than that of hard real-time threads that control sensors and actuators, so compound operation improves the performance of soft real-time threads whose instruction issue rate is low, because compound operation executes many vector operations at once, such as whole inner loop.

7. Fabrication

We designed the RMT Processor from front-end to back-end design, designing and verifying the mask pattern (GDSII). TSMC fabricated the actual chip.

- Manufacturer: TSMC
- Process: 0.13μm CMOS 8-layered Cu wiring
- Gates : 14M
- Voltage:
  - Core: 1.0V
  - I/O: 2.5V
- Die size: 10.0mm$\times$10.0mm
- Chip size: 4.0cm$\times$4.0cm BGA (**Fig.15**)

Table 2. Context switch.

Software	590 clocks	(save: 277 + restore: 313)
Hardware	4 clocks	

Figure 14 shows the RMT Processor layout. Responsive Link II, which has 600kgates, is at upper right. The RMT PU, which has 6Mgates, is at the center of the chip.

8. Development Environment

We developed cross-development tools including a C compiler and an assembler based on GNU tools.

These tools are available to the public on our website [10], together with pamphlets, manuals, circuit diagrams of control boards, and a boot strap.

9. Evaluation

9.1. Context Switch

We evaluate context switch overhead. The software context switch means that software (OS) saves a context to memory and restores a saved context from memory to the register file. The hardware context switch means that hardware swaps an active context and a cached context by the thread swap instruction (SWAPTH instruction) of the RMT PU. **Table 2** shows the costs of the context switch.

The RMT PU provides very fast context switching by hardware. New operating systems, which are not restricted by the frequency of context switching and whose tick is very short, will be developed.

9.2. Real-Time Processing

We evaluated real-time processing, using inverse discrete cosine translation (IDCT) as a benchmark program. Multiple IDCT programs with different priority run on the RMT PU simultaneously and the same data is used. They start at the same time in parallel, so the same cache blocks are accessed by multiple programs at a time – the worst-case scenario. **Fig.16** shows execution time without priority and **Fig.17** execution time with priority. With priority, Thread 0 has the highest priority, Thread 1 the next highest, and Thread 7 the lowest.

Without priority, as the number of threads increases, thread execution time becomes longer because of resource conflicts as the number of threads increases. When the number of threads is one, Thread 0 is executed in 1,100μs. When the number of threads is eight, Thread 0 is executed in 2,250μs, so single thread performance falls to half. Total throughput of all threads is improved four times.

With priority, the execution time of Thread 0 that has the highest priority is almost constant (1,200μs). As the number of threads increases, there is no change in the execution time of Thread 0. Similarly, the execution time of

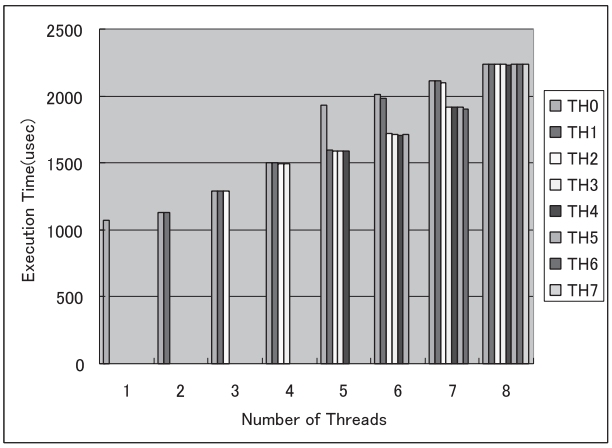

Fig. 16. Execution time without priority.

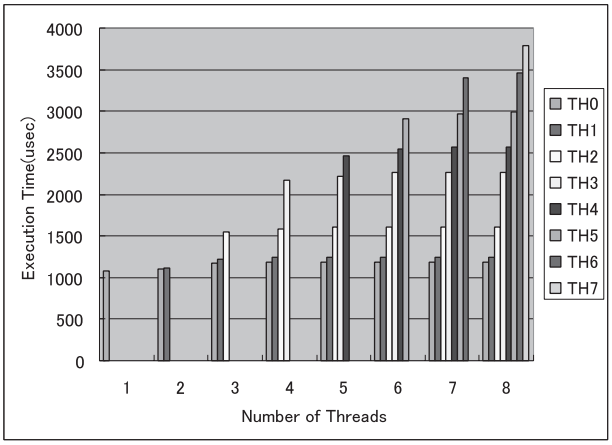

Fig. 17. Execution time with priority.

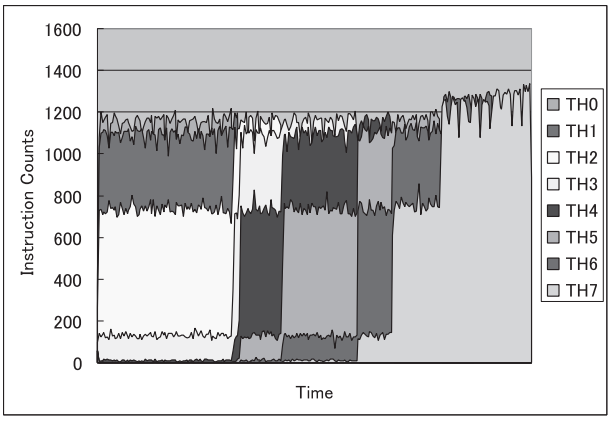

Fig. 18. Real-time execution of each thread with priority.

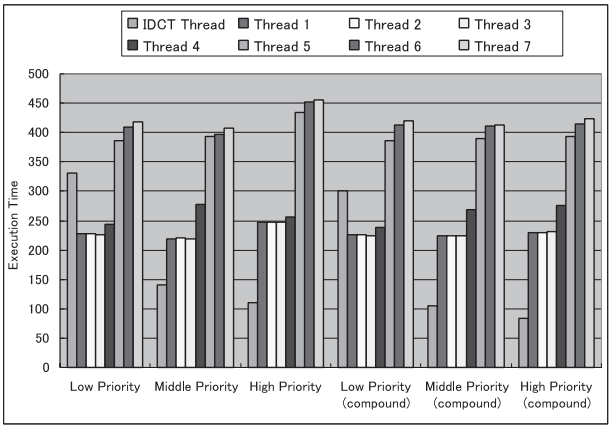

Fig. 19. Execution time of vector unit with priority.

threads with other priority is less than a constant time according to the given priority. For example, the execution time of Thread 2 is less than 1,600μs.

A real-time scheduler assigns lower priority to a thread because its deadline is longer, so the lower priority thread must wait for the finish of higher priority threads, even if the lower priority thread is ready to run. Thread 7 that has the lowest priority is kept waiting on its hardware context until higher priority threads have finished execution. After Thread 0,1,2,3 have finished execution, Thread 7 begins to run simultaneously with other threads including Thread 4,5,6 (**Fig.18**). These mechanisms for real-time execution are realized by hardware.

Figure 18 shows the number of instruction executions per micro-second in eight thread executions with priority (**Fig.17**). The execution rate of the highest priority thread (Thread 0) is highest (1,200times/μs) and the execution rate of the next highest priority thread (Thread 1) is slightly lower than that of Thread 0 (1,100times/μs). Thread 2 runs slower using remaining resources, and thread 3 also occasionally runs using further remaining resources. Thread 4,5,6 7 are not almost running at first. After thread 0 finishes execution, Thread 4 begins to run. The RMT PU can execute threads in real-time as shown in **Fig.4**. The RMT PU guarantees execution time based on the given priority.

9.3. Vector Processing Unit

To evaluate vector processing units, we use a program that has an IDCT of 8×8 array with scalar operations (Thread1-7) and with vector operation (IDCT Thread). The configuration of the vector register specified by the VRES instruction is 128-entry with 8 vector length. **Fig.19** shows the execution time.

Thread 1 has the highest priority among scalar calculation threads (Thread1-7) and Thread 7 has the lowest priority. The IDCT Thread is the vector operation thread. These eight threads are executed at the same time.

Low/middle/high priority (**Fig.19**) means the vector IDCT thread has the lowest/middle/highest priority respectively among eight threads. The execution time of the IDCT thread using the compound operation decreases compared with no using the compound operation. Scalar thread performance (Thread0-7) using compound operation is also improved.

Table 3. Peak performance of RMT PU.

32bit Scalar Integer	1.2GIPS
64bit Scalar Floating Point	600MFLOPS
32bit Vector Integer	9.6GIPS
64bit Vector Floating Point	4.8GFLOPS
Power	Max. 8W

9.4. Peak Performance

Table 3 shows peak performance of the RMT PU. The RMT PU performs simultaneously using multi-threading.

10. Conclusions

We designed and implemented the RMT Processor for distributed real-time control. The RMT Processor integrates real-time processing (RMT PU), real-time communication (Responsive Link II), computer peripherals (DDR SDRAM I/Fs, DMAC, PCI-X, USB2.0, IEEE1394, etc.), and control peripherals (PWM Generators, Pulse Counters, etc.) onto a single VLSI chip.

Priority of real-time systems is introduced into all functional units including cache, fetch, and execution mechanisms, so the RMT PU guarantees the real-time execution of prioritized threads. When the number of threads is less than or equal to eight and a static real-time scheduling algorithm is used, the RMT PU executes these threads in real-time only by hardware, so no software scheduler is needed. To execute more threads in real-time by hardware, the RMT PU has the context cache, reduces the overhead of context switching, and execute multiple threads in real-time to improve real-time schedulability. Vector processing units are designed so that multiple threads shared vector registers efficiently by changing their configuration. Applying compound operation to the low priority thread increases vector processing unit utilization.

The main features of the RMT PU are as follows:

- RMT architecture
  - Eight simultaneous prioritized thread executions
  - Converting the software context switch to RMT execution by hardware
  - Context cache (32 threads)
  - Context switching by hardware
  - 256-level priority
  - Control of all functional units by priority
  - Control of threads by hardware in interrupts
  - Shared registers for multiple threads
  - Synchronization unit for multiple threads
- High-performance vector processors (2-INT, 2-FP)
  - Shared vector units by multiple threads
  - Flexible compound operations defined by software

Flexible real-time processing is achieved by these unique features.

The RMT Processor guarantees the communication time of packets with priority given by a real-time scheduler by Responsive Link II.

The quantum time (tick) is shortened by these mechanisms. Large-scale distributed real-time systems including robots are realized by the RMT Processor.

Since the RMT Processor is very small and easy to connect, it can be easily embedded in the wall, so it is usable as the controller of office automation, home automation, factory automation, intelligent buildings, and ubiquitous computing systems.

We expect the RMT Processor and Responsive Link to be used widely in many systems.

Acknowledgements

This study was conducted through Special Coordination Funds of the Ministry of Education, Culture, Sports, Science and Technology of the Japanese Government. This study was also contributed to by the fund of the CREST, JST.

References:

[1] N. Yamasaki, “Design Concept of Responsive Multithreaded Processor for Distributed Real-Time Control,” Journal of Robotics and Mechatronics, Vol.16, No.2, pp. 194-199, April 2004.

[2] N. Yamasaki, “Responsive Processor for Parallel/Distributed Real-Time Control,” International Conference on Intelligent Robots and Systems, pp. 1238-1244, November 2001.

[3] http://www.itscj.ipsj.or.jp/ipsj-ts/02-06/toc.htm.

[4] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, “Simultaneous multithreading: A platform for next-generation processors,” IEEE Micro, Vol.17, No.5, pp. 12-19, 1997.

[5] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,” In Proceedings of the 23rd Annual International Symposium on Computer Architecture, 1996.

[6] J. Kreuzinger, A. Schulz, M. Preffer, T. Ungerer, and U. Brinkschutle, “Real-Time Scheduling on Multithread Processor,” Real Time Computing Systems and Applications (RTCSA), pp. 155-159, 2000.

[7] S. E. Raasch, and S. K. Reinhardt, “Applications of Thread Prioritization in SMT Processors,” Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC), 1999.

[8] R. Jain, C. J. Hughes, and S. V. Adve, “Soft Real-Time Scheduling on Simultaneous Multithreaded Processors,” In Proceeding of the 23rd IEEE Real-Time Systems Symposium (RTSS), 2002.

[9] J. W. S. Liu, “Real-Time Systems,” Prentice Hall, 2000.

[10] http://www.ny.ics.keio.ac.jp/.

Name:
Nobuyuki Yamasaki

Affiliation:
Associate Professor, Department of Information and Computer Science, Keio University

Address:
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama 223-8522, Japan

Brief Biographical History:
1996- Researcher with Electrotechnical Laboratory, Agency of Industrial Science and Technology, MITI
1997-2000 Researcher with PRESTO (Sakigake21), JST
1998- Research Associate with Faculty of Science and Technology, Keio University
1998-2001 COE Researcher with Electrotechnical Laboratory
2000-2004 Assistant Professor with Faculty of Science and Technology, Keio University
2002- Guest Researcher with Digital Human Laboratory, AIST
2004- Associate Professor with Faculty of Science and Technology, Keio University

Main Works:
• "Responsive Processor for Parallel/Distributed Real-Time Control," IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1238-1244, 2001.

Membership in Learned Societies:
• The Institute of Electrical and Electronics Engineers, Inc. (IEEE)
• The Robotics Society of Japan (RSJ)
• Information Processing Society of Japan (IPSJ)