A NEW PROCESSOR ARCHITECTURE FOR REAL-TIME SYSTEMS

Masato Uchiyama, Tatsuo Itou, Junichi Sato,
Nobuyuki Yamasaki, Yuichiro Anzai

Anzai and Yamasaki Lab.
Faculty of Science and Technology
Keio University
3-14-1, Hiyoshi, Kouhoku-ku,
Yokohama Kanagawa, 223-8522, JAPAN

Abstract: It is difficult to realize a fine-grain real-time system because of overheads caused by restrictions of traditional processor architectures. This paper describes a processor architecture designed for reducing the overheads by applying a simultaneous multithreading (SMT) architecture to real-time systems. The processor has eight hardware contexts in the processor and it can hold 32 contexts in a high-speed on-chip memory for context backup. The processor provides fast context switching capability up to 39 threads. The processor has a mechanism which selects instructions in an issue stage of the SMT architecture, based on priority. Because the mechanism executes several threads in parallel, the processor can improve the total throughput and can provide flexible thread scheduling. Copyright ©IFAC 2001

Keywords: Real-time, Processor, Parallel processing, Priority

1. INTRODUCTION

Nowadays, many real-time systems are used widely in various fields. However, there are many problems caused by scheduling overheads, frequent context switching, etc. in current real-time systems. In order to realize a fine-grain real-time system, it is necessary to reduce the overhead of context switching and to guarantee to execute a thread with high priority by improving processor architecture.

Hardware multithreading mechanisms reduce the overhead of context switching. High-speed on-chip memory for context backup provides the fast context switching. It is considered that the total throughput can be improved by the adoption of a Simultaneous Multithreading (SMT) architecture (Tullser et al., 1995) that fuses a superscalar architecture with a multithreading architecture. This can execute eight threads in parallel. In order to guarantee the execution of a thread with high priority, priority-based arbitration is introduced to deal with various access conflicts.

2. BACKGROUND

2.1 Related Work

A processor architecture for real-time systems (Uchiyama et al., 2000) was proposed in our laboratory. The processor adopted Bock...
Multithreading (Gauthier et al., 1997) to support real-time operations. The processor has four pipelines and shared functional units (FUs). Eight threads are held in the processor. These threads can be assigned to each of the four pipelines. However, its performance is not good when there are few threads, and it needs a special cache system. I demands four read ports of a cache tag table and an instruction cache for itself.

2.2 Multithreading

On the traditional single-thread architecture (e.g., Superscalar), processor utilization is reduced by a dependency of instructions and long latency instructions. Since a multithreaded processor can switch contexts very fast, it can execute a thread while other threads wait for some operations. Thus processor utilization and total throughput can be improved.

When an interrupt occurs in a real-time system, the processor can reduce the overhead of context switching, and can reduce the response time. Of course some context switching occurs in the case of normal thread scheduling. The scheduling time is reduced by the mechanism. The previous study has adopted these mechanisms for real-time systems.

2.3 SMT Processor

SMT (Tullsen et al., 1995) is a technique that fuses a superscalar architecture with a fine-grain multithreading. Various threads can issue their instructions at each clock cycle in an SMT processor. Several threads can be executed in parallel so that, while a long latency operation is waiting, instructions of other threads can be executed. So the SMT processor can achieve high total throughput of all threads. Flexible systems can be constructed by executing various threads including a scheduler, an interrupt handler, real-time threads, normal threads, etc. in parallel. The SMT processor can achieve the same average performance as a superscalar processor, even if there is only one thread.

2.4 Priority

Issue instructions are selected from the instructions of several threads at each clock cycle in an SMT processor. The threads of real-time systems have priority and are scheduled according to priority. It is considered that if priorities are introduced to the issue instruction selection, the selection mechanism will become useful for real-time systems.

When there is a shared resource (like an SMT processor), it causes a situation where several threads request the resource at the same time. In this situation, an execution time of the thread that is selected earlier is short and the execution time of the thread that is selected later is long. This case also needs arbitration based on priorities.

2.5 Cost

Recently, the number of transistors on a chip is increasing fast, and it is expected to continue in future. It is preferable to design a processor architecture to make good use of extra transistors to gain performance, than to cut down the total number of transistors. The multithreading is typical of an architecture that gain performance by using many transistors.

3. APPROACH

The processor proposed in this paper is based on the SMT architecture for real-time systems. The processor has several advantages. First, it provides fast context switching capability, when an interrupt occurs or when thread scheduling is executed. Second, it executes several threads including a scheduler, real-time threads in parallel. Third, it can improve total throughput.

In the processor, if a conflict that requires shared resources occurs, then an arbitration of requests is based on priority of a real-time system. When it selects issue instructions, the selection mechanism introduces a priority of a real-time system with processor statements. These two mechanisms can help the real-time capability.

The processor is also designed for multimedia extension. The processor also integrates Responsive Link (Yamashita and Matsu, 1997) that is a real-time network standard in the future, then it will be possible to use it for various real-time systems.

4. DESIGN

4.1 Overall

The instruction set architecture is compatible with MIPS in the processor, because of its simplicity and its applicability to embedded systems. Some additional instructions are added for thread control and synchronization.

A DDR-SDRAM (Double Data Rate SDRAM) interface with 128-bit width is designed for the main memory.
Fig. 1 Overview of Processor

IF1: 1. Instruction Fetch Stage 1
IF2: 2. Instruction Fetch Stage 2
IA: 3. Instruction Decode and Analysis Stage
IS: 4. Instruction Select Stage
REG: 5. Register Rename Stage
EXE: 6. Execution Stage
WB: 7. Write Back Stage
COM: 8. Commit Stage

Fig. 2 Pipeline

Figure 1 shows an overview of the processor.

The fetch bus is 128 bit wide, so that it can fetch four instructions per clock cycle. Four fetched instructions are decoded in parallel, then held in an instruction buffer. Instructions in the instruction buffer are selected and issued to FUs through renaming buffer, and entered reorder buffer on a parallel of renaming. Instructions are divided broadly into two categories: integer instructions and floating-point instructions. They are executed at each Integer Unit (IU) and Floating-Point Unit (FPU). The IU contains branch units, load/store units that are shared by a FPU, a synchronization unit, and a reservation station of a thread control unit. There are as many reorder buffers as threads. Each reorder buffer commits 2 instructions per clock cycle in order.

The pipeline is divided into eight stages as shown in Figure 2. The operations on each stage are as follows:

IF1: (Instruction Fetch Stage 1) A fetch request is sent. The virtual address is converted to the physical address in the MMU.

IF2: (Instruction Fetch Stage 2) Four instructions are fetched from the instruction cache. If a cache miss occurs, only the thread can’t send a fetch request until the instructions are received.

IA: (Instruction Decode and Analysis Stage) Four fetched instructions are decoded. If they contain a branch or a jump instruction, it predicates branch address and decides the next fetch address. Then it decides whether the instruction is valid or invalid. Decoded instructions are put in the instruction buffer.

IS: (Instruction Select Stage) Four instructions are selected from the instructions that are ready for issue in the instruction buffer. Selected instructions are issued to renaming buffer.

REG: (Register Rename Stage) The destination registers of issued instructions are renamed to GP (32 bit width) or FP (64 bit width) renaming buffer. The renamed number of destination registers are used in the following pipeline stages. The source data of issued instructions are obtained from register file. On a parallel of renaming, issued instructions enter reorder buffer. The entry number of reorder buffer is used for ordering of instructions in part of FUs. Renamed instructions are put in the reservation station of each FU.

EXE: (Execute Stage) The instruction is executed. This stage is not necessarily a single clock cycle. (e.g., a multiplier instruction) The branch instruction is also executed in this stage. Whether the branch is taken or not taken is decided.

WB: (Write Back Stage) Four execution results (e.g. ALU result data, load data, branch signal) are selected. Result data is written back to renaming buffer and finish states are sent to reorder buffer through common data bus.

COM: (Comm: Stage) This stage is divided from WB stage for coherence control. Four instructions are selected and they are committed. Execution results are written back to each register set from renaming
buffer and store data is written to the data cache or the main memory. An exception handling and a recovering of a branch misprediction start as soon as their corresponding instructions are selected to commit in this stage.

The number of clock cycles executed by the IS stage, the REG stage, the EXE stage, and the COM stage are not always constant. The number of clock cycles from the IFI stage to the COM stage is not constant.

4.2 Hardware context

A multithreading architecture provides fast context switching capability with several hardware contexts. The mechanism of a hardware context switching is described next.

A hardware context consists of a general register set and special registers (e.g. Program Counter), and thread status. The processor holds eight hardware contexts and all instructions that these contexts can issue. It is then possible to realize flexible scheduling. It can switch contexts very fast by hardware, but there are only 8 sets of hardware contexts. If the total number of threads is over eight, the cost of context switching for threads in main memory is high, because an access to main memory is caused. Instead it saves the contexts to on-chip memory that it can access faster than the external main memory with very wide bandwidth (over 500-MHz). The processor can save or restore contexts very fast. The backup-memory holds 32 contexts, so it can provide fast context switching capability up to 39 contexts. To distinguish from threads in on-chip memory, a thread in the hardware context is named an active thread.

Hardware contexts are managed with a thread table (Figure 3).

The STATE field indicates a thread state. If the STATE is RUN, the thread is executed. If the STATE is READY, the thread can be executed but it is not executed. If the STATE is TWAIT, the thread is instructed to stop fetch and issue of instructions and it waits for finishing the execution of issued instructions. If the KEEP bit is enabled, the thread can’t be saved to backup memory, then the thread is kept in hardware context. It is used to prevent a context switch from stopping the execution of a high priority thread. The PBASE field holds base priority, and the PRIOR field holds current priority. These two fields and the TIMER field help to thread scheduling by software.

There are also some instructions to manage threads.

<table>
<thead>
<tr>
<th>27</th>
<th>26</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLE</td>
<td>ID</td>
<td>STATE</td>
<td>KEEP</td>
<td>PBASE</td>
<td>PRIOR</td>
<td>TIMER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig.3 Thread table entry

There are instructions to read the thread table, to set the thread states including the KEEP bit, to set the PBASE field and PRIOR field, to set the PC of start point, and to save or restore contexts. The policy of save/restore contexts fundamentally depends on the scheduling policy of the operating system. The response time to an interrupt is important in real-time systems because applications often require to output the result as soon as input from the outside world. A hardware interrupt handler is provided to reduce the response time. As soon as the interrupt handler detects interrupts with a special number set by software, the interrupt handler sends signals to the thread control unit to restore the context of an exception handler.

The thread table is placed in the thread control unit (Figure 1). A bus that transports eight set of priorities is wired to almost all units in the processor from the thread control unit. An instruction does not hold priority but hold only its thread number. When the instruction needs priority, it refers to the bus wired to the thread control unit to get its priority with its thread number. In this way, the instruction is not controlled based on the current priority of the IA stage, but is controlled with the current priority of its thread.

Thread control instructions are executed in the thread control unit, but its reservation station is placed in IU. The instruction is issued to the reservation station similar to other reservation stations in IU. The reservation station is implemented as a FIFO queue and out-of-order execution is prohibited.

4.3 Access control based of priorities

The processor controls instructions according to the priority of their thread. The arbitration of access to a shared resource is based on priority and guaranteeing an execution of the thread with a high priority.

4.3.1 Instruction fetch and issue

The instruction fetch and issue mechanism is implemented in instruction unit (Figure 1). The mechanism consists of two selection pars (Figure 4). In the first selection part, a thread sends a fetch request to the instruction cache is selected at each clock cycle. It is implemented in fetch thread select unit. In the second selection part, four instructions that are issued to FUs are selected at each clock cycle. It is implemented in issue instruction select unit.
The thread that sends a request is selected from all of the threads that are ready for a fetch. The selection is basically based on priority. However, when all threads have the same priority, it is considered that the performance can be improved with another selection mechanism that takes account of whether the fetch address of each thread is a result of the branch prediction or is unrelated to branch prediction.

A sequence of four instructions is fetched at a time with 128-bit width. The fetch address of the next clock cycle can’t be decided until it is known whether the fetched four instructions include branch instruction or not and the branch address has been predicted. Consequently, the interval from the previous fetch request of a thread to the next fetch request of the same thread is four clock cycles. If there are few threads, the rate of instruction fetch is not so good. To improve the rate, the fetch mechanism introduces another prediction. The prediction sets like a prefetch. The mechanism is as follows:

1. The thread sends a fetch request as soon as it is signaled that the previously request hits the instruction cache or that the previous requested data will have been received at the next clock cycle. The fetch address is generated by incrementing the PC.

2. When the previously requested instructions are received and analyzed, it is decided whether the predicted instructions are necessary or unnecessary. If the instructions are unnecessary, it sends a cancel signal to the instruction cache.

3. If the cancel signal is not in time and the predicted instructions are received, they are discarded.

A thread can fetch four instructions per two clock cycles with the prediction mechanism.

The number of issue slots per clock cycle is four. The selection policies of instructions which are issued to FUs have various possible patterns. The selection policies can be classified according to how these four slots are assigned to threads. If all four slots are assigned to only a thread and it is switched to an other thread each cycle, the policy is similar to TDM (Time Division Multiplexing) scheduling. If each slot is assigned to a fixed thread, the number of slots that are assigned a thread is decided based on the priority of the thread. Furthermore, it is considered several policies, a policy that a slot is assigned to several threads and they are switched with the TDM scheduling, a policy that a slot is assigned to a fixed thread and the thread that no slots is assigned can issue instructions when an issue slot becomes empty, and so on.

![Fig.4 Instruction fetch and issue mechanism](image)

It is confirmed that the selection policy that takes account of the number of instruction in the reorder buffer is effective (Tullsen et al., 1996).

On the other hand, there is a restriction that any selection mechanism must finish within a clock cycle time. A good performance improvement is expected with a complex selection policy, but its complexity leads to increasing the clock cycle time. It is necessary to make comparisons on real programs in order to select effective selection policies.

4.3.2 Execution of instructions in EXE stage
Instructions in each reservation station are sorted according to their priorities. Then the instruction with the highest priority which is ready to be executed is executed. The instructions are sorted every clock cycle. The mechanism prevents the execution of a thread with a high priority from being disturbed by a thread with a low priority.

4.3.3 Access to memory
The cache subsystem consists of an instruction cache and a data cache. The main memory is a shared resource so there will be conflicts of requests from several threads. So it is necessary to prevent the execution of a thread with a high priority from being disturbed by the execution of a thread with a low priority. As in the reservation station, the prevention is implemented by selecting threads based on priority.

Each thread has an individual virtual address space in the processor. The virtual address is converted to the physical address in the MMU. The MMU has not only addresses but also thread numbers as TLB tags. In the instruction cache, the fetch request is held after an address conversion. Then a conflict of memory accesses is arbitrated. In the data cache, operations are held in the write buffer or the read buffer at load/store unit one of the FUs. If the operation is a read, it looks up in the write buffer whether its read address and its thread number are matched or not matched. If the address and the thread number are
mached, the data is read from the write buffer. Otherwise the operation is held in the read buffer and it sends a read request to the data cache after an address conversion in the MMU. If the operation is a write, the operation is held in the write buffer at once. Operations in the write buffer wait for a signal that is generated in the reorder buffer at the time that instruction is ready to commit. Signaled operations in the write buffer are arbitrated, and the selected operation accesses main memory after an address conversion in the MMU. When cache misses are caused at the instruction cache and at the data cache at the same clock cycle, it is necessary to arbitrate requests. The arbitration is based on priority. If a request from the instruction cache and a request from the data cache have the same priority, the request from the instruction cache is selected first.

The caches have four-way banks and the caches are physical address cache. The throughput of a thread with high priority is improved by introducing a policy of cache bank replacement based on priority.

4.4 Register Set

The processor has eight register sets. A register set consists of 32 GF registers that is 32-bit wide for integer operations, 32 FP registers that is 64-bit wide for floating-point operations, and special registers. These register sets are saved to or restored from the memory for the context backup through a bus that is 512-bit wide, and then it can finish both save and restore operations only for four times of the memory accesses.

4.5 Implementation

The processor has been specified in Verilog-FDL and simulated. It is implemented on a FPGA board to be evaluated with real programs and to be verified in performance. Some software for the processor including a real-time OS has been developed on this board. Finally, it will be implemented in a chip. The processor is intended as a CPU core for a Responsive Processor (Yamazaki and Matsui 1997; that is a System-On-a-Chip for parallel distributed real-time control systems with Responsive Link; a real-time communication protocol.

5. SUMMARY

A processor architecture which introduces several mechanisms for executing real-time systems is described in this paper.

The processor adopts SMT architecture. The processor can switch contexts very fast by holding several threads in the processor as hardware contexts. Thus the processor can reduce the overhead of thread scheduling. The mechanism of instruction fetch and issue can issue instructions flexibly from several threads. The mechanism can improve the total throughput and provide flexible thread scheduling. There are conflicts of accesses to shared resources in various parts of the processor. Arbitrations that are based on priority can prevent the execution of a thread with high priority being disturbed by the execution of a thread with a low priority.

The processor can improve the performance of real-time systems, and it can support new scheduling policies.

REFERENCES


