Outline of OROCHI: A Multiple Instruction Set Executable SMT Processor

Hajime Shimada
Graduate School of Informatics,
Kyoto University
shimada@kuis.kyoto-u.ac.jp

Takashi Shimada, Takekazu Tabata, Toshiaki Kitamura,
Graduate School of Information Sciences,
Hiroshima City University
{takashi, tabata, kitamura}@arch.ce.hiroshima-cu.ac.jp

Tomoya Kojima, Yasuhiko Nakashima
Graduate School of Information Science,
Nara Institute of Science and Technology
{tomoya-k, nakashim}@is.naist.jp

Kenji Kise
Graduate School of Information
Science and Engineering,
Tokyo Institute of Technology
kise@cs.titech.ac.jp

Abstract

In recent years, enjoying multimedia contents with portable devices become popular. These multimedia processing workloads are too heavy workload for a conventional processor so that current portable devices implement additional dedicated processor for multimedia processing. But we have to left conventional processor to execute OS and miscellaneous processing so that this solution enlarges cost, footprint, and power consumption compared to one chip solution.

In this paper, we propose the processor called OROCHI which can execute two instruction sets simultaneously. The processor can execute VLIW instruction set for multimedia processing and conventional instruction set for OS and miscellaneous processing. In this paper, we introduce OROCHI processor which is based on a VLIW pipeline. The processor decodes either of the two instruction sets in its corresponding front end. After that, the processor decomposes and translates the conventional instructions and insert them into available slots in VLIW instructions. By these means, we can successfully unite the two processors of different purposes into one specific processor. As a result, we can reduce hardware cost, footprint, and power consumption to meet the rising demands of portable media processing market.

1 Introduction

In recent years, enjoying high-quality multimedia contents with portable device becomes popular. But decoding high-quality multimedia contents is very heavy workloads so that it incurs a heavy load for the processor. So, the processor which is implemented in such portable devices has to achieve high performance on multimedia processing to suit this trend. If we implement the processor which runs with high clock frequency and supports wide superscalar execution, the problem will be resolved easily. But those processors usually consume much power and it is unacceptable for the portable devices in two points. Firstly, the portable devices are usually implemented in small chassis so that those power-hungry processors are hard to be employed because of heat liberation problem. Secondly, they commonly work on batteries so that the processor also have to reduce power consumption to extend the battery life as far as possible. The power-hungry processors are not suitable for this requirement. So, the processors which are utilized in these devices have to achieve not only high performance but also low power consumption.

Putting emphasis on both performance and battery life, the processor must exploit high instruction throughput with a simple hardware implementation. Considering the heavy weighted multimedia workload in the modern portable devices, a VLIW processor is usually a good candidate since many ILP in multimedia programs can be easily detected by the compiler. Hence utilizing the well designed multimedia related libraries, VLIW processor achieve a good performance. However, respiting its good efficiencies in the media processing area, a VLIW processor is less competitive in the application with few ILP in static. Moreover, the library support for general purpose applications in VLIW is comparatively weak. In most portable device implementations, a general purpose processor is also included to handle the OS and miscellaneous codes. But implementing two processors increases footprint of chips, cost of chips, and power consumption of chips compared to the one chip de-

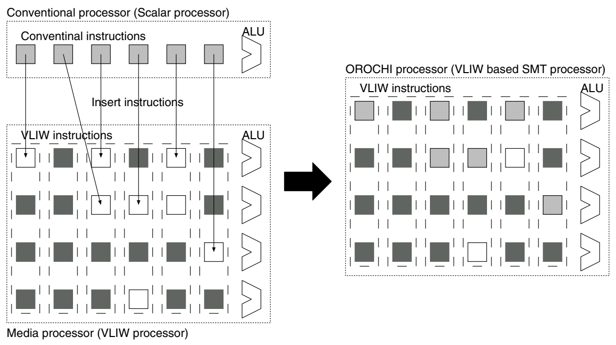
Conventional processor (Scalar processor)
Conventinal instructions
ALU
Insert instructions
VLIW instructions
ALU
Media processor (VLIW processor)
OROCHI processor (VLIW based SMT processor)
VLIW instructions
ALU

Figure 1. Outline of the OROCHI processor execution model (VLIW base).

sign.

To alleviate this problem, we propose a processor called OROCHI which can execute both the conventional instruction set and the VLIW instruction set simultaneously. From the microarchitecture viewpoint, there are various design candidates to implement the idea of OROCHI processor. To consider the OROCHI processor implementation, first of all, we have to choose the base pipeline of the processor. Basically, it can be either VLIW processor pipeline based or superscalar pipeline based. Currently, we are considering both of them in our research. In this paper, we introduce the outline of one OROCHI processor implementation which is based on a VLIW processor pipeline. In this design of the OROCHI processor, the processor executes the instructions of conventional processor by inserting them into available slots in the VLIW instructions after decomposition and translation. The OROCHI processor enables good power performance ratio in multimedia applications and utilizes legacy software resources in one chip. As a result, we can reduce cost, footprint, and power consumption.

In this paper, we introduce outline of the OROCHI processor which we are currently designing. The rest of this paper is organized as follows. Section 2 describes the outline of the OROCHI processor. Section 3 explains more detailed microarchitecture of the OROCHI processor. Finally, Section 4 concludes this paper and describes future works.

2 Outline of the OROCHI Processor

As described in Section 1, there are many design candidates to implement the OROCHI processor. In this paper, we describe the OROCHI processor based on a VLIW pipeline.

2.1 Outline of the OROCHI Processor Based on a VLIW Pipeline

Figure 1 outlines concept of the OROCHI processor based on a VLIW pipeline. The left part of Figure 1 denotes organization of current portable devices which deals with high-quality multimedia contents. It utilizes a conventional processor (scalar processor) and a media processor (VLIW processor) for this purpose. The conventional processor usually executes OS codes and miscellaneous low workload applications. To reduce the develop period of the device, exploiting the conventional processor is very important because there are much inheritance codes and libraries for the conventional processor. On the other hand, the media processor is implemented to accelerate media processing. In usual media processing, there are many much data parallelism in the program execution so that the typical media processor employs instruction set which can easily to exploit data parallelism such as VLIW, SIMD, and so on. In our implementation, we assumed a VLIW instruction set based media processor as shown in Figure 1.

The right part of Figure 1 denotes the organization of our proposal, the OROCHI processor based on the VLIW pipeline. As shown in Figure 1, there are some empty instruction slots in the VLIW instructions. Even if the VLIW processor executes multimedia application which can easily fill instruction slots in the VLIW instructions, there's possibility that the compiler cannot feed the whole instruction slots. So, we thought that if we can effectively insert the instructions of the conventional processor into empty slots in the VLIW instructions, we can integrate the two processors of different purpose into the one processor without performance loss. As shown in right part of Figure 1, the instructions of the conventional processor are inserted into nearest empty instruction slots of the VLIW instructions.

One possible problem for this design is that there may not be sufficient empty slots to insert other instructions under a multimedia workload. However, as in many current VLIW processors, the number of functional units is usually larger than the instruction slots in a VLIW instruction. As an example, Itanium 2 has 11 functional units even if it can issue 6 instructions simultaneously in maximum[1]. This architecture is aimed to reduce NOP operation in the instruction slot of the VLIW instruction. Even if we execute multimedia program, there are possible low parallelism areas in the program. In that area, there's a possibility that we can only fill one or two instruction slot in the VLIW instruction. If the VLIW instruction is too long and it has too many instruction slots, we have to fill many NOP operations into the empty instruction slots correspondingly and it causes low code density. The low code density caused by the NOP instruction increases instruction cache misses so that the processor reduces the number of the instruction slots compared to the number of the functional units. Additionally, many of current VLIW processors have limitations in combining operations or insertion points of operations in the instruction slots. If the processor permits possible combinations of operations and free insertion method, it has to prepare complex data paths which will degrade clock frequency consequentially. To alleviate this problem, the processor limits combination of operations or insertion point of operations in the instruction slots, and this limitation furtherly increases free functional units in the execution. For these two reasons, we can draw a conclusion that we can still find sufficient slots in VLIW instructions to insert other general purpose instructions even under a heavy multimedia workload.

In this way, the VLIW instructions and the conventional instructions are executed simultaneously. Thus, the execution of the OROCHI processor is simultaneous multi-threading or SMT execution. By using an OROCHI processor, the portable device can utilize an unified processors and thus reduces cost, footprint and power consumption.

2.2 Quality of Service in SMT Execution

For the conventional SMT execution, there is usually not a Quality of Service or QoS requirement and the corresponding measurement. Hence, our SMT execution based OROCHI processor does not verify QoS basically. But in many embedded systems including the portable devices, QoS guarantee is one important requirement. For example, a movie usually plays 30 frames per second so that the decoding of each frame must be done in 1/30 second. So, the processor have to guarantee that the frame must be decoded with one thirties seconds. To satisfy this requirement, we have to add an arbitration hardware to support QoS guarantee.

Under typical usage of the OROCHI processor, the processor executes both multimedia processing thread written in the VLIW instruction set and OS thread written in the conventional instruction set simultaneously. From the multimedia processing side, there are many deadlines. The processor has to guarantee that the completion of task before the deadline to meet the media QoS requirement. From the OS side, real time operations are required in many embedded systems. The processor has to response faster than the allowed delay after interrupt occurs. To fulfill these requirements, we are planning to add the hardware mechanism which controls the instruction fetch from threads.

2.3 Current Implementation Plan

Currently, we are designing one implementation of the OROCHI processor to manufacture chips by way of trial. We selected ARMv4 instruction set [2] for conventional in-

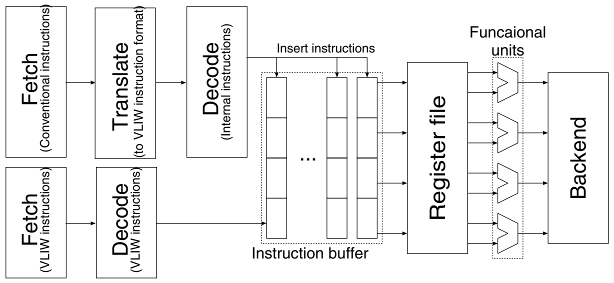

Figure 2. Outline of block diagram.

struction set and FR-V 550 instruction set [3] for VLIW instruction set. In this subsection, we will introduce some notable characteristics of ARMv4 and FR-V 550 instruction set.

1. ARMv4 instruction set

   Each instruction of ARM instruction set is consist of 4 byte length. It employs load/store architecture like RISC instruction set. But on the other hand, it includes complex operation like CISC instruction set. These features are adopted to achieve one of the most important demands in embedded processors; to improve the code density. Such policy is clearly shown in the “multiple load/store instruction” which aims to reduce load/store instructions in spill codes around function call codes. Under this policy, further instruction reduction is achieved by treating program counter as a specific register number 15. By including the register number 15 into “multiple load/store instruction” of spill code around function call codes, we can reduce jump and link instruction and jump register instruction around function call codes. ARM applies flag(sign, zero, carry, and overlow) based conditional execution to all instruction. It can reduce not only branch misprediction penalty but also the number of instructions by eliminating branch instructions in short if-then-else clause.

2. FR-V 550 instruction set

   Each VLIW instruction of FR-V 550 contains 8 operations. From limitation from number of functional units, it can include 4 integer ALU operations in maximum, 4 FP or media ALU operations in maximum, or 2 branch operations in maximum into each VLIW instruction respectively. Similar to current major VLIW processors, FR-V 550 employs many registers (64 entries), conditional execution, and pre-load instructions to exploit ILP in static.

The target devices of the OROCHI processor are portable devices so that we selected the instruction set used in those areas. There are some complicated instructions in ARM processor so that we have to develop a complicated translator for translation. Details of ARM instruction translation are described in Section 3.2.

3 Outline of Microarchitecture

In this section, we talk more detailed microarchitecture of the OROCHI processor which we are currently designing. Firstly, we talk outline of pipeline. After that, we introduce details of some complicated blocks in the pipeline.

3.1 Outline of Pipeline

Figure 2 outlines the block diagram of the OROCHI processor. There are two fetch units in the figure. One of them fetches the conventional instructions and the other fetches the VLIW instructions. After fetching both instructions, the VLIW instructions run through a simple decoder and are enqueued into the instruction queue. In current plan, we treat one slot of the VLIW instruction as an internal instruction. Thus, we don't need to translate the VLIW instructions. Compared to the VLIW instructions, the general purpose instructions of ARM have to pass more complicated logic units which translate them into the internal instructions. According to the complexity of the ARM instruction set, some ARM instructions must be decomposed into several internal instructions like Intel P6 architecture [4] or Netburst architecture [5]. After decomposition, all the translated internal instructions from ARM are inserted into the empty slots of the VLIW instructions in the instruction queue. The internal instructions in the queue will be issued if all of the previous line instructions have been issued. The issued instructions work similarly as normal FR-V 550 instructions —- accessing register files and utilizing functional units for execution. The pipeline of the OROCHI processor is based on the VLIW implementation so that the processor has to stall the internal instruction execution when one of instruction which will be issued simultaneously occurs pipeline stall.

Figure 3 outlines the instruction decomposition and insertion in the OROCHI processor based on FR-V 550 architecture and ARM architecture. The backend pipeline is based on FR-V 550 pipeline, which has 4 integer ALUs, 4 FP or media ALUs, and 2 branch units. A VLIW instruction of FR-V 550 contains 8 internal instructions in maximum. When the VLIW instruction moves into the instruction queue, the internal instructions are decomposed and inserted into the instruction slots which are connected to the suitable functional units. As discussed in its concept, the internal instructions from ARM will occupy the empty slots in the instruction queue after the decoding of VLIW instructions.

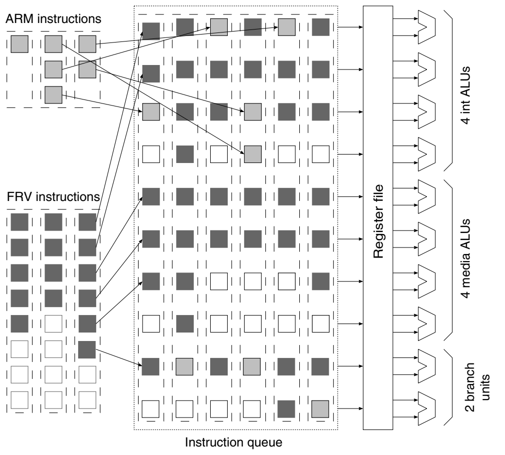

Figure 3. Instruction decomposition and insertion in the OROCHI processor based on FR-V 550 architecture and ARM architecture.

3.2 Decomposition and Translation of ARM Instructions

From the above discussion, one key component in the OROCHI architecture is the instruction translation unit. It decodes ARM instruction for latter insertion. Moreover, the decomposition of some complicated ARM instruction is also handled in this unit.

To achieve insertion easily, we decompose the ARM instructions into simple internal instructions which can be executed in one clock cycle.

3.2.1 Outline of ARM Instruction Set

Figure 4 shows samples of ARM opcodes and operations. To simplify operation notation, we use "+" to represent all ALU operations and "<<" for shifts. Also, we use "load" operation to represent load/store operation.

As shown in Figure 4(a), the ARM instruction set can apply the shift operation before the ALU operation. In our implementation, we divide this kind of instruction into two operations as follows.

1. Shift Src2 by Src3.

2. Add 1. to Src1.

As shown in Figure 4(b), the ARM instruction set can apply the shift operation before effective address generation similar to the ALU operation. Moreover, the ARM instruction set permits a load/store operation to writeback the generated effective address into Src1 or the base register. In this case, we need to divide the load/store operation into a

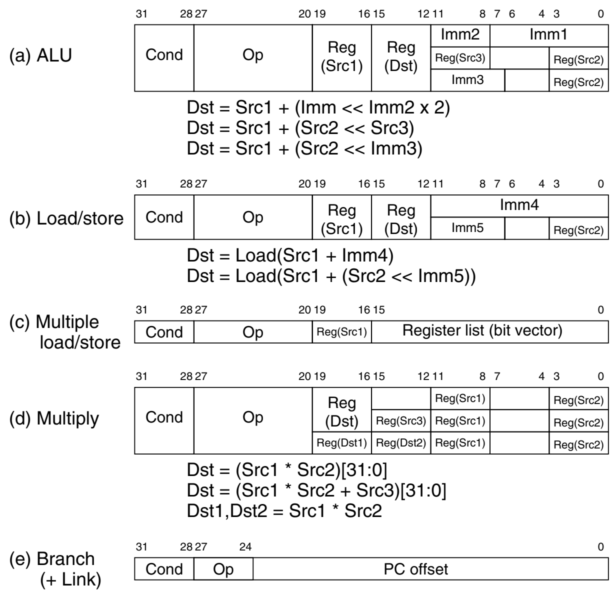

Figure 4. Sample of ARM opcodes and operations.

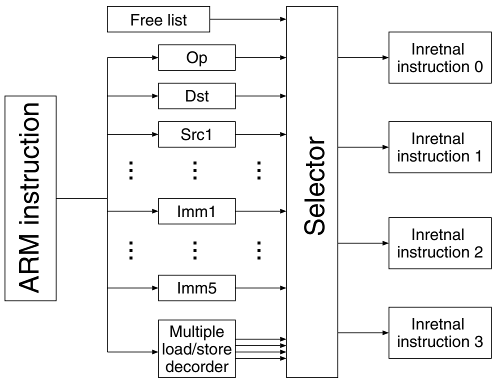

Figure 5. Outline of ARM instruction decomposer and translator.

Src1 update operation and a load/store operation to acquire register write port for updated Src1. Furthermore, the ARM instruction set permits both pre updating and post updating for Src1. Thus, in the worst case, we need to divide one load/store instruction into following three internal instructions.

1. Shift Src2 by Imm5.
2. Add 1. to Src1 and writeback it to the register file.
3. Execute load/store operation with effective address generated by 2.

Moreover, There is more complicated instruction called multiple load/store instruction in the ARM instruction set. The multiple load instruction can load consecutive data from the main memory into several register entries. The register entries which are used for this operation marked with "1" in the register list potion of Figure 4(c). To decompose this instruction, we need to prepare following two instructions for each load/store operation.

1. Update Src1 to generate effective address.
2. Execute load/store with effective address generated by 1 iteratively.

In the worst case, we have to decompose this instruction into more than 30 instruction in maximum.

Figure 4(d) shows multiply instructions of the ARM instruction set. We decided that the processor decomposes a multiply instruction into several partial product (32bit $\times$ 8bit multiply) operations and accumulation of partial product operations. This decomposition is aimed to reduce multiple latency instructions: to simplify instruction scheduling in insertion of the decomposed instruction. We assumed that a 32bit $\times$ 8bit multiply operation will be finished in 1 clock cycle. The multiply instructions are decomposed into several 32bit $\times$ 8bit partial product calculations and several partial product accumulations.

Figure 4(e) shows large PC offset branch of ARM instruction set. As introduced in Section 2.3, the short PC offset branch is achieved by arithmetic operations into the register number 15. But it only permits 8 bit length offset so that we use this instruction to achieve a large offset.

3.2.2 Organization of ARM Instruction Translator and Decomposer

The types internal instruction, which is the target form of ARM instruction translation results are listed in Table 1. By using those internal instructions, the ARM instructions are decomposed and translated with given patterns which are shown in Table 2.

This processing is done in succedent stage of the fetch stage as shown in Figure 2. It is denoted as "Translator" in a generalized OROCHI processor as in Figure 2, and actually also uses part of its logic to perform as a "decomposer" in ARM instruction set. Figure 5 demonstrates the outline of the translator and decomposer which are based on the discussions in Section 3.2.1. We design that the translator and decomposer can output up to 4 internal instructions in each clock cycle. If one ARM instruction is decomposed into more than 4 internal instructions (e.g. multiple load/store),

Table 1. Type of internal instructions

Type	Behavior
E	3 operand style ALU arithmetic (for general arithmetic and logical operation)
S	Shift (logical shift, arithmetic shift, rotate)
M	8 types of 32b * 8b multiple for MAC instruction (combined with “m” internal instruction)
m	Support arithmetic instruction for MAC instruction (absolute, partial product accumulation, sign check, invert sign, etc.)
a	Address generation via 3 operand arithmetic (which can writeback generated effective address to base register)
L	Load / store instruction
B	PC offset conditional branch
s	Select one register value from 2 registers

Table 2. Instruction decomposition pattern.

ARM Instruction	Internal instructions	Number of instructions
ALU(without shift)	E	1
ALU(with shift)	SE	2
Multiply(32b * 32b + 32b = 32b)	[Mm]*4	8
Multiply(32b * 32b + 64b = 64b)	[Mm]*8 m	17
Multiply(signed32b * signed32b = 64b)	[Mm]*8 E m*3	22
Load / store (without shift, post base update)	La	2
Load / store (without shift, pre base update)	aL	2
Load / store (with shift, post base update)	LSa	3
Load / store (with shift, pre base update)	SaL	3
Multiple load / store (N: number of possible registers)	aa [aL]*N a	2 * N + 3
Branch	B	1

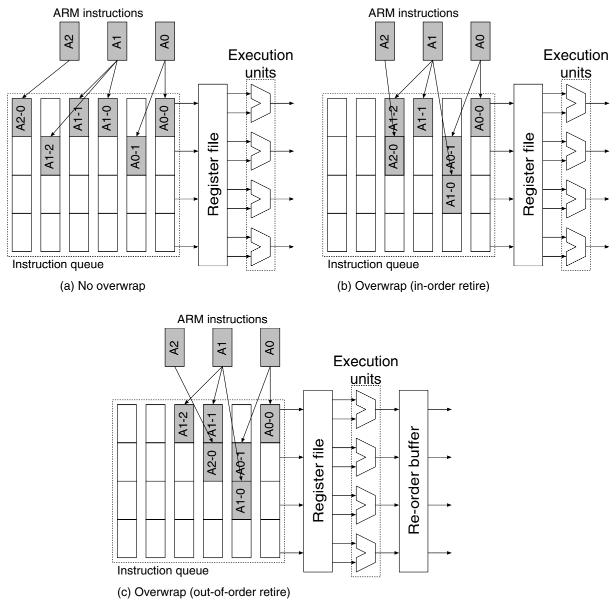

Figure 6. Candidates of ARM instruction insertion method.

the translator and decomposer work as a multi-cycle operation. During decomposition and translation, the “Translator” also needs to assign additional register entries for the decomposed ARM instructions. Thus, register renaming logic is also included as part of the “Translator” circuit.

Figure 5 assumes that the “Translator” decomposes and translates only one ARM instruction in each clock cycle. But there is high possibility that indecomposable ARM instructions may continue in the program execution. So, in current plan, we prepare two ARM “Translator” to increase throughput of ARM instructions.

As introduced in Section 2.3, the ARM instruction set can apply conditional execution to all of above instructions. So, we have to keep this conditional execution information after decomposition and translation. Fortunately, the conditional flags of the FR-V 550 instruction set and the ARM instruction set is the same so that we can easily to keep the conditional execution information after decomposition and translation.

3.3 ARM Instruction Insertion to Instruction Queue

There are some trade-offs when inserting the decomposed and translated ARM instructions. Figure 6 illustrates some of insertion method candidates.

Figure 6(a) shows the simplest insertion method that the decomposed and translated ARM instructions are inserted to be executed in sequential. In this case, because of the serialized execution, the data dependencies are not needed to be checked during insertion. And the execution latencies of precedent instructions are not the issue because all ARM instructions are decomposed into the internal instructions which can execute in one clock cycle. Thus the insertion hardware can be designed to contain only a detector to check weather or not the instruction slot connected to a possible functional unit is free.

Figure 6(b) shows the insertion method that permits overwrap execution. In this method, we don’t permit out-of-order completion so that the insertion hardware only have to check data dependencies in insertion. By decomposing ARM instructions into internal instructions which can execute in one clock cycle, the insertion hardware does not have to consider out-of-order completion caused by multi cycle latency instructions.

Figure 6(c) shows the insertion method that permits out-of-order completion. In this method, we have to add reorder buffer for ARM instructions to guarantee the program order commit and the precise exception. This implementation increases hardware costs drastically. But it increases performance of ARM instruction execution due to increased resource utilization.

Currently, we are evaluating processor performance of each insertion method with a software simulator. After that evaluation, we can choose suitable organization with considering the performance demands of ARM program side.

3.4 Issue Instructions from Instructions Queue

In the instruction issue stage, the VLIW issue hardware issues a whole instruction line from the queue. If there’s a instruction which cannot be issued due to unresolved data dependency, the processor stalls the issue of all the instructions from the same line. This is a major strategy in VLIW processor. This method can reduce complexity around instruction queue, but it also reduces performance due to stall.

The most possible stall is cased by L1 data cache miss. In the usual instruction scheduling, the data dependent instruction which depends on a load instruction will be scheduled that the prior load instruction indicates a hit in L1 data cache. This strategy reduces execution latency greatly if the prior load instruction can be successfully found the data in the L1 data cache. But once the L1 data cache miss occurs, it stalls instruction issue for not only the dependent instruction but also instructions in same line. Moreover, usual instruction queue in VLIW processor does not permit issue of following lines so that The L1 data cache misses degrades performance greatly.

In usual VLIW program, they schedule instruction carefully to reduce performance loss caused by the L1 data cache miss via utilizing a large instruction scope. But in ARM instruction insertion of the OROCHI processor, we only have a limited instruction scope so that there's a high possibility to suffer pipeline hazard due to L1 data cache misses. To alleviate this problem, we may implement cache hit/miss predictor like Compaq Alpha 21264[6]. By moving the instruction which is predicted to cause L1 data cache miss, we can reduce the performance loss.

4 Conclusions ant Future Work

In this paper, we outlined the concept of OROCHI processor which can execute both conventional instruction set and VLIW instruction set simultaneously. We can replace both multimedia processor and general purpose processor in current portable devices by one OROCHI processor. The OROCHI processor is suitable for the demands of reducing processor footprint and power consumption compared to two chips organization.

There are many design candidates to implement the OROCHI processor and there are also many trade-offs. Currently, we are under discussion with software simulator results to propose them most effective implementation at the microarchitecture level. In the future work, we finalize those discussions and carry out chip level design. We are planning to manufacture chips by way of trial to evaluate the accurate performance and power consumption.

Acknowledgment

This research is joint research with Semiconductor Technology Academic Research Center. This work is also supported by VLSI Design and Education Center(VDEC), The University of Tokyo with the collaboration with Synopsys Corporation.

References

[1] C. McNairy and D. Soltis, "Itanium 2 Processor Microarchitecture," *IEEE Micro*, Vol. 23, No. 2, pp. 44-55, March 2003.

[2] ARM Limited, "ARM Architecture Reference Manual," *ARM DDI 0100E*, 2000.

[3] http://www.fujitsu.com/ global/ services/ microelectronics/ product/ micom/ frv/

[4] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, Vol. 9, No. 2, pp. 1-7, February 1995.

[5] P.N. Glaskowsky, "Pentium 4 (Partially) Previewed," *Microprocessor Report*, Vol. 14, Archive 8, pp. 1-4, August 2000.

[6] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, Vol. 19, No. 2, pp. 22-36, March 1999.