社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－ＳＬＤＭ－113 （４）
2004／1／22

Design of Producer-order Parallel Queue Processor Architecture

ArsenijMARKOVSKIJ† MasahiroSOWA† BenABDERAZEK†

SoichiSHIGETA† TsutomuYOSHINAGA†

† Graduate School of Information Systems, University of Electro-Communications, Chofugaoka 1-5-1, Chofu-shi, Tokyo, 182-8585 Japan

E-mail: †{markov,ben,shigeta}@sowa.is.uec.ac.jp, ††{sowa,yosinaga}@is.uec.ac.jp

Design of Producer-order Parallel Queue Processor Architecture

Arsenij MARKOVSKIJ†, Masahiro SOWA†, Ben ABDERAZEK†,

Soichi SHIGETA†, and Tsutomu YOSHINAGA†

† Graduate School of Information Systems, University of Electro-Communications, Chofugaoka 1-5-1, Chofu-shi, Tokyo, 182-8585 Japan

E-mail: †{markov,ben,shigeta}@sowa.is.uec.ac.jp, ††{sowa,yosinaga}@is.uec.ac.jp

Abstract In this paper we describe the design of Producer-order Parallel Queue Processor architecture. It is based on Producer-order Queue Computational Model, which uses Queue (FIFO memory) instead of registers as an intermediate storage of operands. Short program length, ILP orientation, and simple instruction issue mechanism are its main advantages, especially if the target is embedded system. Our processor successfully deals with complexity of superscalar machines.

Key words Queue Computational Model, Parallel Queue Processor, ILP, code size

1. Introduction

Besides traditional demand for high performance, recent efforts in microprocessor design are concentrated on architectures that offer short program size, simple hardware, and low power consumption — features that are especially critical for embedded systems.

Superscalar processors, that support out-of-order instruction execution, need extra hardware (for example, Register Renaming) that solves problem of false dependencies. This is what makes them quite complex.

A wish to have simple, but still fast machine pushed us to look for alternatives. Our research was inspired by several original ideas: [1] proposed to use Queue (FIFO memory) instead of registers (Random Access Memory) as intermediate storage of operands. Queue-based processor that executed instructions in parallel was presented in [2], [3]. Queue Processors have short program size, simple architecture, and are parallelism oriented. [4] elaborated further the theory of Queue computation, and derived 3 modifications of Queue Computational Model (QCM).

1 Producer-Consumer-order QCM (QCMpc)
2 Consumer-order QCM (QCMc)
3 Producer-order QCM (QCMp)

It is proved in [4] that QCMp is the most flexible among these 3 models: it effectively uses data in Queue, makes it easy to build Queue programs, and gives the shortest code size.

QCMpc- and QCMc-based processors have been already designed. In this paper we present the first design of QCMp-based processor architecture, which is called Producer-order Parallel Queue Processor (PQPpf). Letter 'p' in PQPpf acronym comes from QCMp, and 'f' means fixed-length instruction set.

This paper is organized as follows: Producer-order Queue Computational Model is explained in Section 2. Section 3. gives main points about PQPpf architecture, and Section 4. comes with conclusions.

2. Producer-order Queue Computational Model

In this section we explain Producer-order Queue Computational Model (QCMp) that underlies our processor.

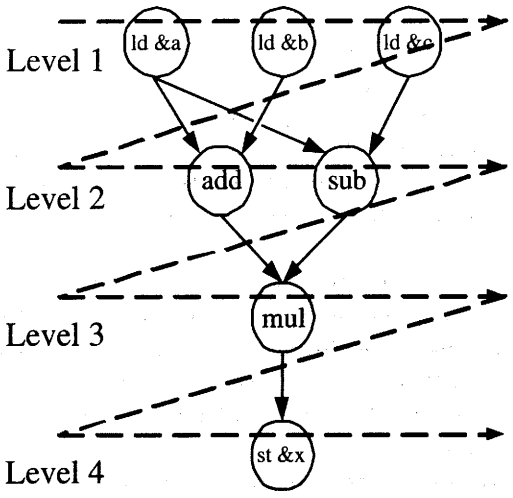
At first we construct *graph* for a simple expression $x = (a + b) * (c - a)$ (see Fig. 1). Nodes of the graph represent operators. `ld &a`, `ld &b`, and `ld &c` load data `a`, `b` and `c` from memory. `&a`, `&b`, and `&c` specify memory addresses of corresponding data. `add`, `sub` and `mul` correspond to 'addition', 'subtraction', and 'multiplication' operators. `st &x` stores the result back into memory.

Arcs specify *data dependency* among nodes. Node that is located at the beginning of arc 'produces' datum. We call such node *'producer'*. Node that is located at the end of the same arc 'consumes' this datum. We call it *'consumer'*. Therefore, consumer is data-dependent on producer. For example, `mul` is data-dependent on `add` and `sub`.

To make program for QCMp we perform *Level-order Traversal* of graph starting from the top level. This is denoted with dashed lines of Fig.1. Obtained QCMp program is:

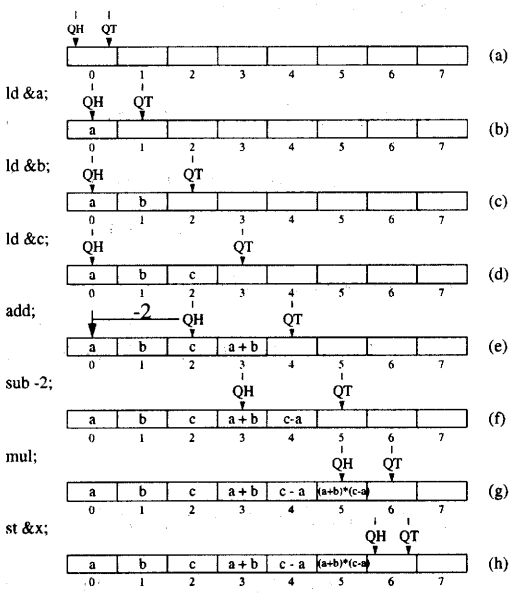
```
1    ld &a;
2    ld &b;
3    ld &c;
4    add;
5    sub -2;
6    mul;
7    st &x;
```

`-2` in `sub -2` is operand which shows one of words in Queue.



1

Fig. 1 Graph of expression $x = (a + b) * (c - a)$.



2

Fig. 2 Content of Queue during execution of program $x = (a + b) * (c - a)$.

Fig. 2 shows the content of Queue during execution of this sequence of instructions (on the left side). Registers between QH (Queue Head pointer) and QT (Queue Tail pointer) make Queue. Operands are implicitly found at QH, and results are written according to QT.

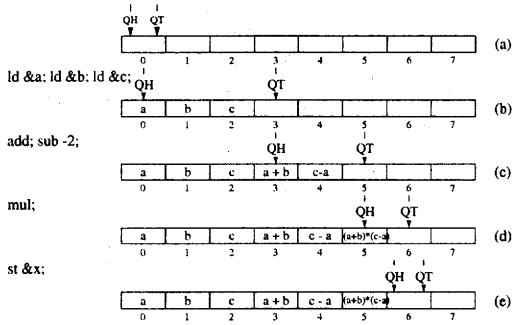
At the beginning Queue is empty (Fig. 2(a)). Fig. 2(b) shows the Queue after execution of `ld&a` instruction. It loads datum a from memory and enqueues it at QT. QT advances by 1 position to reflect writing 1 datum. `ld &b` and `ld &c` are executed in the same way (Fig. 2(c) and Fig. 2(d)).

On Fig. 2(e) the execution of `add` dequeues its operands a and b (QH advances by 2 positions), performs addition and writes the result back into Queue at QT.

Next instruction is `sub -2`. It corresponds to expression $c - a$. c is at QH. a is not at QH because preceding `add` instruction already consumed it. We can access a by specifying its relative position from QH. Thus `-2` points to the second word to the left from QH, which is a (Fig. 2(e)). This technique allows to access previously consumed data. QH advances by 1 position because only 1 operand (c) was consumed from QH (Fig. 2(f)).

As this figure shows, producers always put data at QT. Hence the order data appears in the Queue corresponds to the execution order of producers. This is contrasted with data consumption where we have certain freedom to access operands (for example, `sub-2`). It is principle of QCMp.

Important thing to notice is that nodes that are located



3

Fig. 3 Content of Queue during parallel execution of program $x = (a + b) * (c - a)$.

on the same level (Fig. 1) represent data-independent operators. Therefore we can execute them simultaneously. For example, 3 `ld` instructions from `Level 1` can be executed in parallel as shown below.

```
1   ld &a; ld &b; ld &c;
2   add; sub -2;
3   mul; st &x;
```

Now it takes only 4 steps (height of graph) to execute the program versus 7 steps in case of serial implementation. Corresponding Queue content is shown on Fig. 3.

Now we list prominent advantages of QCMp.

1 QCMp instructions are shorter than RISC-type instructions: zero-operand (`add`) and 1-operand (`sub -2`) versus traditional 3-operand format (`add R1,R2,R3`). QCMp effectively uses data in Queue — operands can be easily accessed after they were consumed. Short instructions and flexibility of QCMp promises reduction of total program size when compared with RISC code.

2 Another feature of paramount importance is absence of *false* (name) dependencies in QCMp. Instructions of register-based machines explicitly specify names of registers. To make length of instructions reasonable, number of registers is limited. Program lacks registers and assigns the same registers for different instructions. This introduces false (name) dependencies and limits the ability to execute these instructions in parallel. To remove false dependencies and boost performance superscalar processors use *Register Renaming* or *Reorder Buffer*. This makes them complex. In contrast, QCMp instructions do no specify register names. Instead, Queue of infinite length is implied to be used. Producer instructions write data into Queue according *single-assignment* rule, i.e. Queue word may be written only once. Therefore, false dependencies do not exist among QCMp instructions. Thus, primary parallelism of program is not reduced. Register Renaming / Reorder Buffer are unnecessary, and processor becomes simpler.

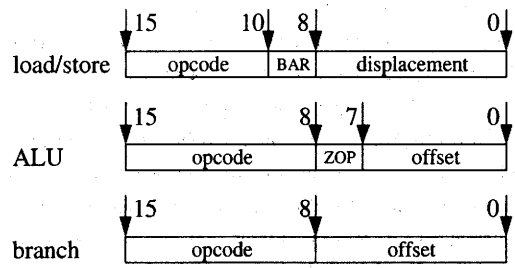
3 One more virtue of QCM is its parallelism orientation. Level-order Traversal of program graph makes Queue programs to look like a sequence of block of independent instructions. Each block corresponds to a particular level of program graph. Instructions inside a block are data-independent and may be executed in parallel. Such localization of Instruction-Level Parallelism makes utilization of Instruction Window more efficient when compared with superscalar processors. This is another source of hardware simplicity of PQPpf.

3. PQPpf architecture

3.1 Instruction Set

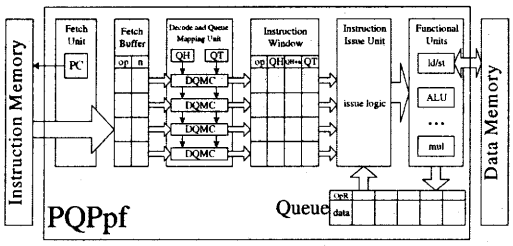
PQPpf is general-purpose load-store processor. We use 2-byte fixed-length instruction set. Format of main instruction classes [4] is given on Fig. 4.

- **Memory access** type instructions (load/store) use *displacement addressing mode.* 6 most significant bits (15–10) specify opcode. PQPpf architecture specifies 4 special-purpose registers that store Base Address. They are called *Base Address Registers* (BAR). Bits 9–8 denote which particular BAR is used. Least significant byte (7–0) is displacement from Base Address. *Effective Address* (EA) is computed as follows: $EA = BAR + displacement$.
- **ALU** Most significant byte (15–8) is opcode. Bit 7 is called *Zero-Operand* (ZOP) bit. If ZOP is cleared, the instruction finds both operands at QH. Bits 6–0 are left unused. When ZOP is set, only 1 operand is consumed from QH, the second one is located by adding 7-bit offset (6–0) to QH.
- **Branch** instructions are PC-relative. Most significant byte (15–8) specifies opcode. Least significant byte is offset. *Branch target* ($target_B$) is computed by adding PC of branch and offset: $target_B = PC + offset$.



4

Fig. 4 Format of main instruction classes.



5

Fig. 5 PQPpf block diagram.

3.2 PQPpf block diagram

PQPpf block diagram is given on Fig. 5. PQPpf consists of the following units: *Instruction Fetch Unit* (IFU) fetches 4 instructions from Instruction Memory according PC, and places them into *Fetch Buffer* (FB). *Decode and Queue Mapping Unit* (DQMU) reads instructions from FB, decodes them, maps onto Queue by assigning QH and QT values, and writes into *Instruction Window* (IW). *Instruction Issue Unit* (IIU) performs multiple out-of-order instruction issue from IW. *Functional Units* (FU)execute instructions and write the result back into Queue.

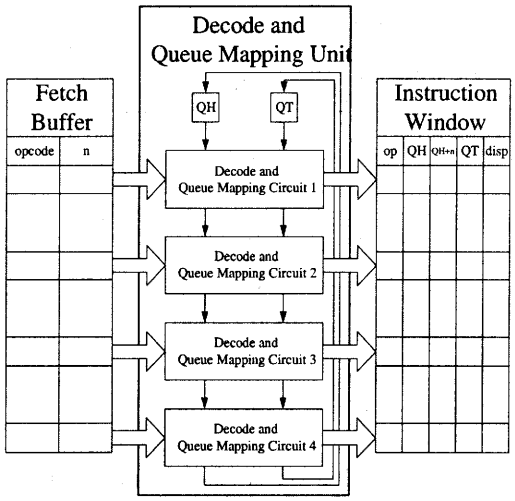
PQPpf has the following specification:

- 32-bit memory address space.
- 32-bit word size.
- 256-word Queue.
- 4-instruction fetch width.
- following Functional Units:
  - 2 ALU's,
  - 1 integer multiplier,
  - 1 shifter,
  - 2 load/store units,
  - 1 branch unit.
- 7-instruction issue width.

3.3 Decode and Queue Mapping Unit

Decode and Queue Mapping Unit (DQMU) is the most distinguishing part of PQPpf. DQMU assigns for an instruction its operand and result addresses that are used later during execution. DQMU is shown on Fig. 6.

DQMU consists of cascaded *Decode and Queue Mapping Circuits* (DQMC). Each DQMC operates on one instruction. Width of DQMU is equal to fetch width (4 instructions), thus there are 4 DQMC circuits. Inputs to DQMC are current HQ and QT values, and an instruction from *Fetch Buffer* (FB). The instruction is decoded, and then mapped onto Queue, i.e. assigned QH and QT values that will be used to read operands and write result during instruction issue and execution. DQMC updates QH and QT and passes them to the next DQMC circuit. New QH and QT are computed according to the number of words *consumed* (C) and *produced* (P)



6

Fig. 6 Decode and Queue Mapping Unit.

by the instruction:

- $QH_{i+1} = QH_i + C$
- $QT_{i+1} = QT_i + P$

Decoder at the beginning of each DQMC provides all information, particularly *consume-produce* values, necessary to map an instruction and update QH and QT. Consume-produce (C-P) values of major instruction classes are given in Table 1.

At the end of cycle all mapped instructions are written into *Instruction Window* (IW). Entry of IW consists of several 1-byte fields:

- $opcode_{IW}$ specifies the type of Functional Unit used by instruction and opcode for that FU.
- QH_{IW} field specifies the location of the first operand.
- $(QH + n)_{IW}$ points to the second operand.
- QT_{IW} specifies where the result of operation will be written.
- $disp_{IW}$ specifies displacement from Base Address for load/store instructions.

Subscript 'IW' used for field naming means 'Instruction Window'.

Instruction Classes	Consume	Produce
ALU	2/1	1
Store	1	0
Load	0	1
Branch	0	0

1

Table 1 Consume-Produce characteristics of major instruction classes.

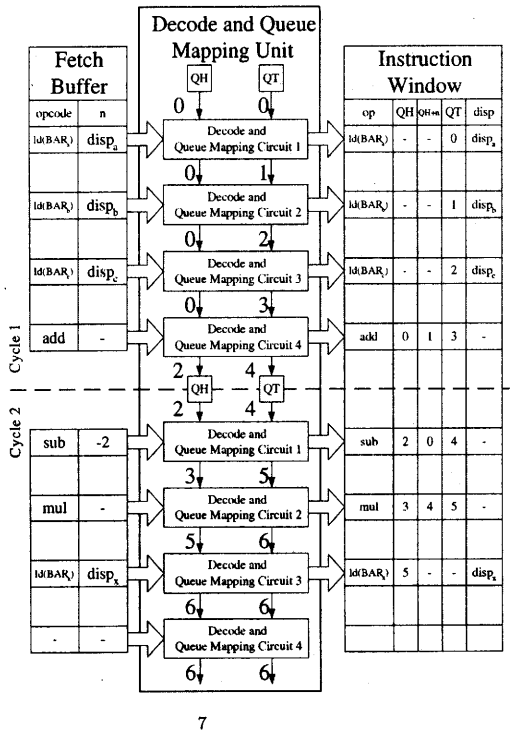


Fig. 7 Operation of Decode and Queue Mapping Unit.

Fig. 7 exemplifies operation of DQMU using program from Fig. 2.

At the beginning (Cycle 1) Fetch Buffer contains the first 4 instructions of our sample program: $ld(BAR_a)$ $disp_a$, $ld(BAR_b)$ $disp_b$, $ld(BAR_c)$ $disp_c$, and *add*.

Queue is empty: $QH = QT = 0$. We start from instruction $ld(BAR_a)$ $disp_a$. Mapping occurs as follows. Because this instruction does not consume data from Queue, QH_{IW} and $(QH + n)_{IW}$ fields are left unused. The result will be written at QT: $QT_{IW} = QT = 0$. *disp* field contains displacement from BAR_a: $disp_{IW} = disp_a$. QH and QT are updated as follows. Zero words are consumed: $QH = QH + 0 = 0$, and 1 result will be produced: $QT = QT + 1 = 1$. $ld(BAR_b)$ $disp_b$ and $ld(BAR_c)$ $disp_c$ are processed in the same way.

Next instruction is *add*. It takes both operands from QH: $QH_{IW} = QH = 0$, and $(QH + n)_{IW} = QH + 1 = 1$. QH advances by 2 positions: $QH = QH + 2 = 0 + 2 = 2$.

Cycle 1 is finished. At positive clock edge mapped instructions are written into Instruction Window. QH and QT are stored in corresponding registers. Content of Fetch Buffer is replaced with newly fetched instructions: *sub* -2, *mul*, and $ld(BAR_c)$ $disp_c$. Cycle 2 begins.

We start mapping from *sub* -2. This is QH-relative instruction. First operand is found at QH: $QH_{IW} = QH = 2$. The position of the second operand is obtained by adding offset to QH: $(QH + n)_{IW} = QH + offset = 2 - 2 = 0$. QH advances by 1 position: $QH = QH + 1 = 2 + 1 = 3$.

3.4 Instruction Issue Unit

Instruction Issue Unit (IIU) provides multiple out-of-order issue in order to exploit available ILP most efficiently.

The main beauty about PQPpf' IIU is its simplicity. In previous section we explained that due to *single-assignment* nature of QCMp there are no *false* dependencies among instructions. This eliminates the necessity for Register Renaming / Reorder Buffer — traditional techniques that remove false dependencies in superscalar processors and make them quite complex.

Another nice point about PQPpf is that can utilize Instruction Window (IW) and Functional Units (FU) more efficiently than superscalar machines do. Because Queue program looks like a sequence of blocks of independent instructions, ILP is very *localized*. This improves *look-ahead ability* of processor. Although compiler for superscalar processors makes certain efforts to group independent instructions, it is still very far from ILP localization of Queue programs. Therefore, we expect that for the same configuration (fetch width, size of IW, issue width, and number of Functional Units) PQPpf can issue and execute more instructions per cycle, and outperform superscalar processor.

Now we are ready to describe issue algorithm of PQPpf. Handling load/store instructions that refer special-purpose BAR registers differs from ALU class. Hence, we consider them separately.

Issue of ALU instructions. It is very easy to issue ALU instructions because only true and resource dependency exist for them. Thus, if a) instruction' operands are ready, and b) corresponding FU is vacant, the instruction is issued — operands are read from Queue and passed to FU, FU is marked as busy, IW entry becomes empty. ALU instructions are issued out-of-order. Older instructions have priority when competing for FU.

Each Queue-word has a corresponding *Operand-Ready* (OpR) bit. Issue logic checks OpR to determine whether operand is ready (OpR is asserted) or not. OpR is set by FU when it writes the result into Queue.

Because Queue is implemented as circular register file, a mechanism to reset OpR is necessary. The problem is following. Assume empty Queue (all OpR are also deasserted). QMU starts mapping instructions. After last word is assigned Queue warps around — mapping again starts from the first word. By this time part instructions from the first pass have completed execution — results were written into Queue and corresponding OpR were set. This make a trouble for the instructions of the second pass — they see their

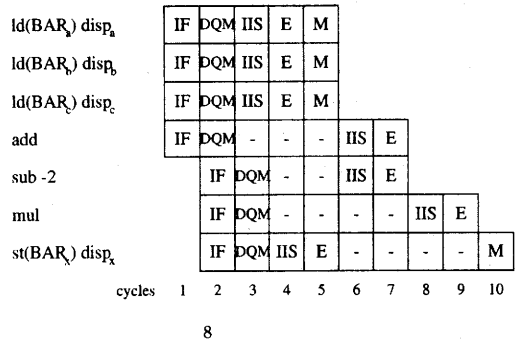


Fig. 8 Sample program execution on PQPpf pipeline.

operands as ready, while actually Queue contains obsolete data. The solution is simple. In order to preserve OpR consistency instructions must reset OpR that correspond to their result destination (QT field). Reset occurs when instructions are written into IW. OpR will be set again when new result is produced.

Issue of load/store instructions. Load/store instructions are executed in 2 steps. At first, Effective Address (EA) is computed by adding base-address (stored in BAR) and displacement, and is written into *Load/Store Buffer* (LSB). Then *Load/Store Unit* (LSU) uses effective address to perform actual memory access. To boost performance LSU implements well-known technique called *load bypassing with forwarding*: a load may bypass preceding store instructions if they do not refer the same memory address; forwarding means that load takes data directly from preceding store instruction that refers the same memory address.

To preserve memory consistency load/store instructions in LSB must be in program order. To satisfy this requirement they must be issued for EA calculation also in-order. Therefore, although use of registers makes a risk for false dependencies, this does not happen — BAR registers are accessed in-order.

Fig. 8 shows symbolic execution of our sample program on PQPpf pipeline. Pipeline stages are abbreviated:

IF: Instruction Fetch

DQM: Decode and Queue Mapping

IIS: Instruction Issue

E: Execution

M: Memory access

We assume that there is no contention for Functional Units. For example, we are able to execute 3 loads simultaneously. All ALU instructions are executed in 1 cycle. Load/store instructions are executed in 2 cycles: effective address is computed during Execution stage, and memory is accessed during Memory stage.

Important thing to notice here is out-of-order execution of $ld(BAR_c)$ $disp_c$instruction. Its effective address is calculated 4 cycles earlier then program result is stored into memory. It reduces execution time to 10 cycles vs. 12 cycles for in-order execution.

3.5 PQPpf Verilog HDL simulator

We wrote behavioral PQPpf simulator in Verilog HDL and proved functional correctness of proposed architecture. At this moment we are creating a set of benchmarks that will be used to compare code size and ILP of PQPpf vs. RISC machine.

4. Conclusions

In this paper we presented the first design of novel processor called Producer-Order Parallel Queue Processor (PQPpf). In our work we made clear PQPpf architecture. We also wrote behavioral PQPpf simulator in Verilog HDL and proved functional correctness of proposed architecture.

PQPpf instructions are short because register names are not specified. Flexibility of underlying Producer-order Queue Computational Model promises that total program size will be also reduced when compared with RISC code.

Queue Mapping Unit (QMU), the central part of PQPpf, assigns to instructions operand and result addresses that are used in instruction issue and execution stages.

Localized ILP of Queue programs improves look-ahead ability of processor. It is expected that for the same configuration PQPpf may show higher performance than corresponding superscalar processor.

Because of single-assignment nature of Producer-order Queue Computational Model false dependencies (WAR and WAW) among instructions do not exist. This eliminates Register Renaming / Reorder Buffer and makes Instruction Issue Unit, usually one of the most complex and power consuming parts of processor, simpler.

[1] B. R. Preiss, "Data flow on a queue machine", PhD thesis, Department of Electrical Engineering, University of Toronto, 1987.

[2] S. Okamoto, A. Maeda, M. Sowa, "Superscalar processor based on queue machine computation model", IPSJ SIG Notes, vol.98, no.37, ARC-129-3, pp.13-18, May 1998.

[3] S. Okamoto, H. Suzuki, A. Maeda, M. Sowa, "Design of a superscalar processor based on queue machine computation model", IEEE PACRIM'99, pp.151-154, August 1999.

[4] M. Sowa, "The foundation of parallel queue machine", Technical Report SLL-00330, Distributed Processing Laboratory, Graduate School of Information Systems, University of Electro-Communications, Tokyo, 2000.

[5] A. Markovskij, "Producer-order parallel queue processor architecture design", Master thesis, Graduate School of Information Systems, University of Electro-Communications, Tokyo, 2002.

[6] J. L. Hennessy, D. A. Patterson, Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers, San Francisco, California, 2003.