A thesis submitted in partial satisfaction of the requirements for the degree of Master of Computer Science and Engineering in the Graduate School of the University of Aizu

Development of Parallel Queue Processor Architecture and its Integrated Development Environment

by

Hiroki Hoshino

February 2011
The thesis titled

 Entwicklung of Parallel Queue Processor Architecture and its Integrated Development Environment

 by

 Hiroki Hoshino

 is reviewed and approved by:

__________________________
Main referee

 Assistant Professor   Date

 Ben Abdallah Abdrazek

__________________________
Assistant Professor   Date

 Hiroshi Saito

__________________________
Assistant Professor   Date

 Song Guo

The University of Aizu

February 2011
# Contents

Chapter 1  **Introduction**  
 1.1 Background  
 1.2 Related Works  
 1.3 Research Objective  
 1.4 Thesis Organization  

Chapter 2  **Queue Computation Model Overview**  
 2.1 Produced Consumed Order Queue Computation Model  
 2.2 Consumed Order Queue Computation Model  
 2.3 Produced Order Queue Computation Model  

Chapter 3  **Produced Order Parallel Queue Processor Architecture**  
 3.1 Instruction Set Architecture  
 3.2 Processing Steps  
 3.2.1 Fetch Stage  
 3.2.2 Decoding and Dynamic Operand Calculation Stage  
 3.2.3 Queue Computation Stage  
 3.2.4 Issue and Reservation Station Stage  
 3.2.5 Execution Stage  
 3.2.6 Queue Write Back Stage  
 3.3 Issue Algorithm  
 3.3.1 Issue Logic  
 3.3.2 Reservation Station Scheme  
 3.4 Circular Queue Register Architecture  

Chapter 4  **Integrated Development Environment**  
 4.1 Queue Compiler  
 4.1.1 Characteristics  
 4.1.2 Structure  
 4.2 Queue Assembler  
 4.2.1 Characteristics  
 4.2.2 Structure  


List of Figures

Figure 2.1  Example of queue program generation ............... 4
Figure 2.2  Example of queue computation steps ................. 5
Figure 2.3  DAG including problems of the queue computation model . 6
Figure 2.4  DAG solving the problems with the produced consumed order queue computation model . 6
Figure 2.5  DAG solving the problems with the consumed order and the produced order queue computation model .................... 7

Figure 3.1  QC-3 instruction format and computation examples ........ 9
Figure 3.2  QC-3 system architecture .......................... 10
Figure 3.3  Pipeline stage example ............................. 11
Figure 3.4  Instruction allocation module. The instruction is distributed according to the information generated by this. .................. 13
Figure 3.5  Instruction allocation information generation algorithm. This chart is repeated four times. ................................. 15
Figure 3.6  Tag generation algorithm. The initialization step is run once at one cycle and generation chart is repeated four times. ............ 16
Figure 3.7  The SET RS tag matching scheme ...................... 18
Figure 3.8  The LDST RS tag matching scheme ..................... 19
Figure 3.9  The circular queue register (QREG) structure .......... 19

Figure 4.1  Queue Compiler structure .......................... 22
Figure 4.2  Queue Assembler structure .......................... 23
Figure 4.3  Top view of the Queue Simulator ...................... 25
Figure 4.4  Simulation result: The intermediate QREG content is shown in middle, Simulator behavior is shown in right .................. 26

Figure 5.1  Altera Stratix III development board .................. 29
Figure 5.2  The actual power consumption on Altera Stratix III development board (mW) ...................... 30
List of Tables

Table 5.1 Benchmark analysis ........................................ 27
Table 5.2 Simulation Result ......................................... 28
Table 5.3 The number of Logic Elements (LEs). Target device: Stratix, EP1S80F1020C5) ........................................ 28
Table 5.4 Synthesis result of QC-3. Target device: Stratix III, EP3SL150F1152C2 29
Acknowledgement

I wish to express my appreciation to Dr. Ben A. Abdrazek for his excellent advice and diligent efforts to guide me through this project. And he gave me good opportunities to improve my skills, which are a way of study and English. Also I would like to appreciate to Dr. Hiroshi Saito and Dr. Song Guo of University of Aizu to revise my thesis. Moreover Dr. Kenichi Kuroda, Dr. Yuichi Okuyama, and Dr. Junji Kitamichi helped my troubles and problems about my project.

Finally, I am deeply grateful to the members of the Adaptive Systems Laboratory at the University of Aizu. These people have supported not only researches but also my school life. I could not have marvelous days in the University of Aizu without a lot of great friends.
Abstract

The high performance processors have been required long time. The instruction level parallelism (ILP) is one of the important essences to enable processors to be high performance processors. There are two main techniques to exploit ILP, VLIW and superscalar. The complex compiler to generate the VLIW instruction is needed in the aspect of software. The superscalar scheme needs large area for the finding ILP with big instruction window and register renaming techniques.

The queue instruction set processor architecture is the approach to get ILP with simple techniques. The intermediate results are saved in the queue register, which follows the first-in first-out rule, instead of the random access register. The instruction reads the operand from the head of the queue register implicitly. The execution result is written into the tail of the queue implicitly. There is no need to specify the register number in the instructions. The instructions for the queue processor are generated by traversing the data acyclic graph in level order manner. There are promising advantages in the queue processor; high ILP and short instruction width.

In this thesis, the superscalar, out-of-order, produced order parallel queue processor (QC-3) is designed. The design is implemented by Verilog-HDL to get high accurate evaluation result with synthesis tools provided by a company. And also the queue compiler, the assembler and its simulator are proposed. These design suites are useful to design the programs for the queue processor.
Chapter 1

Introduction

1.1 Background

The high performance processors have been required long time. The instruction level parallelism (ILP) is one of the important essences to enable processors to be high performance processors. The higher parallelism the program has, the more instructions are executed in parallel. Then the execution time is shorter. To extract ILP, the compilers calculate the relationship among the program whether there are any independent instructions and arrange the register number and the instruction order. Also the existing processors run many in-flight instructions with dynamic scheduling scheme and register renaming. The compilers are very complex in order to deal with the register pressure program and parallelism extracting problem. The processors need more area of the scheduling scheme and register renaming [1]. They are very complex and the bigger area consumes more power.

Existing processors have implemented the instruction window, the register renaming scheme [2] and the reservation stations to exploit ILP. The MIPS R10000 [3] has the register renaming schemes which are free list, active list, and register map table. Also there are 16-entry instruction queues to extract ILP. The PowerPC 750 [4] has a 6-entry issue queue and dispatches up to 2 instructions in the bottom of the queue. Each functional unit has a reservation station to wait for the availability of all operand data. The Alpha 21246 [5] has two register renaming schemes and two issue queues to extract ILP. The UltraSparc I [6] has the renaming scheme and the instruction buffer which has 12 entries to extract ILP. Also the re-order buffer to realize out-of-order scheme is implemented in these processors.

The keys are simple and fast execution time to get high performance processors. The queue computation model is an approach to get the keys, which is originated by the ideas in [7][8]. The most important idea of this computation is the mechanism of the register handling which holds the intermediate calculation results. In the queue computation model, an instruction uses the First-in First-Out register, which is so called queue, instead of the random access register as the storage of the intermediate results. The instructions remove the required data from the head of the operand queue and insert the result at the tail of the operand queue.

The queue computation model needs only the operation code. The source is taken implicitly from the queue entry pointed by the queue head pointer. The result is in-
serted into the queue tail pointer implicitly. There is no need to specify the operand register number in the instruction. All we have to do is just specifying the operation code. The instructions do not specify any operands and destination address. This advantage causes the program size smaller. Also there is no false data dependency because the register number is not specified in the compile time. Thus there is no register renaming scheme in hardware.

1.2 Related Works

The QC-2 was proposed in [9][10][11][12], which employs the produced order queue computation model. In the produced order queue model, the first source operand is always read from the head of the queue register and the second source operand is read from the position addressed with an offset from the queue head. The instructions are generated by level-order manner which traverse the data flow graph from upper left to lower right [7][13][14][15]. In this traversal, the nodes in the same layer are not dependent. Thus there are executed in parallel. This characteristic enables the compiler and the processor to extract ILP easily. The advantages of QC-2 are high instruction level parallelism and short code size.

1.3 Research Objective

In this paper, the superscalar, out-of-order, produced order parallel queue processor (QC-3) is designed. And also the queue compiler, the assembler and its simulator are analysed. These design suites are useful to design the program for the queue processor. The independent instructions are found in the superscalar processor with the issue logic [16]. This issue logic is complex and many papers are looking for the efficient issue design including the superscalar processor design [17][18][19]. In the queue processor, the program has the natural ILP. So the complex issue logic is not needed. Moreover the instruction window, which extracts ILP in general processors, is small in the queue processor. Also the survey of the queue processor characteristics [20][8] mentions that when every functional unit, such as EX unit and LDST unit, has 9 execution units with 15-instruction fetch width, the most parallelism is exploited, assuming that each execution unit can execute in the same cycle. Also in small number of execution units, the out-of-order scheme runs efficiently. Adapting the out-of-order scheme enables the queue processor to get more efficient calculation and to support the functional units having different execution cycle.

In the previous prototype design QC-2, the in-order scheme is employed. The QC-2 is assumed that the needed execution cycles in the functional unit are the same. For example, the addition, division and multiply execution instructions take single cycle in execution stage. Generally the division and the floating point execution need a lot of cycles to accomplish high clock frequency. If the multi-cycle execution units are implemented, the in-order scheme does not run efficiently. The following instructions need to wait the completion of the previous instruction which uses a multi-cycle functional unit. Thus the QC-3 employs the out-of-order scheme. The out-of-order program execution needs a re-order buffer to keep in-order program completion correctly. However
the instructions for the QC-3 are not specified the register number at compile time. The register number is assigned at run-time in program order using the head pointer of the queue register and the tail pointer of the queue. Thus an instruction for the QC-3 complete anytime, which means out-of-order completion can be adapted. The QC-3 can execute instructions in an out of the program order with multiple functional units with new issue logic and reservation station.

There are 4 arithmetic execution units, 4 set register units, 1 load store unit, 1 branch / control unit in the QC-3. The number of functional units is same as the QC-2 because it is easy to compare new one with previous one. Each unit has reservation stations, which holds the instruction until all operand available. To distribute the instruction to the proper reservation station, the issue unit is developed with simple tag matching. The QC-3 is implemented by HDL to make the evaluation easy and accurate. The QC-3 is synthesized for the Stratix III FPGA device by the Quartus II software, which both are provided by Altera inc.. This design approach results in more accurate speed, area and power consumption evaluation.

1.4 Thesis Organization

The rest of this paper describes as follows: Capter 2 reviews the Queue Computation Models. In Chapter 3, the QC-3 processor architecture is mentioned. Chapter 4 gives the Integrated Development Environment, which are The Queue Compiler, Queue Assembler and Queue Simulator. In Chapter 5, the design results of QC-3 is described. Last chapter mentions the conclusion.
Chapter 2
Queue Computation Model Overview

The queue computation model uses the queue data structures for operands manipulation. In this model, the source operand is read from the queue head (QH) and result is written into the queue tail (QT). The instructions do not specify the address explicitly because the addresses of the source operands and destination are defined implicitly. There are advantages of this computation model. No specifying operand derives the shorter instruction width. Thus the program size is small. Also the register addresses are computed at the run-time. Thus there are no false data dependency. Moreover the program of the queue computation model has a high instruction level parallelism (ILP). The program of queue computation model is generated by traversing the level-order manner. The reason why it has a high parallelism is that the same level nodes are independent each other.

The figure 2.1 shows the queue program generation steps for the formula \( x = (a + b) \cdot (c - d) \). The instructions are generated from the data flow graph by traversing level-order.

![Data Flow Graph]

Figure 2.1: Example of queue program generation

The instructions are executed with the queue register shown in figure 2.2. Initially, the registers are empty as illustrated by state 1. The "load a" is executed and the data "a" is written into the QT. Then the QT is added by 1, moves to the one right entry. The state 2 shows the queue register status after the first load instruction execution.
The "load b" instruction is executed with the same way as the "load a" instruction is executed in state 3. All load instructions are calculated in the state 4. In the state 5, the "add" is calculated. The operand "a" and "b" are taken from the QH, then result "a + b" is inserted into the QT. The instruction consumes two data. Thus the QH moves the two right entry. The following instructions are executed like this steps. In the step 7, the store instruction stores the data pointed by the QH to the memory. After that, the register status becomes as the state 8.

However the queue computation model forces the program to get the operand from QH and to write the result to QT. There are three problems as follows; Multi Data Production Problem (MDPP), Cross Arc Problem (CAP) and Instruction Hole Problem (IHP). The figure 2.3 shows the DAG including these problems. The MDPP is that the data is consumed more than 2 nodes. The queue computation model assumes that the data is consumed only one node. In figure 2.3, the node 0 (n0) produces the node 3 (n3) and the node 4 (n4). Also n1 produces n3 and n5. In the queue computation model, one node can produce one node. The CAP is that the arc crosses a lot of arc. The data is always read from the QH. However the node sometimes needs the data from not the QH position. In the figure 2.3, the arc from n0 to n4 gets across the other arcs. The IHP is that the arc jumps over the level. The data is read from the previous level. In the figure 2.3, the arcs from the n1 to the n5 and the n4 to the n6 jump over the level.

To solve these problems, three queue computation models are defined. First queue computation model is "Produced Consumed Order Queue Computation Model". Second one is "Produced Order Queue Computation Model". Third one is "Consumed Order Queue Computation Model". These models are described in the following section.
2.1 Produced Consumed Order Queue Computation Model

The produced consumed order queue computation model is a normal model following the FIFO rule. The data is always read from the QH. The result is always written into the QT. The restriction is very strong. In this model, to solve the three problems, many instructions are inserted. The number of the load instruction is increased for solving the MDPP by the number of the production nodes. To solve the CAP, the node is spilled out in the memory and refilled at an appropriate time using the load and store instructions. Also to solve the CAP, the "dup" instructions that can copy the data in the QH to the QT are inserted into every places jumping the level. The DAG solving the problems is in the figure 2.4. However these solutions make its program bigger obviously.
2.2 Consumed Order Queue Computation Model

In the consumed order queue computation mode, the first source data is always read from the QH and second source data is read from the next QH. And the result is written into the QT and the place pointed by sum of the QT and the specified offset. Therefore this model can produce two data with one instruction. In the figure 2.5 a), the n0 is the load instruction. This load instruction specify the offset in 2, like a "load a 2". Since the data "a" is inserted into the QT and \( QT + 2 \), the n4 calculates completely with the FIFO rule. In this solution, upper two data is generated in one instruction and another one is inserted into the place away from the QT with specifying offset. However to solve the MDPP, many load instructions are still needed. Also the instruction width is wider so that the operand is specified. Thus the program size is bigger.

2.3 Produced Order Queue Computation Model

In the produced order queue computation model, the first source data is read from the QH. And the second source data is read from the place pointed by sum of the QH and the specified offset. Then the calculation result is always written into the QT. The used data can be used again. If the data is not used anymore, the data is removed from the queue register. In the figure 2.5 b), the n4 is sub instruction, which have a offset like a "sub -2". The deference between "sub" and "subo" is the first and second operands are in an opposite side. The first operand data is taken from the place pointed by the \( QH - 2 \) position where it is in the n0. The second operand data is taken from the QH where it is in the n2. And the result is written into the QT. In this solution, the instructions can use the data again. The efficient usage of the queue register is achived. All of the problems explained above are solved in this model. And the best model of the queue processor is this produced order queue computation model in the paper [21][22].

![a) Consumed order QCM b) Produced order QCM](image)

Figure 2.5: DAG solving the problems with the consumed order and the produced order queue computation model
Chapter 3

Produced Order Parallel Queue Processor Architecture

The produced order parallel queue processor (QC-3) employs the produced order queue computation model same as the QC-2. The QC-3 features are as follows;

- 16-bit fixed instruction width
- Extend operand field instruction support
- Specifying offset support
- 4 instruction fetch at one cycle
- 6 processing steps
- 4 instruction issue width
- Out-of-order execution and completion scheme
- 32-bit x 128 entries circular queue register (QREG) with 4-write 10-read port
- QREG controller supporting write port sharing, register data reusing and queue overflow

3.1 Instruction Set Architecture

The instructions for QC-3 are fixed 16-bit width. First 8-bit indicates the opcode and the last 8-bit is operand. There are 87 instructions, which are divided into 6 types. Each instruction is translated into the meaningful information in the decode stage and the register addresses are calculated at the queue computation stage, which are describes in following section. The integer type has 15 instructions such as ”add”, ”sub” and so on. The floating point type has 11 instructions such as ”addf”, ”subf” and so on. These instruction types are categorized in the EX type instruction shown in the figure 3.1 a). The set register type has 35 instructions such as ”setd0ll”, ”seta3hh” and so on. This set register type instruction enables to solve the instruction width shortage. This set register type instruction can set the base address (a0-a3, d0-d3) in the general purpose register (GPR) which are used by the memory instructions and branch instructions. The figure 3.1 b) shows this type of instruction format. The memory type has 8 instructions such as ”ldw0” and ”stw1”. Last digit indicates the base address number, which is set by the set register type instructions. The figure 3.1 c) shows this type of instruction format. The branch type has 16 instructions such as ”beq”. And the control
type has 2 instructions such as "halt". The figure 3.1 d) shows this type of instruction format.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op-code (8 bits)</th>
<th>Offset (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>01000000</td>
<td>01011101</td>
</tr>
</tbody>
</table>

QREG[QH] + QREG[QH+01011101] => QREG[QT]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op-code (8 bits)</th>
<th>Value (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>setd1ll</td>
<td>01100100</td>
<td>01111000</td>
</tr>
</tbody>
</table>

{upper 24 bits of GPR[d1], value 8 bits} => GPR[d1]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op-code (8 bits)</th>
<th>Displacement (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>stw2</td>
<td>10010010</td>
<td>11100101</td>
</tr>
</tbody>
</table>

QREG[QH] => MEM[d2 + 11100101]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Op-code (8 bits)</th>
<th>Displacement (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp0</td>
<td>00100001</td>
<td>01000101</td>
</tr>
</tbody>
</table>

GPR[a0] + 01000101 => PC

Figure 3.1: QC-3 instruction format and computation examples

The shortage of instruction width suffers from the operand field width shortage. To cope with this shortage, the QCaEXT technique is implemented, which uses the "covop" instruction. This instruction extends the operand field from 8-bit to 16-bit by inserting the "covop" into before the needed instruction. The 8-bit field of the "covop" instruction is connected to the next instruction operand field in the decoding stage.

### 3.2 Processing Steps

The QC-3 system architecture is shown in figure 3.2.

There are 6 stages; instruction fetch stage, instruction decoding and dynamic operand calculation stage, queue computation stage, issue and reservation station stage, execute stage and queue write back stages. In the decoding an dynamic operand calculation stage, the instruction is decoded and also the "covop" instruction, which is used for extending the operand field from 8-bit to 16-bit, is calculated to generate the complete operand field. The queue computation unit calculates the addresses of instruction source register and destination register using the queue head pointer (QH) and the queue tail pointer (QT). The issue unit calculates the allocation information to distribute instructions to reservation station where the instruction should be saved. Each execution unit checks the instruction ready; all source register availability. If the instruction is ready, the execution unit reads the source data from the queue register (QREG) and writes the result to the queue register through the queue write buffer (QWB).

The set register instructions are saved in the SET RS. The SET RS has 4 entries and it works as a first-in first-out buffer. The set register instruction is divided into 4 instructions because of the instruction width shortage. Thus 4 set register instructions should be executed sequentially.
Figure 3.2: QC-3 system architecture
The integer execution instruction pipeline stage: 6 stages

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>QC</th>
<th>IS&amp;RS</th>
<th>RF&amp;EX</th>
<th>WB</th>
</tr>
</thead>
</table>

The load instruction pipeline stage: 7 stages

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>QC</th>
<th>IS&amp;RS</th>
<th>RF&amp;AG</th>
<th>RMEM</th>
<th>WB</th>
</tr>
</thead>
</table>

IF: Instruction Fetch  
ID: Instruction Decode  
QC: Queue Compute  
IS&RS: Issue and saving Reservation Station  
RF&EX: Register Fetch and Execution  
RF&AG: Register Fetch and Address Generation  
RMEM: Read Memory data  
WB: Write Back

Figure 3.3: Pipeline stage example

The memory instructions are saved in the LDST RS, which has 4 entries. The two instructions from the head of the RS are checked dependency. If there are independent, two memory instructions are executed in the execution stage.

The branch and control instructions are saved in the BRANCH RS. The BRANCH RS has 1 entry. If the RS have an instruction, the issue unit stop working until the branch instruction is executed. The RS also waits for other RS free. It means the branch and control instructions needs completion of all previous instructions. If the other RSs are free, the RS activate the instruction ready. The execution unit executes the branch and control instruction.

The load instruction pipeline stage of Queue Core has 7 stages. The integer execution instruction pipeline stage has 6 stages as a figure 3.3 illustrating.

3.2.1 Fetch Stage

The fetch unit (FU) gets the instructions from the instruction memory. The fetch width is 8-byte. Each instruction is 2-byte width, so the 4 instructions are fetched in one cycle. The fetched 8-byte data is saved in the fetch buffer. Also the program counter is added by 8.

3.2.2 Decoding and Dynamic Operand Calculation Stage

The decode unit (DU) gets the 8-byte data from the fetch buffer. The data is divided into 4 instructions and decoded. The same flag, the consumed number and the produced number are assigned to each instruction. Then 4 decoded instructions are saved in the decode buffer. The consumed number (CN) indicates the number that this instruction consumes and the produced number (PN) indicates the number that this
instruction produces, which are used by next unit, the queue computation unit. Also the “covop” instruction is calculated in this stage. The covop register is implemented to hold the operand data and concatenate the next instruction operand field with the covop operand filed data.

3.2.3 Queue Computation Stage

The queue computation unit (QCU) computes the addresses of source and destination. This unit is one of the key units of the queue processor. The register numbers are calculated dynamically. This unit holds the head pointer of the queue register, which is called QH, and the tail pointer of the queue register, which is called QT. Also this unit holds the live queue pointer, which enables to reuse the data in QREG, called LQH. The first source address is decided by the QH, and second source address is decided by the sum of the QH and the offset that its instruction has. Also the destination address is decided by the QT. After that the pointers are increased with following formula. The next QH is calculated by the formula, $Q_{H_{\text{next}}} = Q_{H_{\text{current}}} + \text{consumednumber}(CN)$. The next QT is calculated by the formula, $Q_{T_{\text{next}}} = Q_{T_{\text{current}}} + \text{producednumber}(PN)$. LQH is basically same as QH. If the STOP LQH instruction, which is named “stplqh”, comes into the QCU, QH current is held by LQH until the AUTO LQH instruction comes. If the AUTO LQH instruction, which is named ”autlqh”, comes to the QCU, LQH pointer moves automatically from the LQH value plus offset position. The 4 instructions which have calculated addresses are inserted into the instruction queue (IQ) which holds the executive instructions. The instruction queue has 8-entry buffer. The IQ can send up to 4 instructions depending on the issue stage availability. If the IQ is full, previous units are stalling until the IQ is not full.

3.2.4 Issue and Reservation Station Stage

The issue unit (IU) has the issue logic and the reservation stations (RSs). The issue logic checks the RS availability and makes the allocation information for 4 instructions from the IQ. Each instruction is distributed into appropriate RS depending on its instruction type. This issue mechanism is described later.

3.2.5 Execution Stage

There are 4 EX, 4 SET, 1 LDST and 1 BRANCH units. The EX unit includes the execution function of integer and floating point. The ready instruction is fed from each RS and executes each execution modules. When the execution module finishes calculating the instruction, the module sends the finish signal to the RS which releases the RS slot.

3.2.6 Queue Write Back Stage

The results generated by each EX unit and LDST unit are written into the queue write buffer (QWB). The number of write port of the QREG is restricted to 4. The QWB and QREG controller are implemented to share and arbitrate the QREG port.
The load data from the memory has a high priority. When the QWB has the data from the data memory, the data is written into the QREG first.

### 3.3 Issue Algorithm

In the in-order scheme, if there is an instruction which takes a long time to execute, the next instruction needs to wait for previous instruction completion. The previous QC-2 system assumes that every functional unit is executed in one cycle even if the functional unit handles the floating point execution. The survey of the queue processor characteristics [20] mentions that when the every functional unit, such as EX unit and LDST unit, have 8 execution units with 12-instruction fetch width, the most parallelism is exploited, assuming that each execution unit can execute in the same cycle. Also in small number of execution units, the out-of-order scheme runs efficiently. Adapting the out-of-order scheme enable the queue processor to get more efficient calculation and to support the functional units having different execution cycle.

The issue unit distributes the coming 4 instructions to each reservation station. There are 1 slot RS for each EX unit, 4 slots RS for each SET unit, 4 slots RS for LDST and 1 slot RS for BRANCH as illustrated by figure 3.2.

#### 3.3.1 Issue Logic

![Instruction Allocation Module](image)

Figure 3.4: Instruction allocation module. The instruction is distributed according to the information generated by this.
An instruction has the RS type information and the destination address. And the all RS have a free table which indicates whether the RS has any free RS slot or not. Also the tag register is in the issue module. Since the queue instruction set is fixed 16-bit width, the operand field is 8-bit width. This instruction width is short to have an enough immediate value and memory displacement offset. The address-register and data-register are implemented in the QC-3. These registers are base register, which enable instructions to access the full 32-bit memory space. And the set register instructions are implemented in the queue processor to set these base registers. However the set register instructions and the memory access instructions are executed completely in program order. For example when the load instruction using the d0 register is executed before executing the set d0 register instruction, the load instruction refers an incorrect address. Thus the tag matching system is adapted to keep program order. After executing all previous set register instructions, the memory access instructions are executed. Also after all previous memory access instructions are executed, the set register instructions are executed. The tag field for set instruction is 2-bit. The tag field for memory access instruction is 2-bit. And the destination of lower 2-bit is needed. If the all previous set or memory instructions are executed, the issue module sets the instruction ready bit to its instruction. If the instruction ready bit of a RS slot is set by 1, the RS can send this slot instruction to the execution stage without checking the tag. The tag field is total 7-bit. The block diagram of this allocation module is shown in figure 3.4. And the allocation information algorithm is shown in figure 3.5. Also the tag information generation algorithm is shown in figure 3.6.
Figure 3.5: Instruction allocation information generation algorithm. This chart is repeated four times.
Figure 3.6: Tag generation algorithm. The initialization step is run once at one cycle and generation chart is repeated four times.
After an instruction is assigned the allocation information, the free tables of RS and the tags are updated. This updated information is used for next following instructions. The issue logic calculates the allocation information for up to 4 instructions at one cycle. If any RS is full, the issue logic stop assigning allocation and updating RS free tables and tags. Also when the branch and control instructions are allocated into the branch RS, the following instructions are stall to keep correctness.

### 3.3.2 Reservation Station Scheme

There are 4 types of reservation station. The RS holds the instruction(s) and sends the ready instruction(s) to execute.

**EX RS**

The EX RS has 1 slot to hold the execution type instructions. There are 4 EX RSs for 4 EX units. The RS checks the operand source registers availability using the queue register ready bit (QREGRDY). If all QREGRDY addressed by source fields are set 1, the instruction can be executed. The RS sends this instruction to execution unit with the instruction ready signal. When the instruction completes to be executed in the execution stage, the execution unit sends the finish signal. The RS gets the finish signal and releases the RS slot free with being set the free table by 0.

**SET RS**

The SET RS has 4 slots to hold the set register type instructions. There are 4 SET RSs for 4 SET units. The RS employs the queue scheme. The instruction of the head of the RS is sent to the execution stage. The RS slot has two bits to indicate the instruction valid and ready. The instruction valid bit is set when the instruction is registered into the slot. The ready is set if the tag sent from the LDST unit is matched with the tag that the slot has. If the valid and ready are set, the instruction of the slot is send to the execution unit. When the instruction completes to be executed in the execution stage, the execution unit sends the finish signal. The RS gets the finish signal and releases the RS slot free with being set the free table by 0.

The tag matching scheme is illustrated in figure 3.7. There are comparators to check the tag is matched or not. If it is matched, the tag ready bit is set and the instruction becomes ready to execute.

**LDST RS**

The LDST RS has 4 slots to hold the memory access type instructions. There is 1 LDST RS for 1 LDST unit. The RS employs the queue scheme same as the SET RS. However the LDST RS has 2 output ports to send 2 instructions. The LDST RS has 2 bits same as the SET RS. If the instruction valid and ready bits are set by 1, the instruction of the slot pointed by the LDST RS queue head. If the instruction of the slot pointed by the LDST RS queue head plus 1 position, and if this instruction is independent from the instruction in the slot pointed by the LDST RS queue head, the second instruction is sent to the execution unit with the second send port. The
dependency is checked by the base register number, the offset, and op-code of both instructions. There is no dependency in case of two load instructions. Otherwise in case of either the op-codes are two store or load and store instruction series, the register and offset are compared. If these are the same, these instructions are dependent. And an instruction is set to the execution unit. When the instruction completes to be executed in the execution stage, the execution unit sends the finish signal. The RS gets the finish signal and releases the RS slot free with being set the free table by 0.

The tag matching scheme is illustrated in figure 3.8. There are comparators to check the tag is matched or not. If it is matched, the tag ready bit is set and the instruction becomes ready to execute.
Figure 3.8: The LDST RS tag matching scheme

**BRANCH RS**

The BRANCH RS has 1 slot to hold the branch and control type instructions. The RS checks the other RS free table to recognize all other previous instructions have been executed. If the other RSs are free, the RS sends the instruction to the execution unit. When the instruction completes to be executed in the execution stage, the execution unit sends the finish signal. The RS gets the finish signal and releases the RS slot free with being set the free table by 0.

### 3.4 Circular Queue Register Architecture

Figure 3.9: The circular queue register (QREG) structure
The key units of the QC are this circular queue register (QREG) and its controller. The QREG has 3 pointers: LQH, QH and QT. The data is written into the entry pointed by the QT. After the write back to the QREG, the QT is added by 1. The data is read from the entry pointed by the QH. After the read operand, the QH is added by 1. The LQH is a special pointer to use the data again. To control the LQH, there are "stplqh" and "autlqh" instructions. If the entry is used again, before the data is consumed, the "stplqh" instruction is executed to stop the LQH pointer. When the data is not used any more, the "autlqh" pointer is moved automatically. The QREG is divided into 3 types of entries: empty entries, live entries and dead entries in figure 3.9.

The QREG provides the operand data to 4 EX unit and 1 LDST unit. The number of read port needs 10 and the number of write port needs 6. The more the number of port increases, the more area is needed. So 6 write ports decreases into 4 write ports. The 2 write port is shared by 2 EX and LDST write port. The queue write buffer (QWB) is implemented to wait for arbitrating the write port sharing.

The QREG controller has the queue register ready bits (QREGRDY). When the data is inserted into the QREG, the QREGRDY is set by 1. This bit indicates the availability of QREG data to solve the true data dependency. When the data is read by instruction, the QREGRDY is set by 0. The data of this entry is not available any more. If the "stplqh" has been executed, the QREGRDY is not reset even if the instruction consumes the data from the QREG. When the "autlqh" is executed, the QREGRDY is set by 0.
Chapter 4

Integrated Development Environment

There are design suites of the development queue program, which are the queue compiler, the queue assembler and queue simulator. The queue compiler translates the assembly code from the high level language with gnu compiler tool set. The queue assembler translates the machine code from the assembly code. The queue simulator shows the behavior of the inside of the Queue Processor. These are analyzed in this chapter.

4.1 Queue Compiler

The Queue Compiler translates high level languages to the assembly codes. The some code optimizations are taken for shorter execution time program, higher parallelism, better resource utilization and among others. The C programs are converted to the assembly program for the Queue Processor using the GCC front-end, middle-end and Queue Related back-end. The queue compiler tasks are mainly three: (1) constrain all instructions to have at most one offset reference, (2) compute offset reference values and (3) schedule the program expressions in level-order manner. This compiler can generate denser code comparing with the compilers for the RISC machine.

4.1.1 Characteristics

Generally, the compilers try to find the instruction level parallelism (ILP) in compile time. The register machine executes the instruction which has three operands, which are addresses of one destination and two sources. The compiler checks the dependency among the register number to extract ILP. Changing the register number and rearranging instruction order are very complex. The Queue Compiler uses the level-order manner by traversing the leveled data acyclic graph to generate the assembly for the Queue Processor. According to the program generation way, higher ILP can be got easily. The Queue Compiler can get more parallelism than the optimizing compiler for the RISC machine. Also the size of the program for the Queue Processor is smaller than the one for the RISC machine.
4.1.2 Structure

The Queue Compiler is constructed of three stages. First stage is generating GIMPLE stage, which is a machine independent format, using the GCC front-end and GCC middle-end. Second stage is calculating offset stage. The instructions for the Queue Processor is constructed of one op-code and one offset. The constrain of 1-offset code is adapted in this stage. This constrained representation is called QTree. The leveled data acyclic graph (LDAG) is generated using this QTree. As generating LDAG, the ghost nodes which serve as a mark for the level-order traversal algorithm are inserted for the algorithm. Then the ghost nodes are replaced by the dup instructions, which are instructions to send the node to the next level. According to this graph, the offset value is calculated. The offset is a distance between the first source position, QH, and second source position. The last stage is scheduling stage. The scheduling algorithm is based on the basic block scheduling, where the only difference is that instructions are generated from a level-order LDAG traversal. All nodes are traversed from left to right, and from deeper to upper. The LDAGs are converted into the Queue Intermediate Representation (QIR), which is low level intermediate representation to express the instructions for the Queue Processor. Then the natural ILP is extracted by using the statement margin transformation by reordering the instructions. It is an option for the Queue Compiler. Finally, the QIR is converted to the assembly code for the Queue Processor. The whole structure of the Queue Compiler is shown in figure 4.1

![Queue Compiler Structure Diagram](image-url)
4.2 Queue Assembler

The Queue Assembler translates the assembly codes to the machine codes for the Queue Processor. Also the Queue Assembler generates the target addresses of jump and branch. Moreover if the offset is more than 127, the offset is extended with "covop" instruction because of a shortage of instruction width.

4.2.1 Characteristics

There are some features in the Queue Assembler.

- Assembling multiple input syntaxes
- Providing a tool suitable for compiler, an intuitive and simple framework for human programmers
- Easy retargetable and portable assembler for queue computes
- Multiple output for different target architecture support
4.2.2 Structure

The Queue Assembler is constructed by three phases: preprocessor, syntax analysis and generation. The first phase preprocessor translates the Queue Compiler output to the intermediate file. The Queue Compiler output can not be allowed as the Queue Assembler input. The translated intermediate file is provided to the second phase, syntax analysis. The program is divided into tokens and these token are categorized with machine description file. The categories are mnemonic, operand1, operand2, label and comment. Finally the third phase generation translates the tokens into the machine code with machine description file. The machine description file contains the some fields: mnemonic, opcode of the machine dependent, value and offset. The machine description file is depends on the queue processor architecture. Thus the multiple output is supported by changing the machine description file. Also the "covop" instruction is supported in the Queue Assembler. To cover the instruction width shortage, the "covop" instruction extends the operand fields from 8-bit to 16-bit. The operand field can cover 16-bit width. Figure 4.2 shows the structure of the Queue Assembler.

4.3 Queue Simulator

The Queue Simulator simulates the behavior of the simple Queue Processor. This simulator is developed using Java language. Figure 4.3 shows the simulator top view.

The left field shows the Queue Register contents. The value inside the QREG can be watched to debug the program. The assembly program is used as a simulator input. The programmer can edit an opened assembly in the middle field of the simulator. Also the intermediate results both after load instructions execution and after ALU instructions are shown in the middle field. The right field shows the simulator progress results. What instruction is executed is shown in this panel. The Queue Simulator shows which instruction uses which QREG value. So the Queue Simulator enables the programmer to debug easily. Figure 4.3 shows the simulation result.
Figure 4.3: Top view of the Queue Simulator

25
Figure 4.4: Simulation result: The intermediate QREG content is shown in middle, Simulator behavior is shown in right
Chapter 5

Design Result

5.1 Benchmark Analysis

There are two benchmarks to evaluate the QC-3 performance. One is 4x4 matrix multiplication. Another is 8x8 DCT. The analysis result of programs for the Queue Processor is shown in table 5.1.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>4x4 Matrix</th>
<th>8x8 DCT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inst Num</td>
<td>Percentage</td>
</tr>
<tr>
<td>INT</td>
<td>264</td>
<td>40.0</td>
</tr>
<tr>
<td>FP</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>MEMORY</td>
<td>144</td>
<td>21.9</td>
</tr>
<tr>
<td>BRANCH</td>
<td>61</td>
<td>9.3</td>
</tr>
<tr>
<td>Others</td>
<td>190</td>
<td>28.8</td>
</tr>
<tr>
<td>Total number of instructions</td>
<td>659</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.1: Benchmark analysis

As the table 5.1 showing, the INT type is the most executed instruction in the Matrix benchmark. Also the FP type is the most executed instruction in the DCT benchmark. The BRANCH type is a small amount of whole program because some loop constructs are unrolled to extract higher parallelism.

In this time, the simulation result was taken under the NC-Verilog verilog simulator [23]. The simulation result is shown in table 5.2. QC-2 is represented previous our Queue Processor. The benchmarks are executed on each Queue Processor, and the execution cycles are counted.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QC-2</th>
<th>QC-3</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4 Matrix</td>
<td>Cycles</td>
<td>2224</td>
<td>820</td>
</tr>
<tr>
<td></td>
<td>IPC</td>
<td>0.33948</td>
<td>0.80365</td>
</tr>
<tr>
<td>8x8 DCT</td>
<td>Cycles</td>
<td>72064</td>
<td>35382</td>
</tr>
<tr>
<td></td>
<td>IPC</td>
<td>0.36122</td>
<td>0.63456</td>
</tr>
</tbody>
</table>

Table 5.2: Simulation Result

As the table 5.2 describes, each benchmark is run on QC-3 by shorter time than QC-2. And in average, about 2 times faster execution is realized. Thus the QC-3, which employs the out-of-order scheme, does not take an unnecessary wait and get highly efficient execution.

5.2 Synthesis Result

The QC-3 is compared with previous QC-2 in terms of area and speed for the FPGA board. The table 5.3 shows the synthesis result comparison. The QC-3 is synthesized by Quartus II provided by Altera Corporation [24]. The target FPGA device is Stratix family, EP1S80F956C6.

<table>
<thead>
<tr>
<th>Descriptions</th>
<th>Modules</th>
<th>QC-2</th>
<th>QC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch Unit</td>
<td>FU</td>
<td>633</td>
<td>223</td>
</tr>
<tr>
<td>Instruction Decode Unit</td>
<td>DU</td>
<td>2573</td>
<td>589</td>
</tr>
<tr>
<td>Queue Computation Unit</td>
<td>QCU</td>
<td>1949</td>
<td>2325</td>
</tr>
<tr>
<td>Issue Unit</td>
<td>IU</td>
<td>25026</td>
<td>3090</td>
</tr>
<tr>
<td>Execution Unit</td>
<td>EU</td>
<td>7868</td>
<td>10693</td>
</tr>
<tr>
<td>QREG Unit (64)</td>
<td>QREG</td>
<td>35541</td>
<td>19196</td>
</tr>
<tr>
<td>Memory Unit</td>
<td>MEM</td>
<td>4158</td>
<td>19136</td>
</tr>
<tr>
<td>Total LEs</td>
<td></td>
<td>77819</td>
<td>55251</td>
</tr>
</tbody>
</table>

Table 5.3: The number of Logic Elements (LEs). Target device: Stratix, EP1S80F1020C5

The main difference between two processors is the issue scheme. As the table showing, the number of LEs of the issue unit is 25026 in QC-2 and the number of LEs of issue unit is 3090 in QC-3. The issue unit is decreased about 87.7 percent. The instruction window size is 8 for just saving decoded instructions. Also a small number of reservation stations (1-slot 4 RSs for EX unit, 4-slot 4 RSs for SET unit, 4-slot 1 RS for LDST unit and 1-slot 1 RS for BRANCH unit) enables to realize the small area consumption.

The synthesis result is changed depending on the FPGA device. To implement the QC-3 on an actual FPGA board, this is synthesized for Stratix III, EP3SL150F1152C2. The figure 5.1 shows a photo of Altera Stratix III development board. And the LCD
display shows the QC-3 status, such as the program counter, the signals of memory access, jump, branch, and halt. Also the clock is shown by blinking LED on the board.

![Image of Altera Stratix III development board](image)

**Figure 5.1: Altera Stratix III development board**

The synthesis result for Stratix III is shown in table 5.4. The ALUT represents for the Advanced Look-Up Table, which is the cell of Stratix III FPGA device used as the output of logic synthesis. This ALUT is used for evaluating area.

<table>
<thead>
<tr>
<th>Descriptions</th>
<th>Modules</th>
<th>ALUTs</th>
<th>REGISERs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch Unit</td>
<td>FU</td>
<td>124</td>
<td>220</td>
</tr>
<tr>
<td>Instruction Decode Unit</td>
<td>DU</td>
<td>283</td>
<td>272</td>
</tr>
<tr>
<td>Queue Computation Unit</td>
<td>QCU</td>
<td>1923</td>
<td>668</td>
</tr>
<tr>
<td>Issue Unit</td>
<td>IU</td>
<td>2489</td>
<td>978</td>
</tr>
<tr>
<td>Execution Unit</td>
<td>EU</td>
<td>6927</td>
<td>1765</td>
</tr>
<tr>
<td>QREG Unit</td>
<td>QREG</td>
<td>39813</td>
<td>9426</td>
</tr>
<tr>
<td>Memory Unit</td>
<td>MEM</td>
<td>269</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>51828</td>
<td>13329</td>
</tr>
</tbody>
</table>

Table 5.4: Synthesis result of QC-3. Target device: Stratix III, EP3SL150F1152C2

The QC-3 achieves 63.12MHz as a maximum clock frequency. The total power consumption estimation result running the Matrix benchmark is 1267mW where the static power consumption is 569.7mW and the dynamic power is 651.3mW when the clock frequency is 50MHz.
The Stratix III development board can measure the FPGA core power consumption with 7-segment LED display. The figure 5.2 illustrates the actual FPGA core power consumption after downloading the QC-3 synthesized file (sof file) using Quartus II version 9.0. The display shows around 600mW running the Matrix benchmark. The execution time is very short when the clock frequency is 50MHz because the Matrix needs 820 clock cycles to be executed. Thus the static power consumption occupies the total power consumption shown in the display on the board.
Chapter 6

Conclusion

In this thesis, the superscalar, out-of-order, produced order parallel queue processor (QC-3) is designed. The issue unit generates the allocation information for each reservation station, which holds the instructions. At this time, to keep the order of memory and set register type instruction, the tag system is attached to instructions. The reservation station checks the true dependency using the queue register ready bit. If all operands are ready, the instruction in the reservation station is sent to the functional unit. The reservation station of the load store unit checks the set register unit completion signal and the tag. The reservation station compares the tags and if it is matched, the matched instruction becomes ready. The queue processor does not wait unnecessary time to complete other independent instructions. Also the queue processor assigns the destination address of register in dynamic according to the queue computation model. Thus the execution result is written into the queue register in any time. These systems enable the queue processor to get more high performance.

The performance evaluation using the 2 benchmarks, which are 4x4 matrix multiplication and 8x8 DCT, shows 2 times faster execution time than previous queue processor. The area of the issue unit is also about 87 percent smaller than previous one. The small issue unit enables the queue processor to get more high performance. Also the power is small. The total estimated power consumption is 1267mW, where the static power is 569.7mW and the dynamic power is 651.3mW, during executing the matrix benchmark. The FPGA board, which is Altera Stratix III development board, shows the FPGA core power consumption. After downloading the QC-3 running at 50MHz as a clock frequency, its power display shows about 600mW. Also the brief analysis of the Queue Compiler, the Queue Assembler, and Queue Simulator is shown in this paper.
Publication


References


Appendix A

Queue Core 3 Verilog-HDL code

The developed parallel queue processor (QC-3) is implemented in the VerilogHDL code. The codes are attached in this appendix.

CODE DESCRIPTION