Introducing Pipelining Technique in an Object-Oriented Processor

Mok Pak Lun, Anthony S. Fong
Department of Electronic Engineering
City University of Hong Kong
83 Tat Chee Avenue, Kowloon, Hong Kong.
E-mail: eefong@cityu.edu.hk

Abstract

High Level Instruction Set Computer (HISC) supports Object-Oriented Programming (OOP) in hardware level. HISC provides the objects’ information to hardware through the use of tables and a data structure called Operand Descriptor. An implementation of the HISC architecture, which currently named jHISC, is under research. It is a 64-bit processor that developed mainly targets for JAVA. In this paper, we will discuss how the jHISC processor can use the feature of pipelining technique to enhance the performance. We will start from defining the stages in HISC, then the architectural design. Finally, some issues about the implementation of jHISC on FPGA will be discussed.

1. Introduction

Object-Oriented Programming is the trend of software development. It provides three major characteristics: encapsulation, inheritance, and polymorphism, to enhance the security, the maintainability, and portability of software.

In the current practice, the OOP are running on processors designed for the original procedure-oriented programming (POP). Translation from the OOP concept to hardware instructions is thus necessary in current machines. This translation slows the performance of OOP in the current system, and it corrupts the security provided in OOP at hardware level.

There are schemes to enhance the performance of OOP in current machines, such as flattening the objects, increasing the clock speed of system, optimizing code translation, etc. However, it is desirable to support OO in hardware level. An Object Oriented Processor is proposed.

High Level Instruction Set Computer (HISC) architecture is proposed to use the structure of tables to represent the idea of objects in hardware. It is a general-purpose descriptor computer to support OOP. HISC gets object information in the hardware via Operand Descriptor (OD).

jHISC is an research machine of the HISC architecture. It mainly targets to support JAVA. jHISC is a 64-bit general-purpose architecture. Different Operand Descriptor Tables (ODTs) and data structures are defined for representing the idea of class, instance, method, and data in OOP. To enhance the performance of jHISC, pipelining technique is introduced. In this paper, we will focus on the design and implementation of pipelining in jHISC.

2. Pipelining stages in HISC

We defined jHISC into a seven stages pipeline machine: Instruction Fetch (IFETCH), Instruction Decode (IDECODE), Operand Descriptor Fetch (ODFETCH), Operand Descriptor Check (ODCHECK), Data Fetch (DFETCH), Execute (EXEC), and Writeback (WBACK).

2.1 Instruction Fetch (IFETCH)

In this stage, 64-bit instructions are fetched from the memory. Instructions would pass from the instruction cache, to the instruction buffer, then to the next stage. Once the instruction is passed to the next stage, the next instruction will be fetched immediately.

Program counter is managed in this stage. It is simple for continuous instructions. For branching instructions, method invocation and return instructions, the next instruction address is difficult to determine. This stage includes branch prediction and caches.

2.2 Instruction Decode (IDECODE)

In this stage, an instruction is decoded into Opcode, Branch Prediction Bits (BPB), Operand Descriptor Table Selectors (ODTS), OD10, OD11 and OD12. The Opcode is decoded into next stages’ control signals. According to the ODs and ODTS, the actual addresses of operand descriptors are calculated. Those addresses are passed to the next stage for OD fetching.

2.3 Operand Descriptor Fetch (ODFETCH)

According to the decoded logics, different numbers of ODs are fetched from memory. The major challenge in this stage is to fetch a maximum of three ODs simultaneously. A three-port OD cache is used. Once all the ODs needed are fetched, it will pass the ODs and the decoded logic to the next stage for further execution.

2.4 Operand Descriptor Check (ODCHECK)

The ODs would be checked with its validity, access right, etc. Any violation caused by the ODs would be handled as an exception. If the checking is passed, the actual addresses of data will be calculated. The address of data would pass to the next stage for data fetching.

2.5 Data Fetch (DFETCH)

With the given address from the previous stage, data is fetched from the memory. Again, a maximum of three data are required to be accessed in this stage simultaneously. A three-port data cache is used. Once all the data is fetched, it is passed to the next stage for execution.
On the other hand, the writeback address its control signals are passed to a latter stage.

2.6 Execution (EXEC)

Respective data would be scheduled into different units for execution, such as the ALU and FPU.

2.7 Writeback (WBACK)

The last stage of the pipeline. It would activate the data cache to store the result in the according address. An instruction comes to this stage as an end of execution.

3. Modules of jHISC

3.1 Stage Controller (SC)

In order to maintain the operation of different stages in the pipeline, Stage Controller is used for managing the flow and operation of each stage. In normal operation, it would perform as the program flows. For the communication between the stage controllers will use the concept of hand shaking. The stage controller will have basically two statuses, idle and busy. If the stage controller is idle, it waits for any operation from the previous stage and starts immediately. If the stage controller is busy, that means the stage is handling a previous operation from the previous stage and is not yet finished. So, any operation from the previous stage cannot pass to it and the pipeline will be suspended until the finish of the stage. These idea are illustrated in the preceeding figure.

3.2 Multi-port cache implementation on FPGA

Inside the FPGA, there are memory modules that can serve as caches for the processor. The total size of memory supported varies in different FPGA device.

Normally, the total size of memory on FPGA is about 10Kb. For the ease of implementation, we need to minimize the cache size in order to implement jHISC on FPGA.

Due to the limitations in FPGA, the cache implementation will be different. In order to implement jHISC as a pipelining machine, we will need to have a multi-port cache module in FPGA. A multi-port cache module can let more than one data to be read simultaneously in the same stage. The FPGA default memories are unable to support a multi-port cache, so, we design to build several buffers on top of the single-port cache module. The buffers and the cache works together to emulate the functions of a multi-port cache. Inside the buffers, it would store the data that is
recently requested by that port. In the jHISC processor, two caches modules are required to design as multi-port cache in order to fit in the pipeline. They are the Operand Descriptor Cache and the Data Cache.

3.3 Functional Modules
To implement jHISC on FPGA, we need to divide the design into different functional modules. These functional modules are responsible for managing different part of the machine. Basically, they can be divided into two categories, storage and execution. Storage modules take cares the storage elements in jHISC, like caches, registers, buffers, etc. They take cares the storage resources in jHISC. On the other hand, the execution modules are the workers of the machine. They produce the result of operations.

3.4 Storage Modules
In the HISC architecture, it uses many table lookup to support the concept of object-oriented programming. Since lots of the tables are involved, so different kind of storage elements are used to temporarily store the table contents. The management of these resources is done by the storage modules.

These storage modules includes, Instruction Cache (ICACHE), Instruction Buffer (IBUF), Operand Descriptor Cache (ODCACHE), Operand Descriptor Buffers (ODBUFs), Data Cache (DCACHE), Data Buffers (DBUFs), Special Function Register (SFR).

3.5 Instruction Cache (ICACHE)
Instruction Cache is a read-only cache that uses for the caching of instructions. It contains a finite state machine for the flow of cache management. Inside the Instruction Cache, there are memory elements for the temporarily storage of instructions. There are status rams to store the validity of instructions in cache ram. As instruction cache is only a read-only cache, no dirty bit is stored. Only valid bit and the tag are stored inside the status rams.

Whenever there is a request of instructions from other unit, given with an address to the instruction, the Instruction Cache will check up whether there is a copy in the cache ram. If it is the case, the instruction descriptor can immediately pass back to the request unit after one cycle. For cache miss, the Instruction Cache will send request to the memory. More than one instruction will be fetched and stored in the cache ram in order to minimize the chance of cache miss.

3.6 Instruction Buffer (IBUF)
Instruction Cache stores the instruction out of sequence, while Instruction Buffer stores the instruction in sequence as the program flows. The Instruction Buffer is made up of registers and a control unit. The control unit determines which is the next instruction that should be fetched. The instruction fetched will be buffered in the registers with its address appended.

3.7 Operand Descriptor Cache (ODCACHE)
Like Instruction Cache, Operand Descriptor Cache is a read-only cache. It is used for the caching of operand descriptors. It contains a finite state machine for the flow of cache management. Inside the Operand Descriptor Cache, there are memory elements for the temporarily storage of operand descriptors. There are status rams to store the validity of operand descriptors in cache ram. There is no dirty bit stored. Only valid bit and the tag are stored in the status ram.

Whenever there is a request of operand descriptor from other unit, given with an address to the operand descriptor, the Operand Descriptor Cache will check up whether there is a copy in the cache ram. If it is the case, the operand descriptor can immediately pass back to the request unit after one cycle. For cache miss, the Operand Descriptor Cache will send request to the memory. More than one operand descriptor will be fetched and stored in the cache ram in order to minimize the chance of cache miss.

3.8 Operand Descriptor Buffers (ODBUFs)
In the worst case, it needs to fetch three operand descriptors for a single instruction. In order to maximize the efficiency of the pipeline, we need to provide some buffers to emulate the function of a multi-port memory. The Operand Descriptor Buffers are made up of registers, with address, valid bit, and the corresponding operand descriptor stored.

When there is a request of operand descriptor from other unit, the Operand Descriptor Buffer checks whether there is a copy in the buffers. If no, it sends request to fetch the operand descriptor and fills up the buffers generally.

3.9 Data Cache (DCACHE)
Data Cache is used for the caching of data. Like any other cache, it contains a finite state machine for the flow of cache management. Inside the Data Cache, there are memory elements for the temporarily storage of data. There are also status rams to store the validity of data stored in cache ram. Data Cache is a read-write cache. It is designed to be a write back cache to minimize the interface between memories. Valid bit, dirty bit and tag are stored in the status ram.

Whenever there is a request of data from other unit, given with an address to the data, the Data Cache will check up whether there is a copy in the cache ram. If it is the case, the data can immediately pass to the requested unit after one cycle. For cache miss, the Data Cache will send the read request to the memory. Before
the new coming data is overwritten the data in cache ram, if any, the Data Cache would write back the data to the memory. After that, the requested data would be fetched into the Data Cache for the future use.

3.10 Data Buffers (DBUFs)

In the worst case, it needs to fetch three data for a single instruction. In order to maximize the efficiency of the pipeline, we needed to make some buffers to emulate the function of a multi-port cache. The Data Buffers are made up of registers, with address, valid bit, and the corresponding data stored.

When there is a request of data from other unit, the Data Buffer checks whether there is a copy in the buffers. If no, it sends request to fetch the data and fills up the buffers generally.

3.11 Special Function Registers (SFR)

In this module, it stores and manages all the special function registers used in jHISC. This module would not manage the content of special function registers itself. It provides just an interface for other units to access the registers. These special function registers include:

<table>
<thead>
<tr>
<th>Name</th>
<th>Short</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>PC</td>
<td>Stores the next running address</td>
</tr>
<tr>
<td>Status Register</td>
<td>SR</td>
<td>Stores the running mode, etc</td>
</tr>
<tr>
<td>System Stack Pointer</td>
<td>SSP</td>
<td>Points to the head of the system stack</td>
</tr>
<tr>
<td>Variable Stack Pointer</td>
<td>VSP</td>
<td>Points to the head of the variable stack</td>
</tr>
<tr>
<td>Class Operand Table Base Register</td>
<td>CODTBR</td>
<td>Points to the base address of CODT of the current running object/class</td>
</tr>
<tr>
<td>Class Operand Table Size Register</td>
<td>CODTSR</td>
<td>Stores the size of CODT of the current running object/class</td>
</tr>
<tr>
<td>Class Variable Pool Base Register</td>
<td>CVPBR</td>
<td>Points to the base address of CVP of the current running object/class</td>
</tr>
<tr>
<td>Class Variable Pool Size Register</td>
<td>CVPSR</td>
<td>Stores the size of CVP of the current running object/class</td>
</tr>
<tr>
<td>Instance Operand Table Base Register</td>
<td>IODTBR</td>
<td>Points to the base address of IODT of the current running object/class</td>
</tr>
<tr>
<td>Instance Operand Table Size Register</td>
<td>IODTSR</td>
<td>Stores the size of IODT of the current running object/class</td>
</tr>
<tr>
<td>Instance Variable Pool Base Register</td>
<td>IVPBR</td>
<td>Points to the base address of IVP of the current running object/class</td>
</tr>
<tr>
<td>Instance Variable Pool Size Register</td>
<td>IVPSR</td>
<td>Stores the size of IVP of the current running object/class</td>
</tr>
<tr>
<td>Method Operand Table Base Register</td>
<td>MODTBR</td>
<td>Points to the base address of MODT of the current running method</td>
</tr>
<tr>
<td>Method Operand Table Size Register</td>
<td>MODTSR</td>
<td>Stores the size of MODT of the current running method</td>
</tr>
</tbody>
</table>

3.12 Execution Modules

3.12.1 Floating Point Unit (FPU)

For jHISC, it targets to support Java in the hardware level. As the floating point arithmetic is using IEEE 754 standard in Java, the Floating Point Unit would performs the calculation of IEEE 754 floating point arithmetic for addition, subtraction, multiplication and division, etc. Both 32-bit and 64-bit would be supported for single and double precision floating points.

In the Floating Point Unit, it would check for any special case that cause an exception throws in hardware. These exceptions includes divide by zero, overflow, underflow, truncation error, etc.

3.12.2 Arithmetic and Logic Unit (ALU)

For any other operations in jHISC, it would be done by the Arithmetic and Logic Unit. It performs all the logic functions, like AND, OR, XOR, etc. Besides, it supports signed integer arithmetic, as it would be in Java, like ADD, SUB, MUL, DIV, etc. Through the use of these simple operations, the jHISC processor can support other more complicated operations. The control flow of the complicated operations like, object manipulation operations, are managed by the Stage Controllers of the pipeline.

Like the Floating Point Unit, it would check for any errors occurred for the input data set. In case of errors, it would raise exceptions to the pipeline and Stage Controllers would manage the flow of exception handling.
3.13 Access Control Unit (ACU)

Operand descriptor stores the access control information about data. For Access Control Unit, it analyzes the operand descriptor and sees whether the operation is valid. It checks whether the requested data is valid to read or write. These units serve many kinds of access right checking. But the detailed procedure is out of scope of this paper.

3.14 Interrupt Controller (INTR)

jHISC supports hardware interrupt. Interrupt signal pins are used for the outside world to signal an interrupt event. When the Interrupt Controller receives this event, it would signal the Stage Controllers to suspend the pipeline and start to serve the interrupt routines. The details implementation of this module is not included in this paper.

4. Conclusion

Everyone is talking about objects. Practicing the concept of Object-Oriented Programming is the trend of software development. OOP gives lots of advantages that traditional Procedure Oriented Programming style don’t have. These advantages include the ease of porting, the ease of maintenance, the enhancement of security, etc. Hardware supports of OOP are more important. Therefore, a processor that can provide support in OOP is proposed. High Level Instruction Set Computer (HISC) is a newly defined architecture that provides object mappings with the use of tables. It is a 64-bit architecture, which extends from the traditional descriptor computer to support OOP, better access control, and multimedia application.

An implementation of HISC architecture is taking, which currently named jHISC. This implementation uses the concepts in HISC to support JAVA, which is the most popular OOP language now practicing. The implementation is targeted to the level of FPGA at the first stage. Seven pipeline stages are defined for jHISC. They are Instruction Fetch, Instruction Decode, Operand Descriptor Fetch, Operand Descriptor Check, Data Fetch, Execute, and Writeback. The major issue about implementing jHISC on FPGA is on the cache implementation. Inside FPGA, it has limitations on the memory elements provided for the implementation of cache. Thus, buffers are proposed to serve the function of a multi-port cache. Different modules are defined, which supports different part of operations in jHISC. Base on the design of different modules and pipeline stages in jHISC that targets on FPGA, implementation with VHDL is currently in progress.

Acknowledgement

This study is partially funded by Strategic Research Grant 7001056 of City University of Hong Kong.

References


