Computers and Electrical Engineering 38 (2012) 746–755

Contents lists available at SciVerse ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

HISC: A computer architecture using operand descriptor ☆

Yijun Liu [a,*], Anthony S. Fong [b], Fangyang Shen [c]

[a] Faculty of Computer, Guangdong University of Technology, University Mega Center, Guangzhou 51006, China
[b] Department of Electronic Engineering, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong
[c] Department of Computer Systems Technology, New York City College of Technology, CUNY, 300 Jay Street, Brooklyn, New York, 11201, USA

ARTICLE INFO

Article history:
Available online 11 January 2012

ABSTRACT

Computing has been evolved from number crunching to today's cloud. Data are no longer numbers but information which needs to be appropriately guarded and easily transportable, but the original von Neumann instruction model does not support them architecturally. This led us to start a new architecture named HISC (High-level Instruction Set Computer), to attach attributes to individual operand on instruction for effective and efficient processing of today's computing. HISC instruction consists of an operation code (opcode), and an index to source or destination operand referenced by an operand descriptor, which contains value and attributes for the operand. The value and attributes can be accessed and processed in parallel with execution stages, introducing zero or low clock cycle overheads. Object-oriented programming (OOP) requires strict access control for the data. The JAVA model, jHISC, executes Java object-oriented program not only faster than software JVMs but has less cycles-per-instruction than other hardware Java processors. We also propose future extensions for operand descriptor beyond OOP.

1. Introduction

Since the introduction of computers, the computer architecture is typically von Neumann architecture [1], with stored-program machine concept. In this model, a problem-solving algorithm is represented by a string of computer instructions, which are executed to manipulate the input data and generate output data. Instructions to be executed, together with data to be manipulated, are stored in memory. The instructions are executed sequentially, until branching or traps take place with program control instructions such as branch and conditional branch are encountered. In addition, there are I/O instructions to interface with the external.

An instruction of a von Neumann architecture computer is typically composed of an opcode (operation code) and zero, one or more operands. The operands are data in either registers or memory, or addresses to memory. The register or registers may be explicitly specified or implicitly implied by the opcode. There are no qualifiers that specify the nature and system attributes for the data except those implied by the opcode. For examples, an integer addition instruction implies the data specified by the operands are integers of certain lengths, and a floating-point instruction implies the operand data are floating-point numbers of other formats and lengths. Any other descriptions and qualifications of the data of the operands are specified by other data which are numbers and patterns of certain pre-assumed data types and meanings in other instructions. The data are to be interpreted by other programs or procedures, in order to carry out the appropriate operations or to inform these qualifications and meanings by the machine. They may be grouped as system attributes of the data in the operands.

☆ Reviews processed and proposed for publication to Editor-in-Chief by Guest Editor Prof. Mei Yang.
* Corresponding author. Tel.: +86 20 3932 3332.
E-mail address: yjliu2002@163.com (Y. Liu).

0045-7906/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compeleceng.2011.10.010

1.1. Object-oriented programming

Computing style since then has been evolving from the original von Neumann to structure programming [2], and more recently, to object-oriented programming. Object-orientation concept started in early 1970's with Ada by the United States Department of Defense [3]. Today Java, C#, J2EE, and .NET are among the most popular technologies, because they provide better productivity, sharing, and security, especially in Internet computing [4–7]. The conventional von Neumann programming model has little protection support such as paging with TLB and segmentation, and security has been compromised.

There were attempts to incorporate access control for security into computer architecture by the computer industry in the 1970's when Ada was introduced, many with so-called capability machines such as Intel iAPX432 [8–10], IBM System 38, Data General Fountainhead, and the less-known then Univac and Burroughs projects (now UNISYS). There are United States patents issued on object-oriented processors, and the more interesting ones are [11,12]. More recent machines include Sun Microsystems PicoJava II [13–15], ARM Jazelle [16] and Fujitsu MB86799 Evaluation Chip (PicoJava II core). There are also academic research projects such as Rekursiv [17], Vijaykrishnan et al. [18], JOP [19], and others offer partial solutions. More recently, works have been done on instruction folding [20–22] and dynamic compiling [23,24].

1.2. Access control

Object-orientated programming (OOP) was introduced to support reliability, reuse, sharing, and security issues [25]. All these require appropriate access control. Object-oriented programming partitions a program into *objects*. In OOP, an *object* is a combination of a *method* (procedure code) and *data*. *Methods* are defined within objects as interfaces of communication. A method is an object-oriented version of subroutines. The calling of methods is named *method invocation* in object-oriented programming. In OOP, communication among objects in generally only performed by *method invocation*. In contrast with the subroutine calls in non-OOP system, method invocation provides a secure version of control transfer from one domain to the other. Because control transfer is secured in an OOP system, multiprogramming can be achieved by the use of *threads* instead of *processes*. In contrast with the process, a thread shares *address space* with other threads, which means the overhead of *context switching* between processes can be avoided.

As development of computers and program languages advances, programming does not only focus on the rough data of an operand but also its natural and system attributes. For example, in OOP languages, such as Java, an operand requires multiple attributes to support object-oriented features, such as abstraction, encapsulation, polymorphism, etc. [25]. They are generally handled by software. Therefore there are significant overhead which creates performance problems, and the uncertainty of full and correct complying by software. In an object-oriented variable access instruction, apart from a 'read' action, system needs to check if access is legal. Many attributes need to read from memory, such as access mode, class field size, base address, indirect addressing and type etc. Therefore there are multiple memory accesses for a single data item, and a single object-oriented (OO) instruction may require hundreds of clock cycles.

To speed up processing OOP and beyond, we propose a new architecture named HISC: High-level Instruction Set Computer, and introduce an instruction extension called operand descriptors (OD) [26]. Each operand in HISC is accompanied with an operand descriptor. The operand descriptor (OD) describes the value and attributes of an operand. A typical HISC instruction consists of an operation code, and indexes to source and destination operands referenced by operand descriptors [27]. The value and attributes of an operand can be accessed concurrently in an instruction, making HISC execute faster than conventional computers.

The remainder of the paper is organized as follows: Section 2 describes the architecture and the design philosophy of HISC. Section 3 proposes jHISC – a HISC processor that supports Java object-oriented program. The implementation and performance evaluation of jHISC are presented in the section. Section 4 presents the future investigation opportunities in multicore, multiprocessing environment and cloud computing. Section 5 concludes the paper.

2. HISC architecture and philosophy

A typical von Neumann architecture instruction uses a three-address format as following:

$$\mathrm{DST} \leftarrow \mathrm{SRC0} < \mathrm{op} > \mathrm{SRC1},$$

in which, the first address is the destination in memory, the second address is the first of the two source operands in memory, and the last address is the second of the two source operands in memory. The opcode (op) is the binary operation specified by the instruction. Fig. 1 shows a typical von Neumann machine instruction with 3-address instruction format. There are no qualifiers that specify attributes for the data except those implied by the opcode. The implied attributes are data types, such as 16-bit, 32-bit integers, or single or double precision floating-point numbers.

As the development of software progresses, operands are assigned a number of attributes to support more and more complex applications. These attributes need to be further interpreted by other programs or procedures, in order to enforce or to inform the qualifications and meanings by machines. To interpret the data in the operands with qualifications and attributes, other data are used and they must be processed by subroutines and programs to implement the qualification and attributes, and verifications if so required. Fig. 2 shows the relationship.

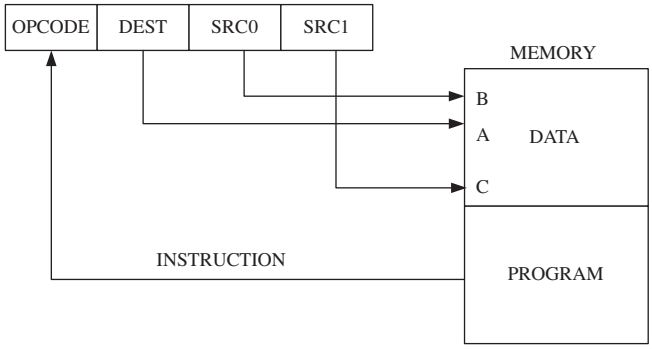

Fig. 1. A three-address instruction format of von Neumann architecture.

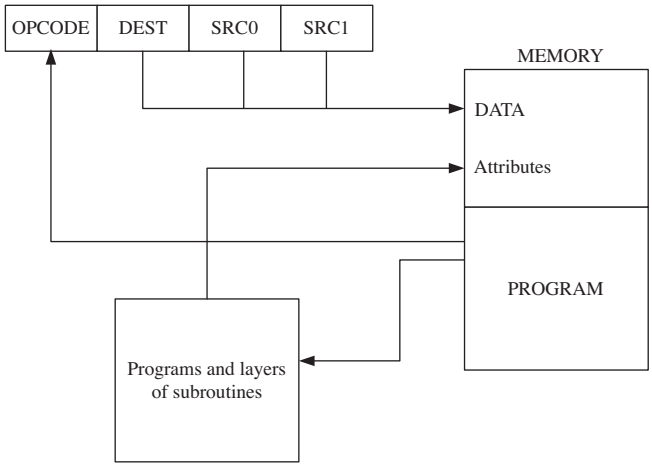

Fig. 2. A three-address machine of von Neumann architecture whose operands require qualification checking or attributes enforcements.

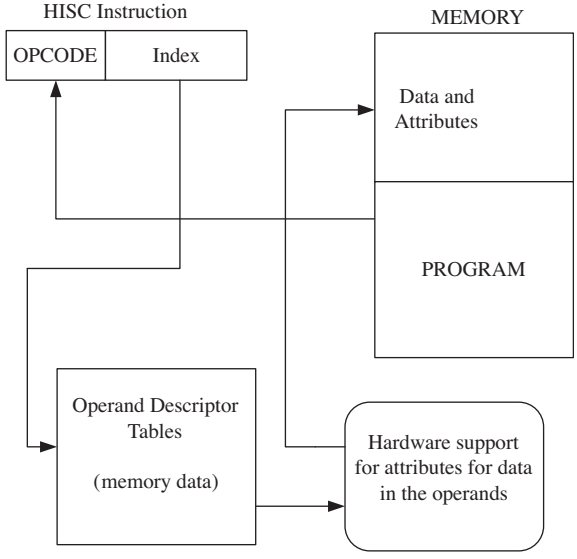

Fig. 3. A three-address HISC instruction whose operand attributes are stored in operand descriptor and directly interpreted by hardware.

Conventional von Neumann computers interpret and execute one instruction at a time, and one instruction fetches one data or attribute. Moreover, subroutine invoking operation causes many clock cycle overhead. This software-based data with many attributes processing mechanism (as shown in Fig. 2) is inefficient.

HISC evolves from CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer). Being a "high-level instruction set" computer, its instructions match rather closely with programming languages. For example, it supports some Java OO instructions directly, which is presented later in the paper.

The core component of HISC is an instruction extension called operand descriptors (OD). An operand descriptor contains data value as well as its attributes. Due to operand descriptor, data value and attributes can be accessed at the same time and processed in parallel using CPU and dedicated hardware, and therefore increasing performance. A typical HISC OOP instruction consists of an operation code, and indexes to source or destination operands referenced by operand descriptors. The difference of servicing the attributes and doing the qualification checking with the classic von Neumann model shown in Fig. 2 and the extended descriptor architecture of HISC shown in Fig. 3, is that the formal requires to process by executing instructions in system programs and subroutines, while the latter requires a pipelining stage of fetching the operand attributes to be interpreted or checked by hardware. The enforcement of the operand attributes is done concurrently with the execution stage, and hence no additional execution time is introduced.

The most important characteristic of HISC is its data manipulation granularity. The system attributes reside in the operand descriptor for each instruction operand. The operand descriptors are stored in Operand Descriptor Tables. Thereby natural and system attributes are specified for individual instruction operand, to achieve very fine operand granularity, while in conventional computers, data manipulation granularity is coarse.

3. jHISC – a HISC supporting JAVA

3.1. Operand descriptor format

In this section, an implementation of HISC computer is proposed, to support JAVA, a dominating object-oriented programming (OOP) language. The essence of OOP is the ability to assign and guide the access control on individual programming parameters.

Since a programming variable can be of many different forms and various sizes, it is essential for jHISC to be able to assign and guide the access control on various lengths of data and instructions. This is transformed to programming variable, to achieve individual programming variable granularity. Because OOP often has accesses to different objects, whose attributes vary from one another without any regular or fixed forms, this operand granularity substantially improves performance. In today's computing systems, the manipulation must be handled in higher software levels, to achieve the effects of system attributes on individual programming variables.

In order not to suffocate performance, many OOP languages such as Java and C# do not require the compliance checking of data types for primitive types in the classes, such as integers, and floating-point numbers in Java. They would issue these straight-forward instructions and let the processors execute them. The processors are usually not able to detect incorrect types, but if there are any damages caused by such errors, they will be confined to the individual objects. Using operand descriptors, jHISC checks the correctness of every object and operands in hardware, with greatly improved performance and security of computer.

The original High Level Instruction Set Computer (HISC) architecture is implemented in a 64-bit processor. It extends typical computer architecture to support object-oriented programming at the hardware level by introducing 128-bit operand descriptors to describe both object references and variables. Each operand descriptor, residing in an operand descriptor table, is maintained by operating system and is read-only to user programs. The original HISC proves that the concept and design philosophy are valid and feasible, but the additional circuit to implement is quite significant. To reduce the circuit of the original HISC, and to spearhead Java, we proposed jHISC of 32 bits. The research work of jHISC was partly published [28,29]. The remaining part of the section is extended from them.

In jHISC, we simplified the operand descriptor to 32 bits according to the Java specification [6] and defined a uniform format shown in Fig. 4. An operand descriptor includes Address Field, Type Field, Static Flag, Access Modifier, Read-Only Flag, and Resolved Flag. The function of each field is introduced as follows.

- Address Field provides a byte offset to locate data in the corresponding data spaces.
- Access Modifier is used for security control, and four access modifiers (public, private, protect and package) are defined in the current system.

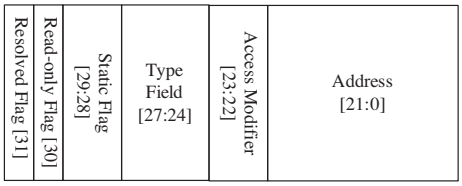

Fig. 4. Operand descriptor format in jHISC.

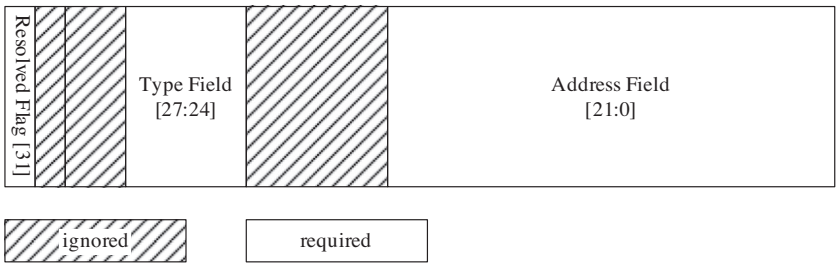

Fig. 5. Class operand descriptor format in jHISC.

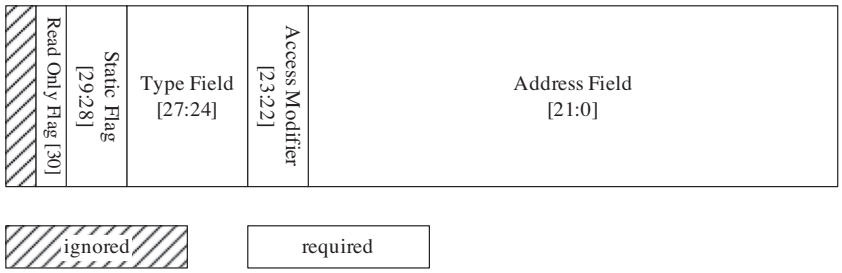

Fig. 6. Class property descriptor format in jHISC.

- Type Field stores the data types defined for both primitive and reference, such as byte, integer, word, and reference, and so on. The primitive data are stored inside data space and, for the references; direct addresses are stored to locate the described resource.
- Static Flag indicates where the data are stored. For non-static fields, data are stored in the instance data space (IDS). For static fields, data are stored in the class data space (CDS). When a static field is inherited from a class extension, a direct address pointing to it is stored in the CDS.
- Read-Only Flag denotes whether the target can be written.
- Resolved Flag indicates whether the reference is resolved or not. If not, the system will be trapped to the operating system routines for the dynamic reference resolution.

Two kinds of operand descriptors, class operand descriptor (COD) and class property descriptor (CPD), shown in Fig. 5 and Fig. 6 respectively, are also defined in jHISC to assert the resources accessed by the class and the properties owned by the class, respectively. A class operand descriptor contains the Address Field, Type Field and Resolved Flag. In a class property descriptor, the Resolved Flag is not included.

3.2. Object representation in jHISC

The object representation method is critical in an object-oriented programming system because of its effects on the speed of accessing objects. In jHISC, an object is represented by the hardware-readable data structure–object context, which consists of object header, data space and the corresponding descriptor tables, and so on. These are the core structure of our system, with the ISA to define the whole architecture as stated in [30]. Three kinds of contexts, namely instance, class, and method contexts, are mapped to the hardware architecture and distinguished by the object header (OH), which is shown in Fig. 7.

The functions of each field in an object header are defined as follows.

- objType stores the object type, such as instance, class, method and array.
- dsSize specifies the size of related data space, for example, CDS, IDS and Method Code Space.
- gcInfo is reserved to give hardware support for real-time garbage collection in the future; garbage collection is performed by the operating system in the current version.
- Class links an instance object with its affiliated class through a reference pointer.
- ArraySize and arrayType specify the number and type of elements in an array, respectively, when the object is an array.

Other than object header, an instance context also includes instance header (IH) and instance data space (IDS); a class context also contains class header (CH), class operand descriptor table (CODT), class property descriptor table (CPDT) and

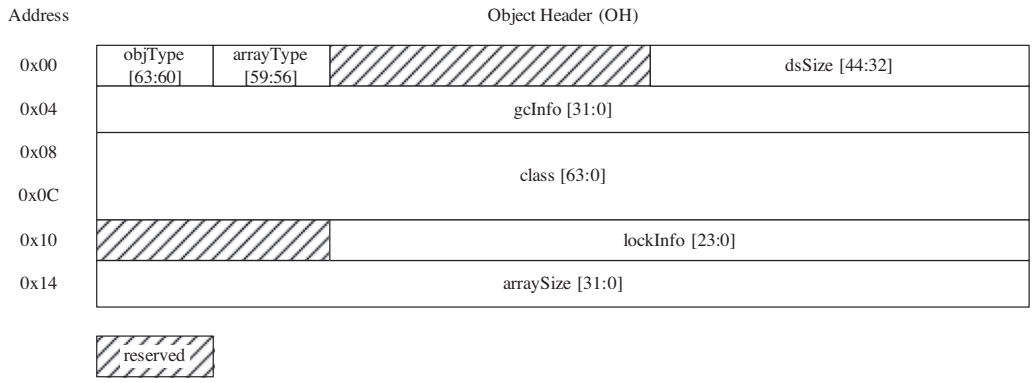

Fig. 7. Object header format in jHISC.

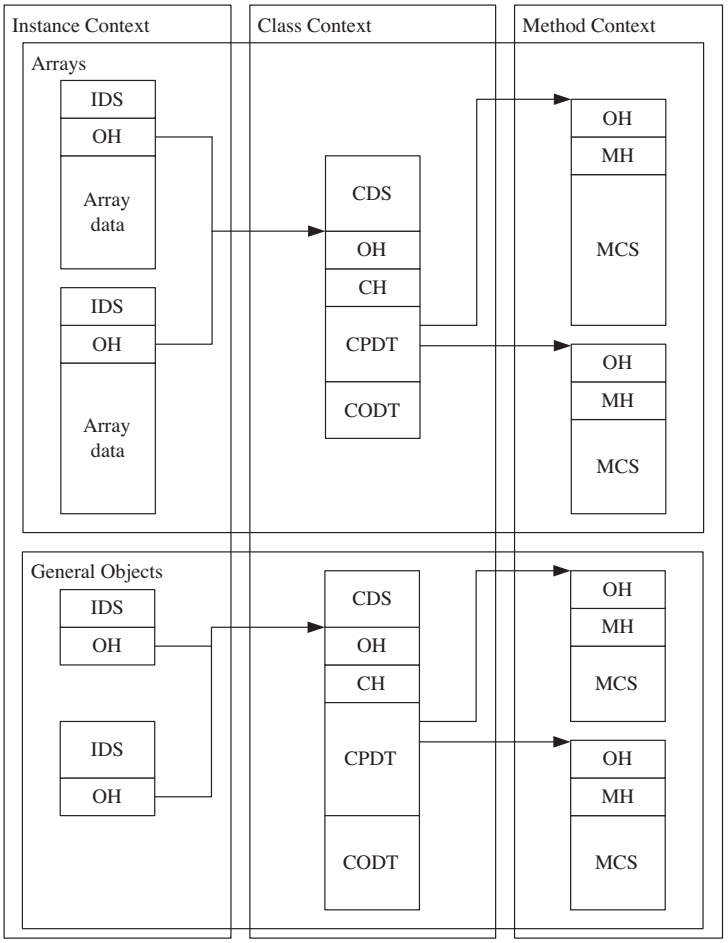

Fig. 8. Different object structures and their relations in jHISC.

class data space (CDS); a method context consists of method header (MH), method code space (MCS) and local variable frame (LVF). When used to represent an array, an instance context also contains an array data area under the instance header. Inside class context, CODT and CPDT store class operand descriptors and class property descriptors, respectively. The different object context structures and their relations are shown in Fig. 8.

Object manipulation mainly includes object creation, method invocation and revocation, getting or putting an object field. In Java virtual machine, different objects are created by the related instructions ‘‘new’’, ‘‘newarray’’, ‘‘anewarray’’ or ‘‘multianeway’’. Static object fields are manipulated by using the instructions ‘‘getstatic’’ and ‘‘putstatic’’. Instances are

```
Class Orange
{
    public void eat() {
    ..
    ..
    }
}

Class Instance_Method_Example
{
    public void Caller() {
        Orange orange = new Orange();
        orange.eat();
    }
```

Fig. 9. A Java program about a method invocation.

accessed through the instructions "getfield" and "putfield", and methods are invoked by using the instructions "invokeclass" and "invokevirtual". These operations constitute about 15% of all the operations [29] in the profiled benchmark. Their executions have significant impact on the execution speed of Java program.

In the Java virtual machine, an object is initially referenced symbolically and the manipulation instructions are converted to their quick formats after the object is resolved. In most of the current solutions to Java virtual machine in hardware, object manipulation instructions are still being executed initially by software traps, and once the related objects are resolved, the quick formats are executed through microcode or other approaches to speed up execution because some bound checks are not performed after object resolution. However, due to the stack architecture of Java virtual machine, in the initial execution of object manipulation operations, most of the time is spent in initializing the operand stack to prepare for operations, retrieving reference information and verifying whether the reference is resolved or not, checking access permission, and so on.

In jHISC, objects are described by the operand descriptors and their reference addresses or values are stored in CDS as shown in Fig. 8. When an object is accessed, the related operand descriptor is read from the operand descriptor tables to verify whether the object is resolved or not, then the specific object header is accessed and the bound control checks are carried out along with the access. Moreover, in the object context, all fields are stored with a constant offset to the object header. Thus they can be accessed in parallel to speed up execution once the address of the object header is obtained. On the other hand, though the quick formats of object manipulation instructions are executed after objects are resolved in some solutions. If the host machine is stack architecture, the instruction level parallelism is prohibited because the Java virtual machine is stack architecture and only one instruction is executed at a time, which affects the system performance. Moreover, because the architecture of the host machine does not provide hardware support on object-oriented programming, the execution of object manipulation instructions is time-consuming.

In jHISC, most of the main object-oriented instructions are implemented in hardware directly and special hardware is provided for object-orientation and object accesses, such as operand descriptor table, which results in the improvement of system performance. For example, as shown in Fig. 9, a virtual method orange.eat() is invoked in the method Caller(). The object context switches and object structures are illustrated in Fig. 10.

In the current method space, the instruction *ivkinstance R0 #1* triggers a virtual method invocation.

The instruction specifies that the class operand descriptor about the invoked method *Orange.eat*() resides in the entry #1 of CODT in the current class context and the reference of the instance *orange* is stored in the register *R0* inside the current LVF. The execution includes nine steps shown as follows:

(1) Read the operand descriptor from the entry #1 of CODT in the current class *Instance_Method_Example* and the instance reference from the register R0. The operand descriptor provides two offsets, one for getting the reference of the class *Orange,* and another for accessing the method reference *public void eat*() inside CPDT of the class *Orange*. We assume the offsets equal to 0 in the example.
(2) Save the instance context and get the operand descriptor about the class *Orange* from the entry #0 of CODT in the class *Instance_Method_Example*.
(3) Obtain the direct address of the class Orange from CDS in the class *Instance_Method_Example*, and verify whether the instance *orange* is an instance of the class *Orange* or not. If not, exception will be thrown in order to resolve the requested reference again.
(4) Access OH and CH of the class *Orange*, save the class context, and switch the current class context from the class *Instance_Method_Example* to the class *Orange*.
(5) Read the property descriptor about the invoked method *eat*() from the entry #0 of CPDT in the class *Orange*.
(6) Obtain the direct address of the method *eat*() from CDS in the class *Orange*.
(7) Access OH of the method *eat*().

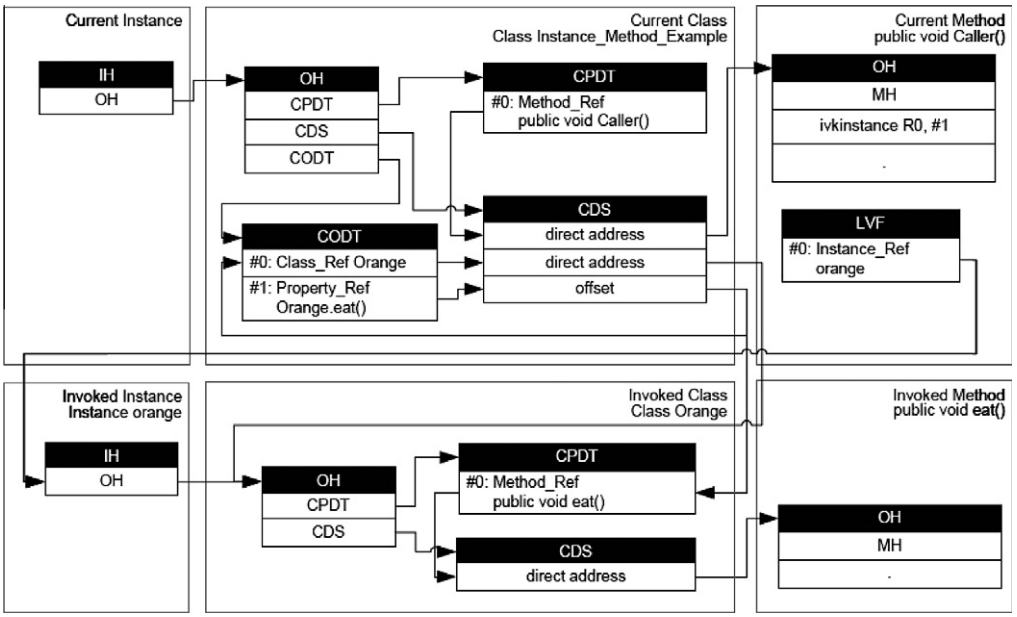

Fig. 10. Object context switching procedure.

(8) Access MH of the method *eat*(), save and update the method context.
(9) Allocate a new method frame for the method *eat*() and update the current method context.

If all data hit in cache and the object has already been resolved, the invocation will consume 9 cycles with each step consuming one cycle. However, in PicoJava II with the same assumption, it takes 195 cycles through software trap and its quick format consumes 15 cycles [30].

jHISC uses operand descriptors (OD) to support object-oriented programming at hardware level and support OO operations directly, in which, clock cycles of object-oriented operand access are greatly reduced.

3.3. jHISC performance evaluation

To verify the architecture of HISC, a FPGA jHISC processor is implemented. The processor with 4 kB instruction cache and 8 kB data cache was designed with VHDL and implemented through a Xilinx Virtex FPGA XCV800. The processor is designed having six pipeline stages: instruction fetch, instruction folding (in hardware) and translation, instruction decoding, data fetch, execution and write-back. The whole system occupied 601,131 equivalent gates in FPGA and its maximum clock frequency was about 30 MHz.

We implemented jHISC in a rather old FPGA chip only to verify the function correctness of the architecture. It is unfair to compare the speed of FPGA jHISC with other JAVA chips. A more reasonable performance metric for comparison is processors CPI (average clocks per instruction). We compared the CPI of jHISC with two hardware Java processors called PicoJava II [30] and JOP [31,32]. The clock numbers per instruction are listed in Table 1.

As can be seem from the table, jHISC is much faster than PicoJava II and JOP in terms of CPI. We compared the performance of the three hardware Java processors and a software Java interpreter (JDK1.5.0_05) using 8 benchmarks in SPEC JVM98 benchmark suit (_200_check, _201_compress, _202_jess, _209_db, _213_javac, _222_mpegaudio, _227_mtrt, _228_jack) [33]. The overall performance comparison is shown in Fig. 11.

In summary, jHISC uses an operand descriptor architecture with a special operand access and manipulation mechanism which boost the performance in executing Java bytecodes. According to Table 1, most object-oriented bytecodes could be executed with fewer cycles in jHISC processor due to the unique architecture for the enhancement in object manipulations. Based on our performance estimation, jHISC processor achieves relatively highest performance gain compared with other Java execution methods, including the JVM HotSpot, picoJava II and JOP.

4. Further investigations

The HISC structure is expandable for additional attributes, to fit in today's computing environment and needs.

The operand descriptor could be expanded with two fields: caching coherency and caching enabling. They will greatly simply the logic to maintain cache coherency in a multiprocessing environment. In a multiprocessing system, maintaining

Table 1
Cycles needed by some main object and array manipulation instructions.

Bytecodes	PicoJava II Cycles		JOP	Instruction in jHISC Cycles		
	Original format	Quick variant		Instruction	Original format	Quick variant
getfield	114	4	12	gfld	3	2
				gifld	2	2
agetfield_quick	4					
putfield	130	4	15	pfld	3	2
				pifld	2	2
getstatic	103	3	6	gsfld	3	2
agetstatic_quick	3					
putstatic	103	3	7	psfld	3	2
invokestatic	86	11	67	ivkclass	6	6
invokevirtual	195	15	88	ivkintance	6	6
				ivkinternal	4	4
invokespecial	208	17	67	ivkintance	6	6
invokeinterface	203	184	96			
checkcast	97	6		checkcast	3	3
instanceof	100	7		instanceof	4	4
ireturn	8		19	oo_rvk	5	5
return			17			
areturn			19			
return			19			
iaload	5		24	arrayload	3	3
aaload						
caload						
iastore	7		26	arraystore	3	3

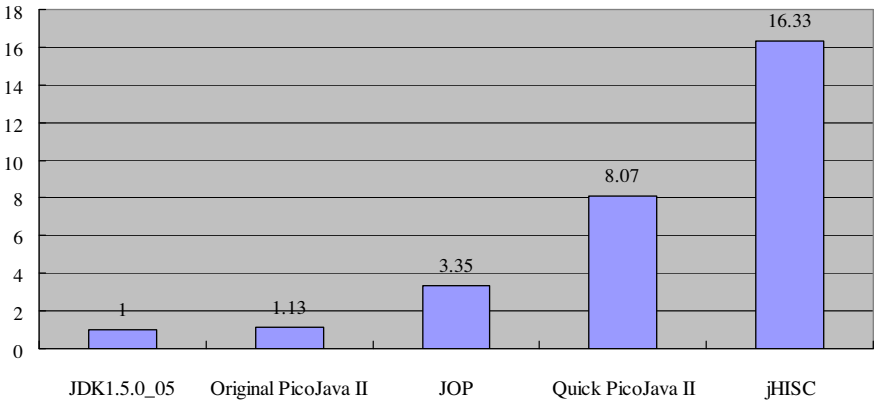

Fig. 11. Relative performance based on CPI.

data coherency is tedious and resource demanding. In a 100-core system, when introducing a piece of data into a core, the data must be checked with the other 99 cores to see if appropriate actions such as setting dirty flag, or invalidating the data must be carried out. These two fields in the descriptor will greatly reduce the amount of traffic and actions to be carried out. In today's computing, most data are read only. More details are described in a US patent [34].

The descriptor could also include data source requirement. In this era of cloud and web computing, web data required for computing is quite likely to reside outside the host system. To facilitate computing, it may be beneficial to classify the source requirement for the data, so that the required data may be obtained from a specific place or a convenient place. The operand descriptor concept could also replace embedded processor in many occasions. It could have application-specific fields in operand descriptor, which will lead to execution by hardware. The circuit may be implemented in programmable logic or reconfigurable logic circuit.

5. Conclusions

In this paper we have proposed a new computer architecture called HISC, in which, from operand descriptor, operand value and attributes are accessed at the same time and processed in hardware concurrently with execution stage, and hence no additional execution time is introduced. To verify the architecture, we designed jHISC processor with FPGA to support

JAVA. Using the mechanism of object descriptor, jHISC handles operand access and manipulation in hardware, and saves clock cycles in object-oriented operations, such as security checking and object invoking, and thereby results in a small average CPI.

Based on the experiments of benchmarks, it has been shown that jHISC is not only faster than Java software interpreters but also some hardware Java processors. In counting numbers of cycles per specific object instructions, jHISC outperforms PicoJava II and JOP by a multiple, and up to 16-to-1 overall CPI performance advantage over the four chosen machines.

Acknowledgement

The authors would like to thank the team members of Project HISC, particularly Moky Mok, Steven Tan, and Jerry Yau. The work described in this paper was partially supported by Strategic Research Grant (7002602) from the City University of Hong Kong and also supported by National Natural Science Foundation of China (61106019).

References

[1] Hayes J. Computer Architecture and Organization. McGraw-Hill; 1998. ISBN 1-07-115997-5, p. 18.
[2] Knuth, Donald E., Von Neumann's First Computer Program. Computing Surveys, 4, 1970, pp. 247-260.
[3] Booch G., Bryan D., Software Engineering with Ada. 3rd ed. The Benjamin/Cummings Publishing Co., 1994, ISBN 0-8053-0608-0.
[4] T. Archer, "Inside C# (With CD-ROM)", Microsoft .Net, 2001.
[5] Craig I. The Interpretation of Object-Oriented Programming Languages. Springer; 1999. ISBN 1-85233-159-3.
[6] Horstmann C.S., Cornell G. Core JAVA 2 Volume I – Fundamentals. 5th ed. Prendice-Hall/Sun Microsystem Press, 2000. ISBN: 0-13-089468-0.
[7] Horstmann C.S., Cornell G. Core Java 2, Volume 2: Advanced Features. Prendice-Hall/Sun Microsystem Press, 2001. ISBN: 0-13-081934-4.
[8] Hansen PM, Linton MA. A performance evaluation of the iAPX432. Comput Archit News 1982;10(4):, p.
[9] Hunter C., Farquhar E., Ready J., Introduction to Intel iAPX 432 Architecture. Reston Publishing Company, 1985, ISBN 0-8359-3222-2.
[10] Organick E. A Programmer's View of the Intel 432 System. McGraw-Hill Book Company; 1983. ISBN 0-07-047719-1.
[11] Bottomley, Thomas, International Application Publication Number: WO 02/48864 A2, "System registers for an object-oriented processor", COTTO Wireless, Inc. of La Jolla CA, 20 June 2002.
[12] Chen Tao Shinn, US Patent 6003,038, Object-oriented processor architecture and operating method by Sun Microsystems, Dec. 14, 1999.
[13] O'Connor JM, Tremblay M. PicoJava I: The Java Virtual Machine in Hardware. IEEE Micro 1997;17(2):45–53.
[14] McGhan JM, O'Conner. PicoJava: a direct execution engine for java bytecode. Computer 1998:22–30.
[15] Sun Microsystems. PicoJava-II: Java processor core. April: Sun Microsystems data sheet; 1998.
[16] ARM, "Jazelle technology for Java application", ARM data sheets, May 2001.
[17] Harland D. Rekursiv, Object-Oriented Computer Architecture. Ellis Horwood Ltd; 1988. ISBN 0-7458-0396-2.
[18] Vijaykrishnan N., Ranganathan N., Supporting Object Accesses in a Java Processor. IEE Proceedings- Computers and Digital Techniques, vol. 147, No. 6, November 2000, pp. 435–443.
[19] Schoeberl, Matin, "JOP – A Java Optimized Processor for Embedded Real-time Systems", PhD Dissertation 2005, http://www.jopdesign.com/.
[20] Radhakrishnan R, Vijaykrishnan N, John LK, Sivasubramaniam A, Rubio J, Sabarinathan J. Java runtime systems: Characterization and architectural implications. IEEE Trans Comput 2001;50(2):131–46.
[21] Tan Yiyu, Yau Chihang, Fong Anthony S, Yang Xiaojian. An instruction folding solution for a Java processor. Comput Syst Sci Eng 2009;24(3).
[22] Yau Chi Hang, Tan Yi Yu, Mok Pak Lun, Yu Wing Shing, Fong Anthony S. A Hardware/Software Co-design and Co-verification on a Novel Embedded Object-Oriented Processor. Lecture Notes in Computer Science, vol. 3824, pp. 371–3805. 2005, Springer.
[23] Suganuma T., Yasue Y., Kawashito M., Komatsu H., Nakatani T. Design and evaluation of dynamic optimizations for a Java Just-In-Time compiler. ACM Trans. On Programming Languages and Systems, vol. 27, No. 4, July 2005 pp. 732–785.
[24] X. Jia, "Object-Oriented Software Development Using Java", Addison Wesley, 2000, ISBN 0-201-35084-X.
[25] A. S. Fong, "HISC: A High-level Instruction Set Computer", Proceedings of 1995 European Simulation Symposium, pp 406-410, 26-28 Oct, Erlangen, Germany. ISBN 1-56555-083-8.
[26] Fong A.S. A Computer Architecture with System Attributes on Individual Operands, ISCA Int'l Conference on Computers and Their Applications, pp 258-261, March 1996, San Francisco, USA
[27] Yiyu Tan, Yau Chi Hang, Anthony S. Fong, "An Object Model for Java and its Architecture Support". 6th International Conference on Information Technology: New Generations, 2009, pp. 831–836.
[28] Tim Lindholm and Frank Yellin, "The Java Virtual Machine Specification", Addison Wesley, 1999, 2nd Edition, ISBN 0-201-43294-3.
[29] Yiyu Tan, Lo Wan Yiu, Yau Chi Hang, Richard Li, Anthony S. Fong: A Java processor with hardware-support object-oriented instructions. Microprocessors and Microsystems, Elsevier, Dec 2006 pp. 469–479.
[30] Sun Microsystems. PicoJava-II: Java Processor Core. April: Sun Microsystems data sheet; 1998.
[31] M. Schoeberl, "JOP: a java optimized processor", Lecture Notes in Computer Science, Vol. 2889, October 2003, pp. 346-359.
[32] M. Schoeberl, JOP: a java optimized processor for embedded real-time systems. PhD dissertation, 2005, http://www.jopdesign.com.
[33] SPEC: JVM98 Benchmark suits. http://www.spec.org/jvm98/.
[34] Fong, A. "A Computer Architecture with Access Control and Cache Option Tags on Individual Instruction Operands", ACM Special Interest Group (SIG) Computer Architecture News, v31 n 3, pp 1-5, June 2003.

Yijun Liu, received a B.S. degree from Beijing Normal University, a MSc degree from Guangdong University of Technology, China, a M.Phil degree and a Ph.D degree from the University of Manchester, UK, all in Computer Science. He is currently Professor in The Faculty of Computer at the Guangdong University of Technology, Guangzhou, China. His interests include computer architecture, Network-on-Chip, VLSI design and embedded systems.

Anthony S. Fong, received his B.E.E. degree from Villanova University, Pennsylvania, M.Sc. degree in Computer Science from the State University of New York at Buffalo, and Ph.D. from University of Sunderland. He worked at Digital Equipment, Data General, and Wang Laboratories, designing computing systems. At present, he is Associate Professor in Electronic Department at the City University of Hong Kong. He has been awarded nine United States patents on computer architecture and design. His interests include computer architecture and design.

Fangyang Shen, received a BEng degree and a MEng degree from Guangdong University of Technology, P.R. China, majoring in computer science and engineering, a PhD degree from Auburn University, USA, majoring in computer science and software engineering. He is currently an assistant professor in Department of Computer Systems Technology at New York City College of Technology (CUNY), USA. His research interests include wireless networks, computer architecture and computer education.