An Object Oriented Machine for Control Applications

Giuliano Donzellini, Stefano Nervi, Domenico Ponta,
Sergio Rossi, and Stefano Rovetta

Department of Biophysical and Electronic Engineering, University of Genova
Via all'Opera Pia 11a – 16145 Genova (Italy)

donzie@dibe.unige.it, nervi@dibe.unige.it, ponta@dibe.unige.it,
rossi@dibe.unige.it, rovetta@dibe.unige.it

Abstract – *Microprocessor design and manufacturing have experienced great improvements in the last years. However object-oriented concepts, in spite of their widespread diffusion as a programming principle, have not been given great attention in hardware design. This paper presents an object-oriented machine, currently under development, which incorporates (at the machine-code level) some mechanisms needed for manipulating objects and methods. The processor, oriented to control applications, is composed of a commercial, full-32-bit RISC processor acting as the computing core, and additional circuitry. The additional elements constitute a shell, providing dedicated registers and functions for dealing with class instances and related methods. A mechanism for tracking called methods, by hardware support of the Virtual Method Table, is provided in parallel to the normal calling operation of the processor. The overhead associated with this mechanism, normally taken in charge by the core processor, is therefore left to the additional circuitry.*

INTRODUCTION

The fast improvement in microprocessor architecture design and manufacturing has caused object–oriented programming to reach a widespread and generalized applicability. In the last few years, it has become a major area of interest mainly as a result of the consequent increase in power of very low–cost machines. However, from a methodologic point of view, we should also account the large number of applicative areas that can benefit of the added value of an object–oriented design, namely, the strong capabilities of information–hiding and generic typing. These features are all aimed at a strong modularity of the applications, so that higher–level programming is easier and the user can concentrate more on the specific application than on details of the programming environment. Re–engineering of existing applications is also greatly facilitated by an object–oriented approach. As a result, most of the recently designed programming and scripting languages (*e.g.*, JAVA [1], Object Pascal [2]) and many operating systems (*e.g.*, OS/2 [3]) incorporate concepts from the object–oriented model. Industrial control applications are an interesting area for the development of object–based solutions. In this context, engineers that are skilled in their specific fields may take advantage of object–oriented development environments and of the availability of dedicated hardware.

The strong development in hardware performances has thus caused a progress in high–level software tools. However, to date this has not caused a corresponding increase in specific hardware support to these tools. This paper presents preliminary results on the design of an object–oriented processor, namely, a RISC architecture [4,5,6,7] augmented with additional circuitry that implements (directly in hardware) some low–level processing and control steps required by the object–oriented model. The specific design presented here refers to a protoype implementation that is currently under development (as a part of Esprit Project 7517 SUMIS). Due to time and availability of resources, this first step is built around a commercial RISC microprocessor (ARM7, by Advanced RISC Machines Ltd) [8] with the additional circuitry interfaced as a coprocessor. This implementation is oriented to control applications, and aims mainly at providing control engineers with flexible and powerful programmable devices, that nevertheless retain a simple high–level programming interface.

A CLOSER LOOK AT OBJECT - ORIENTED TECHNIQUES

An Object Oriented Machine (OOM) should support the basic concepts of Object Oriented Programming (OOP): Encapsulation, Inheritance and Polymorphism [9,10]. Object oriented languages, while featuring no theoretical advantage in this respect over traditional programming techniques, nevertheless make the application of these concepts straightforward [2,9,10]. An object oriented compiler will translate an object oriented solution into suitable, low level, software control and data structures, according to the capability of the target processor.

To examine how the basics concepts of OOP could be supported by the hardware, it is necessary to take a closer look at object oriented techniques, at a quite low level [11, 2].

Encapsulation allows combining both data and code in *classes*. For each instance of a class (an *object*), memory space is allocated to store all the data encapsulated with it. Moreover, for each class type declared in the source code, the compiler constructs a table, here referred as *Virtual Method Table* (VMT), containing function pointers and class type information. In the data space of each instance, a location is reserved to store a reference to the VMT of its own class. In general, the execution of object oriented code is centered on the use of instances and tables.

Inheritance information is registered in the VMT of the class. For instance, if a class CHILD inherits all its characteristics from a class PARENT, in the VMT of CHILD we find a pointer to the VMT of PARENT. This pointer is part of a linked list that ends with the common ancestor of all the instanced classes.

The main aim of the VMT is to support the *polymorphical* behavior of the classes. Polymorphism allows sharing the name of a function up and down the class hierarchy, with each class in the hierarchy implementing the action in a "customized" way.

A function of this type is usually called a *virtual method*. The VMT stores an indexed table of pointers to the virtual methods. The new class CHILD may redefine some functions with respect to the ancestor PARENT. The VMT's of the classes CHILD and PARENT will therefore differ accordingly. Some new pointer entries will be added, some others will be overridden.

DESIGN ISSUES

At run-time, when an instance of a class CALLER needs to call a virtual method of an instance of a class CALLED, a low level trip around tables and pointers needs to be made to satisfy the required call.

The software implementation of this sequence is quite time-consuming. First, CALLER needs to get the address of the instance of CALLED. Then, by pointing in memory with this address, it picks up the pointer to its VMT. At run-time, the source name of the function corresponds to an index into the VMT, so another access in memory will be performed this time to get the address of the function to be actually executed. Finally, the address of the instance of CALLED and the return address will be pushed onto the stack, and the function will be executed.

Because of the need to call a function of an object from everywhere, the software implementation cannot distinguish among the origins of the calls. That is, if a function is called from an instance of the class it belongs to, the call sequence will be the same, and the code will implement a trip to get some pointers that, as matter of fact, it already has.

Without a hardware support, the call sequence must be the same, because the low-level return sequence is inevitably the same.

Another issue can be noted. The executable code of an object needs to access its data. However, an instance is known to itself only because its own address has been pushed onto the stack. Again, all the needed stack accesses may be very time consuming, especially when an object requires repeatedly or recursively virtual functions of its own class.

DESIGN SOLUTIONS

The consideration above suggested us to optimize the access to the VMT, during the method-call sequence. The travel between pointers and tables can be done by a dedicated hardware, while the main processor executes other tasks. In this way we can also limit the effects of the required wait-states in external memory accesses. Note that it is not possible to allocate a VMT-reserved RAM on a chip, because object oriented programs utilize a lot of different classes: the required size of the RAM would be impractically high.

We also chose to reserve an internal register to hold the address of the current instance (*i.e.*, the instance of the class currently in execution). Obviously, this choice permits the maximum efficiency in the accesses to the current instance data fields, but creates the necessity of saving and restoring the content of the register during the

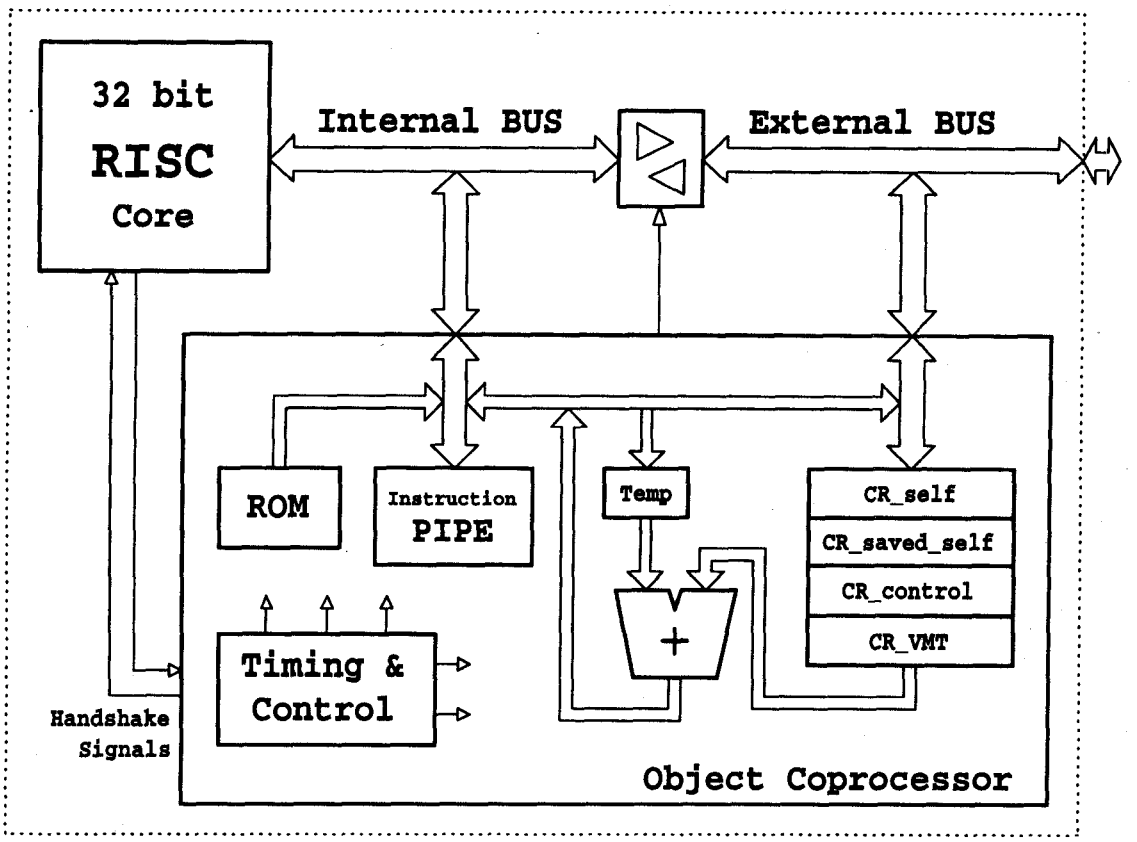

Fig.1: The structure of the processor.

context switching between objects. We save and restore this register via hardware support.

To optimize the method call sequences, we decided to distinguish via hardware a) the calls made between two different instances and b) the calls made internally by the same instance. In the case a) it is necessary to save the current instance address and get the new one; in the case b) we can optimize efforts leaving unchanged the current instance address. This distinction is possible only if the hardware is able to discriminate cases a) and b), at the time of the function return. Actually, as seen before, the same function can be called in both cases and, consequently, the return sequence needs to know wether the previous instance address must be restored or not.

An ideal solution would be to design completely from scratch the whole processor, but this would consume a lot of time and resources. We have decided instead to search, among available CPU standard cells, the one that could be suitable to be "object-extended".

Naturally, we have tried to identify a reasonable compromise among feasibility, efficiency, cost compatibily and the ideal target. For instance, using an existing "black–box" CPU, it is not possible to preview the current state of the internal CPU sequencer, modify the existing instruction execution sequences, or read the CPU flags.

Nevertheless, we found that ARM7 standard cell was suited to the purpose. ARM7 is a full 32 bit RISC processor, that can be efficiently connected to a coprocessor to extend its capabilities. Moreover, among the remarkable features of the ARM architecture, there is an efficient implementation of the indexed–by–register addressing mode, so it is reasonably suitable for the implementation of object oriented languages.

The actual structure is composed by the CPU core and the "object" coprocessor. The resulting set acts as a new processor, with the instruction set extended to "object" treatment capabilities.

SOME TECHNICAL INSIGHTS

The 32 bit RISC core is connected to the object coprocessor (OCP) via an internal bus and a few handshake lines (Fig.1). The internal bus and the external system bus feature a line by line correspondence, but they are separated by a bi–directional buffer. The OCP has access to both the internal and external busses.

The CPU core sees the OCP as a normal coprocessor, but the latter is able to take control of the execution flow when needed. This double connection allows the OCP to disconnect the core and at the same time to control both the core and the external memory systems. During the normal flow of instruction execution, when no "object instruction" (OI) is fetched, the coprocessor stays idle, while the core remains connected to the external bus. As soon as an OI is fetched, the OCP starts its activities.

The typical OI consists in two synchronized sequences: one seen by the CPU core, the other seen by the external system. The OCP begins to fill the CPU core instruction pipe with a few instruction codes, using the OCP internal ROM. In parallel, the OCP interacts with memory to read/write pointers and save/restore registers, according to the needs of the current OI. When needed, the external and internal activities are linked together: for instance, in some situation, the CPU core picks up an address from the OCP rather than from memory, as normally expected according to the instruction format.

Because of the internal ROM, we need no wait state in fetching the OCP-given instructions, so we obtain a gain in speed, as we compare each sequence with the normal case in which fetches from external memory are required.

In the OCP structure some registers are used to store the current and previous instance addresses (CR_self, CR_saved_self), and to control the overall operation (CR_control). A temporary register (CR_vmt) stores the VMT address during the virtual method call operations. An arithmetic circuit is used to calculate the indexed addresses.

CONCLUSIONS

The new OI's let the resulting processor to speed some basic operation, typical of object oriented languages. Even if the object coprocessor is flanked to and not really integrated with the CPU core, preliminary results show sensible gain in speed and generality of formulation of the associated object model.

REFERENCES

[1] M. Campione and K. Walrath, *The Java Tutorial,* Addison-Wesley, to be published. Draft available on–line (http://www.javasoft.com).

[2] *Borland Delphi for Windows: Object Pascal Language Guide,* Borland International, Inc., 1995.

[3] G.Letwin, *Inside OS/2*, Microsoft Press, 1988.

[4] D.A.Patterson, "Reduced Instruction Set Computers", *Commun. ACM*, 28:1, Jan 1985, pp.821.

[5] C.E.Gimarc and V.M.Milutinovic, "A Survey of RISC Processors and Coputers of the Mid-1980s", *Computer*, Sept 1987, pp. 59-69.

[6] W.Stalling, "Reduced Instruction Set Computer Architecture", *Proc. of the IEEE*, Jan 1988, pp.38-55.

[7] D.A.Patterson and J.L.Hennessy, *Computer architecture: A quantitative approach*, Morgan Kaufmann, 1990.

[8] *ARM7DMI Data Sheet*, Advanced RISC Machines Ltd (ARM) 1994.

[9] B.Stroustrup, *The C++ programming language (2nd edition),* Addison-Wesley, 1991.

[10] *Turbo Pascal for Windows User's Guide*, Borland International, Inc., 1991.

[11] *Open Architecture Handbook*, Borland International, Inc., 1991.