Object Oriented ARM7 Coprocessor

Giuliano Donzellini, Stefano Nervi, Domenico Ponta, Sergio Rossi, and Stefano Rovetta
Department of Biophysical and Electronic Engineering, University of Genoa, Italy
E-mail: {donzie, nervi, ponta, rossi, rovetta}@dibe.unige.it

Abstract

This paper presents preliminary results on the design of an Object Coprocessor (OCP) cooperating with a RISC–architecture processor (ARM7, by Advanced RISC Machines Ltd.). This coprocessor implements in hardware some low–level processing and control steps required by the object–oriented model. The processor and the OCP constitute a processing architecture whose extended instruction set features "object" treatment capabilities. Some special coprocessor instruction codes ("Object Instructions") have been introduced. Concepts such as "Polymorphism" and "Virtual methods" are supported at the hardware level. Preliminary results using typical object–oriented sequences show a gain in speed over a pure software implementation, on the same RISC machine. The specific design presented here refers to a prototype implementation that is currently under development and supported by the European Union Esprit Project (7517 SUMIS).

1. Introduction

Object–oriented programming is becoming a major tool for the development of applications ranging from small to large projects in many fields and under a vast majority of platforms. This is mainly due to the notable increase in power of very low–cost machines. As an example, very small consumer appliances such as phones and entertainment consoles have recently been addressed as a potential target for the Windows CE operating system. From a methodological point of view, we should also account the large number of application areas that can benefit of the added value of an object–oriented design. 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. Control applications and embedded signal processing functions are an interesting area for the development of object–based solutions (some examples include industrial process automation, power management and control, field bus distributed control, and biomedical instrumentation). In these contexts, engineers that are skilled in their specific fields, but not necessarily in software development, may take advantage of object–oriented development environments and of the availability of dedicated hardware.

High–level software tools have experienced a notable development as a consequence of the strong advances in performances of conventional-architecture processors, even in the lowest cost range. However, to date a corresponding increase in specific hardware support to these tools is still lacking. This paper presents preliminary results on the design of an object–oriented processor [4], namely, a RISC architecture [5,6,7,8] augmented with additional circuitry that implements (directly in hardware) some low–level processing and control steps required by the object–oriented model. The design presented here is a subset of a microcontroller chip targeted to automatic control of industrial systems, particularly in the field of power generation, conversion and management. The aim is to provide control engineers with flexible and powerful programmable devices, that nevertheless retain a simple high–level programming interface.

The microcontroller is currently under development as a part of the Esprit Project (7517 SUMIS) funded by the European Union (http://www.cordis.lu/esprit/home.html). It will provide signal pre–processing blocks (root-mean-square value, signal phase measurement, frequency and time counters, multichannel A/D conversion) and various kinds of signal output blocks. Due to time and availability of resources, our implementation is built around a commercial RISC microprocessor (ARM7, by Advanced RISC Machines Ltd.) [9] with the additional circuitry interfaced as a coprocessor.

1060-3425/98 $10.00 (C) 1998 IEEE

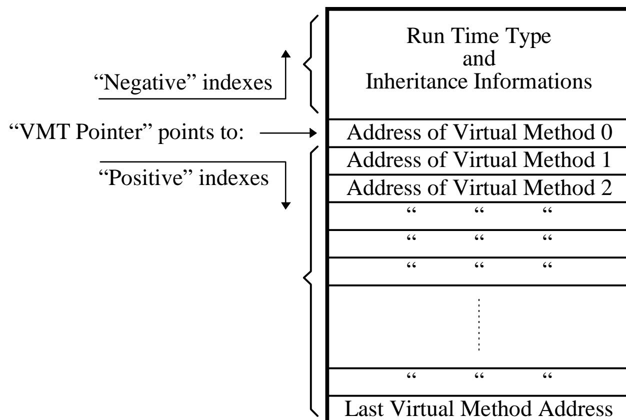

Fig. 1. The structure of a Virtual Method Table.

Notes:
- Virtual Methods Addresses are stored in the "positive" side of the table
- In the "negative" part of the table we found various data, for example: inheritance and run time type information, instance size and fields type information.

2. Object–oriented concepts and techniques

Encapsulation, Inheritance and Polymorphism [10,11] are the basic concepts of Object Oriented Programming (OOP). 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.

An Object Oriented Machine (OOM) should provide additional machine–level instructions and hardware support for typical OOP operations. 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 [12, 2].

Encapsulation allows combining both data and code in classes. For each instance of a class, memory space is allocated to store all the data encapsulated with it. Moreover, for each class type declared in the source code, the compiler builds a table, here referred as Virtual Method Table (VMT), containing function pointers and class type information (see Fig. 1). 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 based 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 a pointer to the VMT of PARENT is present. 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.

3. Efficiency of software–only implementation

Method calls and returns require a larger number of operations than normal calls and returns from subroutines. 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.

As shown in Fig. 2, CALLER needs first to obtain the address of the instance of CALLED. This address is normally known as "Self" [2, 11] or "This" [10]. 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 return address will be saved and the function executed.

The necessity of calling a function of an object from everywhere does not allow the software implementation to distinguish among the origins of the calls. 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.

Fig. 2 shows that the pointer to the current instance Self is saved before being overwritten by the one of CALLED. At the time of return, the Self of CALLER must be restored. The Self address is important because an instance is known to itself only because its own address is accessible by means of its own register. Since the software implementation is not aware of the call type, as outlined above, it is necessary to access the stack very frequently for saving and restoring the Self address.

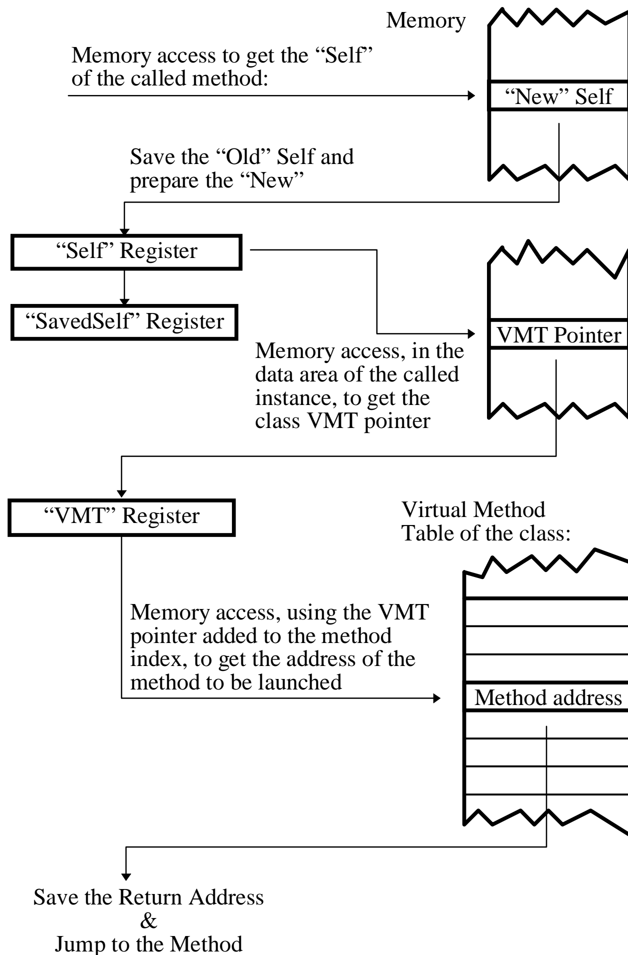

Fig. 2. Sequence of low–level operations necessary to call a Virtual Method.

4. Hardware support issues and design solutions

We decided to optimize the access to the VMT, during the method–call sequence, due to the considerations above explained. The travel between pointers and tables can be performed by a dedicated hardware, while the main processor executes other related tasks. In this way we can also limit the effects of the required wait–states in external memory accesses. Allocating a VMT–reserved RAM on a chip would not be a feasible solution: the required size of the RAM would be so large to be impractical, since object oriented programs are designed for using a lot of different classes.

We optimized the method call sequences by distinguishing, via hardware, the calls made between two different instances (a) and the calls made internally by the same instance (b). In case a) it is necessary to save the current instance address and get the new one; in case b) we can optimize efforts leaving unchanged the current instance address. The hardware is able to discriminate cases a) and b), at the time of the function return.

Designing a completely new object–oriented processor was not feasible, due to time and resource constraints. We have decided instead to select a CPU standard cell suitable to be "object–extended". However, using an existing "black–box" CPU, one has to make some compromises. For instance, 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.

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 interesting 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.

5. Object Coprocessor architecture and basic operations

The object–oriented processor is composed of the CPU core and the "Object Coprocessor" (OCP). Its instruction set is therefore extended to "object" treatment capabilities.

An important feature of the coprocessor is its non–standard connection to the 32 bit RISC core. Usually, a coprocessor is attached to the unique system bus that connects core and memory. Under the control of the main processor, a normal coprocessor is enabled to work when a coprocessor instruction is present in the execute stage of the pipe. The core normally waits for the coprocessor to finish its duty and then proceeds with next instruction; it never stops fetching and decoding instructions from memory, and executing or delegating them to the coprocessor.

In our system, as shown in Fig. 3, the bus connecting core with memory and I/O devices is split into an internal and an external section, separated by a bidirectional buffer. The internal bus and the external system bus feature a line by line correspondence. The coprocessor is placed in a privileged position, because has access to both busses.

It is worth noting that core and coprocessor look at different scenarios. The CPU core sees the OCP as a normal coprocessor, but the latter controls the connection between internal and external busses. The OCP is able to take control of the execution flow when needed, by disconnecting the bus and then controlling at the same time both the core and the external memory system. During the normal flow of instruction execution, when no coprocessor instruction is fetched, the coprocessor stays idle, while the core remains connected to the external bus and controls the whole system.

As soon as a coprocessor instruction is fetched, the OCP starts its activities. A few instructions, like data

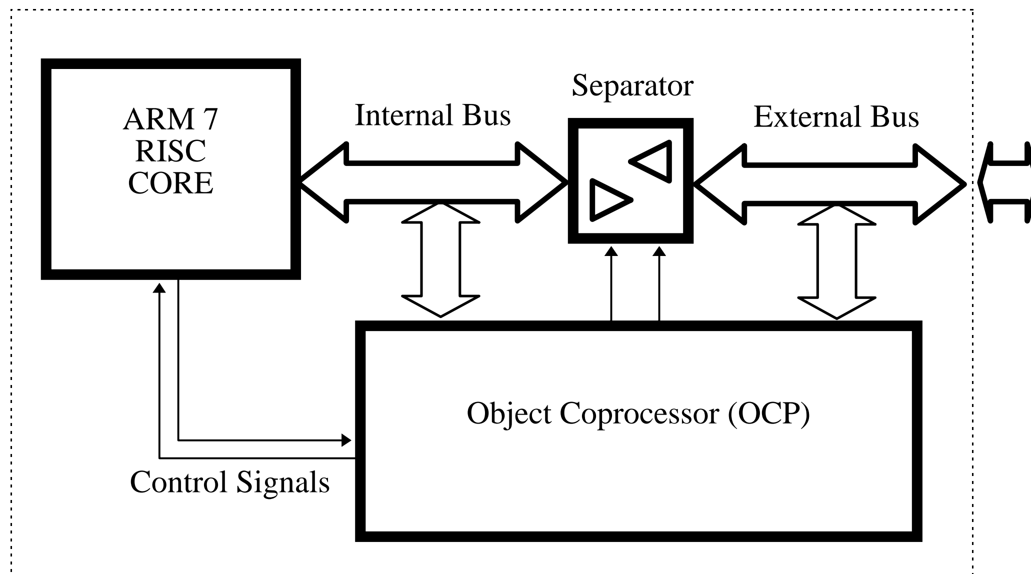

Fig. 3. Block diagram of the connection between the ARM7 core and the Object Coprocessor

Notes:
- Separator: a bi-directional separator / buffer that enable connection / disjunction of the ARM7 core to / from the external system.
- Internal Bus: local bus connecting the ARM7 core to the OCP and, trough the separator / buffer, to the external system.
- External Bus: system bus connecting the whole system to the OCP and, trough the separator / buffer, to the ARM7 core.

transfer between registers and between registers and memory, are executed by the OCP without modifying the standard bus connection. It is interesting, instead, to examine what happens when the coprocessor finds “Object Instructions” (OI), so called because they are designed to support the operations characterizing OO languages.

As shown in Fig. 4, OCP contains, as any other coprocessor of a RISC CPU, an instruction pipe. At any given time, core and OCP pipes have the same content: when the core fetches an instruction, OCP does the same in parallel.

When OCP finds an OI in the decode stage, it gets ready to take over, by preparing the execution of the corresponding sequence. At the execute stage, upon permission from the core, OCP begins its operations, first of all by splitting the bus.

The next phase of the execution process consists in two synchronized sequences: one on the CPU side, the other on the external system side, both under the control of the OCP sequencer.

The Instruction Sequence Generator (ISG) is an architectural component that supports the execution sequence on the CPU side. Controlled by the OCP sequencer, ISG acts in place of memory, generating instruction streams that are fetched, trough the internal bus, by the CPU, for the duration of the OI execution.

ISG builds the opcodes at run–time, processing the information from the operating code fields of OI with the help of an internal table. While the CPU executes the instructions provided by the OCP, the latter 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 the address just calculated by the OCP or, in other cases, OCP grabs a register content from the CPU to use as a memory address.

An Arithmetic Logic Unit (ALU) processes the addresses obtained from the transfers mentioned above, for example to index them.

Two registers are also present, used to store the addresses of current and previous instances (CRSelf and CRSavedSelf registers). Note that a CPU register, called RSelf in our context, is reserved to duplicate the content of CRSelf, to give the CPU immediate availability of the current instance. CRSavedSelf register can be seen as a normal register (used to save the CRSelf content) in its lower part and as a increment/decrement counter in its upper part. As it will be shown in the following, this latter function is used during the call/return sequences to/from methods belonging to the same class instance.

The CRVmt register stores the VMT address during the virtual method call operations and two registers control the overall operation (CRControlA and CRControlB register). The size of all register is 32 bits.

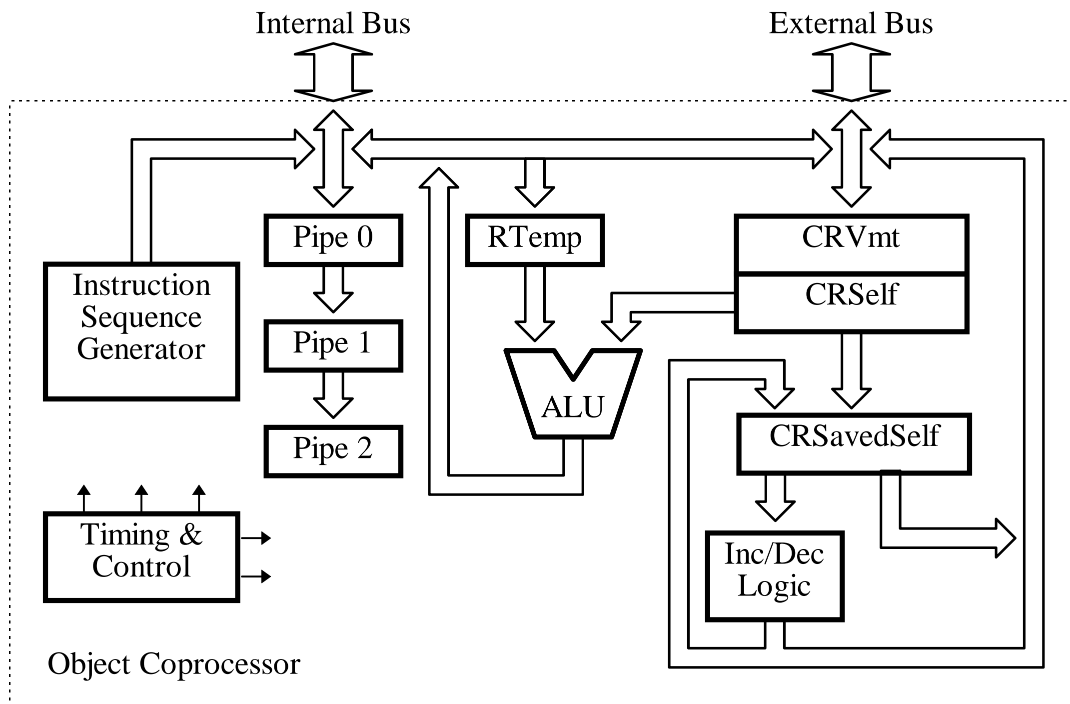

Fig. 4. Block diagram of the Object Coprocessor architecture

Notes:
- Pipe 0,1,2: the instruction pipe registers, 'paralleled' with the ones present in the ARM 7 core.
- Timing & control: the sequencer for the whole object coprocessor.
- ALU: a Logical Arithmetic Unit, used to manipulate method addresses.
- RTemp: a temporary register, internally used by the ALU.
- CRVmt: register used to store the VMT address during call sequences of virtual methods.
- CRSelf: register used to store the Self address of the current instance.
- CRSavedSelf: saved Self register, used to save the address of the previous instance and to maintain the count of the number of method calls within the same instance.
- Inc/Dec Logic: increment / decrement logic used to execute the counting in CRSavedSelf

6. The Object Instructions

OI's support a few of the typical operations of OO languages, in particular the management of methods, taking care of calls and returns in static and virtual modes. Six types of method calls and a unified return (see Table 1) are implemented.

Method calls and returns of OI's are basically similar to the plain calls and returns of normal sub–programs, in that OI's take care of the usual save and restore operations of the return address. However, these are done with the addition of automatic save and restore of the current instance pointer (Self). Moreover, in the case of virtual methods, OI's perform all the trips necessary to find the address (polymorphical late binding) of the method called.

The OI's, obviously, do not support directly stack frame construction and parameters passing.

The same method may be called in a static mode (early binding), with the METSM, METSR and METSI instructions, or in a virtual mode, with the METVM, METVR and METVI instructions. The choice is made by the compiler, on the basis of the definition provided by the programmer in the source code and in accord with the semantic context. The compiler is also in charge of deciding where to get the Self pointer of the method called (it normally can be in memory, but also it could have been already loaded in the processor, for example).

A method can be called: a) from a method belonging to an instance of any class, different from its own class instance, b) from a method belonging to the same class instance (also recursively) and c) from a code not belonging to any class.

In the case a), because the new pointer Self is loaded with the address of the instance of the class of the method called, the need arises of saving the caller's Self pointer. (instructions METVM, METVR, METSM and METSR).

In the case b), because the instance "called" is the same as the one "calling", it is not necessary to get the new Self and, consequently, it is not even necessary to save the current one (instructions METVI and METSI).

The distinction between cases a) and b) is made to optimize the operation sequence in case b).

1060-3425/98 $10.00 (C) 1998 IEEE

Table 1. The Object Instructions

METVM	Virtual Method Call (Memory)	Virtual method call from a class instance different from the called one. The pointer to the instance of the called (new "Self") is taken from memory. The virtual method is identified by its position (index) in the Virtual Method Table (VMT).
METVR	Virtual Method Call (Register)	As above, with the difference that the pointer to the instance of the called (new "Self") is taken from an ARM register.
METVI	Virtual Method Call (Internal)	Virtual Method call from the same class instance of the called one. The pointer "Self" is already available, inside the core (register RSelf), and the coprocessor (register CRSelf). As above, the virtual method is identified by its position (index) in the Virtual Method Table (VMT).
METSM	Static Method Call (Memory)	Static Method call from a class instance different from the called one. The pointer (new "Self") is taken from memory. The method is identified by its address, specified in relative mode by respect to the Program Counter.
METSR	Static Method Call (Register)	As above, with the difference that the pointer (new "Self") is taken from an ARM register. The method is identified by its address, specified in relative mode by respect to the Program Counter.
METSI	Static Method Call (Internal)	Static Method call from the same class instance of the called one. The pointer "Self" is already available, inside the core (register RSelf), and the coprocessor (register CRSelf). The method is identified by its address, specified in relative mode by respect to the Program Counter.
RETM	Return from Method	Unified return from method. The instruction recognizes automatically the mode the method has been called and decides whether it must recover the caller's "Self".

In case c) the current value of the Self pointer is meaningless, but it is anyway necessary to get the Self pointer to the instance of the class of the method called. For the sake of simplicity and homogeneity, the same instructions as in case a) are used. Obviously, in this last case, the Self pointer saved is meaningless.

In the case of the instructions METVM, METVR, METSM and METSR, because the method called belongs to an instance different from the one of the method calling, the new Self is loaded in the registers RSelf of the processor and CRSelf of the coprocessor. The content of CRSelf, before being substituted by the new value, is copied into CRSavedSelf.

In the case of the call instructions METVI and METSI, because the called belongs to the same class of the caller, the registers RSelf and CRSelf are not modified. Instead, the upper part of CRSavedSelf, cleared in the other cases, will be used as counter of the number of nested calls and/or consecutive inside the same class instance.

This counting mechanism is made more complicated by the necessity not to limit the number of nested and consecutive calls possible from within the same class. The limit imposed by the counter's field is overcome by saving the register in the stack every time the counter overflows, i.e. every 256 nested and consecutive calls within the same instance. For sake of simplicity, the details of this stack saving are not reported here.

Obviously, all call instructions save their return addresses, as it is done in the ARM core, copying the Program Counter (R15), after correcting it, in the register R14. Such behavior requires that, if the method called has internally other calls (in other words, it is not a leaf method), in its entry/exit code the content of R14 and CRSavedSelf will be saved/recovered from stack.

It is important to notice that all methods end with the same unified return OI, RETM (see Table 1). RETM recognizes automatically the modality that has been used to call the method, and therefore knows whether it must recover the caller's Self.

The decision is made by controlling the counter in the upper part of register CRSavedSelf. If the count is not zero, the call has originated within the same class instance. RETM in this case must decrement the count of the calls, without modifying the registers CRSelf of the coprocessor and RSelf of the core.

If the count is zero, the call came from code belonging to a different instance, so RETM copies the content of the register CRSavedSelf in the register CRSelf of the coprocessor and RSelf of ARM, restoring in this way the pointer Self to the caller's instance. Actually the control mechanism is more complicated, as outlined above, due to the need to control also the counter's overflow trace on the stack, every 256 consecutive returns within the same instance.

7. Efficiency considerations

In this paragraph a few cases of calls and returns from a method, implemented only by software, will be compared with the equivalent functions supported by the coprocessor. Beside, a few other considerations on the factors affecting speed performances will be made.

As seen previously, a pure software implementation cannot distinguish between a call made from inside the same class instance and a call in between different instances. In fact, every method can be called from any other method of any instance, both in static and virtual mode. As a consequence, the method code, being unique, must end with a sequence of instructions compatible with all call types. The return sequence must therefore include always the recovery of the caller Self, and the calls will be of the type between different instances.

Fig. 5 presents an example of "software only" implementation of a virtual method call and return. In this approach, to enable comparison, we suppose to use two additional ARM registers, in addition to the 'Rself' register used to storage the current instance pointer. We named them 'RSavedSelf' (used to save RSelf) and 'Rvmt' (used for the VMT pointer). The call sequence proceeds as previously described (see Fig. 2). In the return sequence, the ADD instruction has been included to enable a comparison of this sequence with the coprocessor–assisted one.

In Fig. 6, the virtual method call and the unified return, for the case of caller and method belonging to different class instances, are implemented with the coprocessor–assisted OI. The OI's employed are METVM call and RETM return. This sequence, substantially, operates as in the software–only implementation, but the code is more compact, and runs faster.

Note that, due to pipe synchronization, and to the necessity to respect some external timing requirements of the ARM 7, an ancillary instruction has to be placed after each OI.

In spite of the fetching order, this ancillary instruction is actually executed not after, but during the execution of the OI. The OCP is able to recognize the presence of illegal ancillary instructions, and to refuse execution of the OI, forcing the ARM 7 core to execute an "Undefined Instruction" internal exception, to enable run–time debugging of OI's.

For the METVM, METVR, METSM and METSR instructions, the use of the ancillary instruction is necessary to transfer the new Self of the method called in

```
;--------- Virtual Method Call instruction sequence -------------------
;
       MOV     RSavedSelf, RSelf                   ;Save RSelf in RSavedSelf
       LDR     RSelf, [<Register> + Offset]        ;Load RSelf from memory
       LDR     Rvmt, [RSelf]                       ;Load VMT pointer
       MOV     R14, PC                             ;Save the return address
       LDR     PC, [Rvmt + <Index of Method>]      ;Jump to the virtual method
       .
       .
;--------- Method Code
;
METHOD <entry code>                                ;Method entry code
       SUB     SP,SP, #<localsize>                 ;make room for local variables
       .
       .
;--------- Method Return instruction sequence ------------------------
;
       MOV     RSelf, RSavedSelf                   ;Restore RSelf from RSavedSelf
       ADD     SP,SP, #<localsize>                 ;Local variables removal
       MOV     PC, R14                             ;Copy the return address to PC
                                                   ;(to return to the caller)
```

Fig. 5. Software only implementation of Virtual Method Call and Method Return

Notes:
- We suppose to use two ARM registers, in addition to the register 'RSelf' used to store the current instance pointer. We give them the names of 'RSavedSelf' (used to save RSelf) and 'Rvmt' (used for the VMT pointer).
- PC is the Program Counter (R15 register of ARM).
- SP is the Stack Pointer (usually the R13 register of ARM).
- R14 is an ARM internal register, used to save the return address.
- <Register> is a register of ARM, containing an address reference to the pointer to the instance of the class of the method that we are calling.
- <Index of Method> identifies the virtual method to be called.
- For comparison purpose only, the method return code includes here an ADD instruction to remove the local variables, because in the case of usage of the RETM instruction, we are forced to introduce an 'ancillary' instruction after it. Obviously, this instruction could be substituted with any other useful. The SUB instruction is introduced in the method code only to pair the ADD instruction.
- <localsize> is the overall size of the local variables.

the RSelf and CRSelf registers. METVI, METSI and RETM don't need a specific service by the ancillary instruction, but the latter is executed anyway. The programmer can use the ancillary instruction to perform, for instance, a register to register transfer or operation.

Fig. 7 provides an example of METVI virtual method call instruction. METVI is used when caller and called belong to the same instance and the Self does not need to be saved or transferred. Note that the return instruction sequence is the same of the previous example.

Comparing the execution times of the sequences taken here as examples implies taking into account the wait cycles that each implementation adds to the timing of memory accesses and, in general, of external components. In our implementation, the memory access controller adds by default a wait cycle to the basic ARM timing, and it is possible to set additional wait cycles.

Table 2 and 3 present a comparison among the three examples, as a function of the wait cycles imposed by the system. Table 2 refers to the efficiency of virtual method calls, Table 3 of returns.

As anticipated, the improvement is larger in the case of calls from within the same instance: in the case of the instruction METVI, for instance, the average gain in terms of execution time over a pure software implementation is approximately 50%. For the instruction METVM (call from different instances) the gain is instead around 40%.

Calls take better advantage from the hardware support than returns. Returns, in fact, must be served by hardware because it is necessary to distinguish if the call is coming from the same instance or not. As a consequence, we have no substantial advantages by relieving the core from some of the operations.

As Table 2 and 3 show, in the case of one wait state, RETM provides a gain of 30% in the case of a return from a method within the same instance, while in the case of different instances there is a gain only when a larger number of wait states is present.

It can be noted, in fact, that all OI's, but RETM in particular, show a reduced dependence of execution times from wait states, in comparison with a pure software approach, because the coprocessor allows to avoid many external memory accesses.

It may be useful, at this point, to add a few considerations on the low–level interaction between ARM7 and the object coprocessor.

As described earlier in the paper, the coprocessor remains idle when none of its dedicated instructions is fetched: therefore no overhead is associated to the presence of the coprocessing hardware. The interaction between ARM7 and OCP when OI's are executed has been optimized by a careful temporal interlacing of the operation that the two units perform in cooperation. Of course, here and there, constraints inherent with the

```
;--------- Virtual Method Call instruction sequence -------------------
;
       METVM <Index of Method>                       ;Virtual Method Call
       LDR    RSelf, [<Register> + Offset]           ;Load RSelf from memory
                                                     ;(jump now to the Method)
       .
       .
;--------- Method Code
;
METHOD <entry code>                                  ;Method entry code
       SUB    SP,SP, #<localsize>                    ;make room for local variables
       .
       .
;--------- Method Return instruction sequence -------------------------
;
       RETM                                          ;Method Return
       ADD    SP,SP, #<localsize>                    ;Local variables removal
                                                     ;(return now to the caller)
```

Fig. 6. Coprocessor-assisted Virtual Method Call and Method Return, when the class instance of the caller and the method are different (METVM instruction).

Notes:
- <Index of Method> identifies the virtual method to be called.
- <Register> is a register of ARM, containing an address reference to the pointer to the instance of the class of the method that we are calling.
- The LDR instruction is the "ancillary instruction" of METVM and it is executed before jumping to the method. It is in charge of get the Self of the called method.
- SP is the Stack Pointer (usually the R13 register of ARM).
- <localsize> is the overall size of the local variables.
- The ADD instruction is the "ancillary instruction" of RETM and it is executed before returning to the caller. Here it is used to remove the local variables from the Stack. This instruction could be substituted with any other useful. We use here the same instruction that we applied in the software only implementation, for comparison purpose. The SUB instruction is introduced in the method code only to pair the ADD instruction.

1060-3425/98 $10.00 (C) 1998 IEEE

characteristics of ARM7 macrocell are present. For instance, the lack of an interrupt acknowledge signal from ARM7 has obliged to add a clock cycle to the OI's sequences.

OCP control registers are involved in the system initialization phase only, normally at reset, and therefore their load operations do not generate overhead.

In addition to the OI's previously described, a few more coprocessor instructions have been added to the set to allow communication between OCP and ARM7 registers, and between OCP registers and memory, when needed. These "service" instructions are used during the context–switching between processes and the exception servicing (when the contents of CRSelf and CRSavedSelf registers need to be saved/restored), and in the entry and exit code of "not leaf" methods (when the CRSavedSelf register has to be saved/restored, because the method code contains at least one call to a method of another class instance). A properly designed compiler can minimize the usage of these instructions. The couple of push and pop instructions, when inserted by the compiler, generates an overhead of eight clock cycles (if one wait state is used).

The analysis just carried out shows that OI's accelerate in particular the operations of call and return from virtual methods within the same class instance. The advantages in terms of execution times, therefore, will be more relevant in the applications where the calls within the same class instance are more frequent than calls between different class instances and where polymorphism and virtuality are widely applied. Examples may be taken from classes of polymorphic collections, queues, variable sized arrays, linked lists and trees, commonly used in OOP. Another example is represented by recursive algorithms that take advantage of the hardware support, when the function calling itself is a method (and therefore we are dealing with calls within the same class instance). The advantages will be, instead, much less relevant in the applications where the computational aspects and the use of standard (non methods) function dominate.

A real verification of speed performances with different kinds of applications could be done later, when an high level language compiler will be available. The implementation of the compiler had been planned to proceed in parallel with the development of the chip but, after an unforeseen reduction of funding, it has been chosen to delay its realization, in favor of the design of the remaining parts of the chip.

In the current phase of the project, coprocessor testing has been done using the ARM7 macro assembler, integrated with the new coprocessor instructions (written as macros). In our procedure, the memory image of the machine code program produced by the assembler is therefore converted in a VHDL file describing a ROM. Then all the hardware and software verifications are done through the VHDL simulator.

```
;--------- Virtual Method Call instruction sequence -------------------
;
       METVI <Index of Method>                         ;Virtual Method Call
       MOV   R0,R0                                     ;"ancillary" instruction
                                                       ;(jump now to the Method)
        .
        .

;--------- Method Code
;
METHOD <entry code>                                    ;Method entry code
       SUB     SP,SP, #<localsize>                     ;make room for local variables
        .
        .
;--------- Method Return instruction sequence ------------------------
;
       RETM                                            ;Method Return
       ADD     SP,SP, #<localsize>                     ;Local variables removal
                                                       ;(return now to the caller)
```

Fig. 7. Coprocessor-assisted Virtual Method Call and Method Return, when the class instance of the caller and the method are the same (METVI instruction).

Notes:
- <Index of Method> identifies the virtual method to be called.
- MOV R0,R0 here is a NOP instruction. It is the "ancillary instruction" of METVI and it is executed before jumping to the method. The programmer can substitute it with another, useful, instruction.
- SP is the Stack Pointer (usually the R13 register of ARM).
- <localsize> is the overall size of the local variables.
- The ADD instruction is the "ancillary instruction" of RETM and it is executed before returning to the caller. Here it is used to remove the local variables from the Stack. This instruction could be substituted with any other useful. We use here the same instruction that we applied in the software only implementation, for comparison purpose. The SUB instruction is introduced in the method code only to pair the ADD instruction.

1060-3425/98 $10.00 (C) 1998 IEEE

Table 2. Virtual Method Call: some comparisons among software only implementation and METVM and METVI instructions.

Wait States	Software Only Implementation	METVM (different instances)	METVI (same instance)
1	23T	15T	12T
2	33T	20T	16T
3	43T	25T	20T

Notes: - "T" is the clock period.
- The METVM and METVI execution time includes also the "ancillary" instruction.

Table 3. Method Return: some comparisons among software only implementation and RETM instruction (in two different cases).

Wait States	Software Only Implementation	RETM (different instances)	RETM (same instance)
1	10T	10T	7T
2	15T	12T	9T
3	20T	14T	11T

Notes: - "T" is the clock period.
- The RETM execution time includes also the "ancillary" instruction. The same instruction behaves differently, distinguishing if the instance of the method and the caller are the same, or not.

8. Conclusions

We have presented in this paper the design of an ARM7 coprocessor targeted to support via hardware the execution of programs written in object–oriented languages.

Our work represents a first attempt to incorporate the object–oriented language support in the design of a microcontroller for industrial applications. The optimal solution would be, of course, to redesign an entirely new processor, integrating the object–oriented support. However, given the design goals and constraints, a reasonable compromise between performances and development efforts has been found with a hybrid architecture where a commercial macrocell (ARM7 32 bits RISC Processor) has been connected in a non–conventional way with a coprocessing unit designed for the purpose. The combined architecture behaves as a new processor with an instructions set extended to object treatment.

The results obtained confirm and quantify the improvement in term of speed that this structure allows when executing the low–level processing sequences and control steps typical of the object–oriented model.

At the time of writing, a large portion of the microcontroller chip peripheral devices are ready. Of course, the coprocessor is completed, and low level simulations have been possible thanks to the availability of the VHDL model of ARM7.

9. 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] G. Donzellini, S. Nervi, D. Ponta, S. Rossi, and S. Rovetta, "An Object–Oriented Machine for Control Applications", *Proceedings of 3rd IEEE Int. Conf. on Electronics, Circuits and Systems, ICECS'96*, Rodos, Greece, October 13-16, 1996.

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

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

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

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

[9] *ARM7DMI Data Sheet*, Advanced RISC Machines Ltd, 1994.

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

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

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

1060-3425/98 $10.00 (C) 1998 IEEE