A Parallel Architecture Simulator for the λ-matrix Data Flow language

Guilhem de WAILLY Fernand BOÉRI, Senior Member IEEE
Thème Architectures Logicielles et Materielles
Laboratoire d'Informatique, Signaux et Systèmes
URA 1376 du CNRS et de l'Université de Nice - Sophia Antipolis
41, bd Napolélon III - 06041 - Nice CEDEX - France
{gdw—boeri}@unice.fr

July 2, 1996

Abstract

In this paper, a software simulator of a parallel multiprocessor architecture is described. The intent of this work is to show how a dynamic object-oriented language can be used to model complex and realistic "things" in a very short time. The object paradigm allows an homogeneous view of objects with common methods. An inheritance mechanism significantly reduces the code because of the feature sharing. Here, the classical database object-model shows the relationships between object instances. In addition, the inheritance-graph describe the inner construct of objects. A graphical interface is built over the simulator, but only its layout is shown here.

1 Motivations

This paper presents a software simulator of a simple parallel architecture modeled with a functional language and its dynamic object-oriented interface.

A functional data-flow language was defined in our laboratory[1] in order to implement signal processing applications on fast parallel architectures. This abstract language is named λ-matrix. It offers several advantages: It is very simple and uses an accurate mathematical semantics which allows proofs of programs and behaviors. Time and memory determinisms are established in a proved way too. The data flow paradigm preserves the inner parallelism of applications and graphical representations are possible. Parallelism of applications is well exploited, due to the particular solving method. In addition, the parallel compilation algorithm uses an efficient *task interleave* and a *local memory cache* for each controllers.

In order to evaluate the λ-matrix concept relatively to the parallelism exploitation, a simulator of the architecture was needed. Moreover, this simulator is the software model of a hardware board that will be build in our laboratory.

The board is wanted to be cheap and simple. So, vulgar controller are used, directly connected to a single bus, as a common memory. I/Os are accomplished by address decoding. Common memory access conflicts are solved at compile-time. In addition, statistical results may be obtained from the simulator for analysis.

[1]This abstract language is a part of a thesis. Significant results are currently submitted to some conferences.

In order to simulate the parallel board, the programming environment STK was chosen. It offers a graphical toolkit interface (TK), a dynamic object-oriented interface (STKLOS) and a functional language (SCHEME). With the use of this powerful tool, the obtained coding is very short. It can easily be modified and adapted. The graphical toolkit is used to build an user friendly interface of the simulator. The object-oriented layer allows homogeneous view of objects through common methods. Moreover, a great code reuse is accomplished with the inheritance mechanism, even for a such small application. At last, the SCHEME language increases the programming efficiency.

In this paper, the parallel architecture is presented as specifications (§ 2). Then analysis of both the simulator and its graphical interface is given (§ 3). STK is described in a short way (§ 4). Then the STK code of the simulator can be seen (§ 5), and the layout of the graphical interface is shown (§ 6).

2 Specifications: The parallel architecture

The parallel architecture is a very simple cheap board built with vulgar controllers, as shown in figure 1. Controllers are directly connected to the bus: Sharing of bus accesses is allocated at compile-time. The board could be divided into three main parts: Controllers, a common memory which includes the global RAM and the I/Os lines, and the bus.

Each controller has its own local RAM, ROM and ALU. Program of each controllers is put on its ROM. Local RAM can be used as a local stack. In addition, the λ-matrices parallel compilation allows this local RAM to be used as an efficient global memory cache. Each controller has two general registers (**A**, **B**), one code pointer (**CP**), classical stack pointer (**SP**), stack base (**SB**) and base pointer (**BP**). With some operations, negative (**N**), zero (**Z**) and overflow (**W**) flags are altered. All the registers are mapped into the RAM, as shown in table 1.

As shown in table 2, the assembler is very simple.

The assembler instruction set is wanted as realistic as possible, but it is limited. Multi-cycle instructions are allowed, such as the multiplier operator. The instruction set can be divided into several parts: Jump, move, stack procedure-call, math-1, math-2, logic and other instruc-

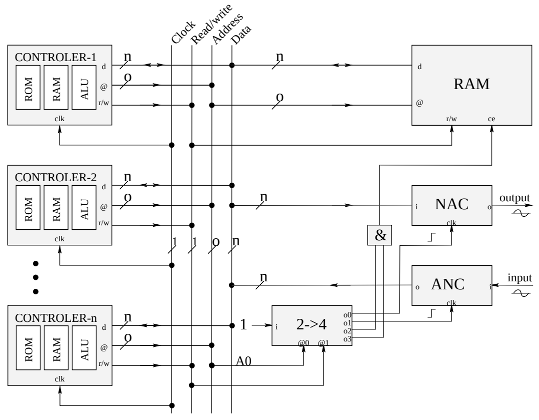


Figure 1: *The parallel architecture for the λ-matrices.*

register	address	name
A	0	first general
B	1	second general
CP	2	code pointer
SB	3	stack base
SP	4	stack pointer
BP	5	base pointer
N	6	negative flag
W	7	overflow flag
Z	8	zero flag

Table 1: Addresses of the controller registers in its local RAM.

code	cycle	NWZ	action
jump			
jmp c	1		$cp \leftarrow c$
jn c	1		$n = 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
jnn c	1		$n \neq 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
jw c	1		$w = 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
jnw c	1		$w \neq 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
jz c	1		$z = 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
jnz c	1		$z \neq 0\ ?\ cp \leftarrow c\ :\ cp \leftarrow cp + 2$
move			
mov @	1		$[@] \leftarrow A, cp \leftarrow cp + 2$
get @	1		$A \leftarrow [@], cp \leftarrow cp + 2$
ldc #	1		$A \leftarrow \#, cp \leftarrow cp + 2$
lda &	1		$A \leftarrow [\&], cp \leftarrow cp + 2$
sta &	1		$[\&] \leftarrow A, cp \leftarrow cp + 2$
stack			
ldsb #	1		$sb \leftarrow \#, sp \leftarrow sb, cp \leftarrow cp + 2$
ldbp &	1		$bp \leftarrow [\&], cp \leftarrow cp + 2$
push &	2		$[sp] \leftarrow [\&], sb \leftarrow sb + 1, cp \leftarrow cp + 1$
pop &	2		$[\&] \leftarrow [sp], sb \leftarrow sb - 1, cp \leftarrow cp + 1$
top #	1		$A \leftarrow [bp - \#], cp \leftarrow cp + 2$
procedure			
call c	4		$push(a), push(bp), push(cp), cp \leftarrow c$
ret	3		$pop(cp), pop(bp), pop(a)$
math-1			
shr &	1	•	$A \leftarrow [\&], A \leftarrow A/2, cp \leftarrow cp + 2$
shl &	1	•	$A \leftarrow [\&], A \leftarrow A * 2, cp \leftarrow cp + 2$
neg &	1	•	$A \leftarrow [\&], A \leftarrow -A, cp \leftarrow cp + 2$
inc &	1	•	$A \leftarrow [\&], A \leftarrow A + 1, cp \leftarrow cp + 2$
dec &	1	•	$A \leftarrow [\&], A \leftarrow A - 1, cp \leftarrow cp + 2$
math-2			
add &	1	•	$B \leftarrow [\&], A \leftarrow A + B, cp \leftarrow cp + 2$
sub &	1	•	$B \leftarrow [\&], A \leftarrow A - B, cp \leftarrow cp + 2$
mul &	3	•	$B \leftarrow [\&], A \leftarrow A * B, cp \leftarrow cp + 2$
div &	3	•	$B \leftarrow [\&], A \leftarrow A/B, cp \leftarrow cp + 2$
mod &	3	•	$B \leftarrow [\&], A \leftarrow A\%B, cp \leftarrow cp + 2$
logic			
and &	1	•	$B \leftarrow [\&], A \leftarrow A\&B, cp \leftarrow cp + 2$
or &	1	•	$B \leftarrow [\&], A \leftarrow A\|B, cp \leftarrow cp + 2$
xor &	1	•	$B \leftarrow [\&], A \leftarrow A \wedge B, cp \leftarrow cp + 2$
nor &	1	•	$B \leftarrow [\&], A \leftarrow !(A\|B), cp \leftarrow cp + 2$
nand &	1	•	$B \leftarrow [\&], A \leftarrow !(A\&B), cp \leftarrow cp + 2$
other			
nop	1		$cp \leftarrow cp + 1$
cmp &	1	•	$B \leftarrow [\&], A - B, cp \leftarrow cp + 2$

• = flags N, W and Z are altered
# = constant
& = local ram address- [&] addressed value
@ = extern ram address- [@] addressed value
c = local code address

Table 2: Instruction set of the controllers.

tions. The particularity of the move instructions is to allow both the communication with the internal RAM (**ldc**, **lda**, **sta**) and with the external global RAM (**mov**, **get**), via the main bus. Because registers are directly mapped into the local RAM, all the instructions which address the local memory may also address the registers: For example, **add 0** is identical to **add A**. An assembler language accomplishes these conversions.

The common memory is a global RAM associated and one I/Os line. In this paper, the number of I/Os line is limited to one, in order to remain simple. I/Os are performed by address decoding: The address **0** is reserved to I/Os.

The main bus of the card is the link between controllers and the common memory. Each controller is directly connected to the bus: Access conflicts are solved at compile-time. Because bus accesses are a bottleneck, some improvements are expected to reduce the data-traffic. The main improvement is to use the local memory of controllers as a cache memory of the global RAM. These considerations as part of the parallel compiler of the λ-matrices, and they are not described here.

3 Analysis

3.1 Object model

The main objective of this analysis is to discover the needed objects with their attributes, and the relationships between these objects. The used method is the classical object-model of the database analysis method [9], as shown in figure 2. Note that in this paper, classes are noted ⟨Class⟩, as the Stk conventions (§ 4).

Controllers have only one RAM and one ROM and they are connected to only one BUS. The COMMON object is the global RAM of the architecture and the associated I/Os lines. It has only one RAM and it is connected to only one bus.

Visual objects are mapped to corresponding simulator objects instead to be specialized to “become” visual. This method is used to preserves the independence of the two parts of the program. In addition, several instance of visual objects are in relation with one simulator object, such as the visual stack and the visual RAM in relation with the simulator local RAM.

From this object model of the applications, the inheritance graph can be directly deduced, as shown in next section.

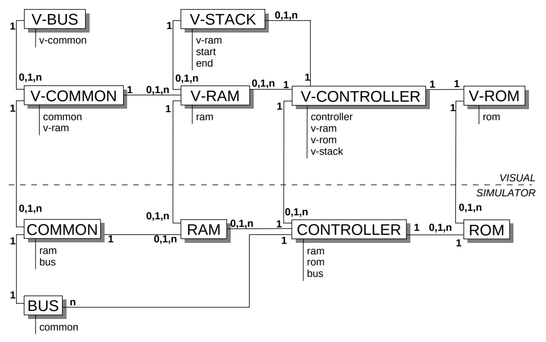


Figure 2: *Object model of the simulator: It shows the relationships between the object instances. Note that **V-** objects belong to the graphical interface.*

3.2 Inheritance graph

This section deals with the classes of the application, deduced from the object-model. The object model focuses on relationships between object instances. The inheritance graph shows how objects are built by specialization of more generic objects: It deals with the *classes* of objects, as shown in figure 3.

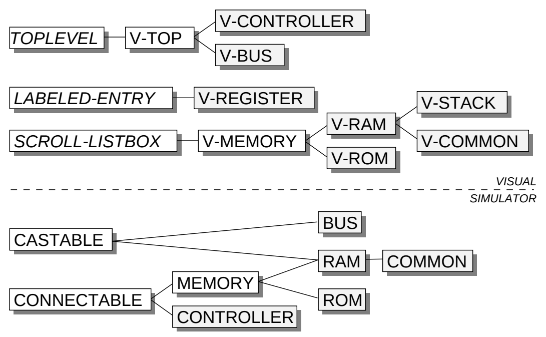


Figure 3: *Object inheritance graph of the simulator: It deals with the object classes. Grayed boxes denote leaf objects that can be directly instantiated. Italic name denote a class belong to the* STKLOS *package..*

⟨Connectable⟩ objects can be connected to a ⟨Bus⟩ object. A ⟨Castable⟩ object convert integer according to a number of bits. Note the multi-inheritance of the ⟨Ram⟩ class. A ⟨Memory⟩ class object can address an array of cells with **read** and **write!** methods. ⟨Ram⟩ and ⟨Rom⟩ inherit from ⟨Memory⟩ class. In order to simplify the code, ⟨Rom⟩ memory also can be accessed in write mode (dumping of programs). Class ⟨Common⟩ is a specialized ⟨Ram⟩ which decodes its address 0 for I/Os.

4 The object functional scheme language: Stk

The simulator is implemented with Stk [5]: It is a SCHEME [3, 1] language compliant with the language described in the *Revised*^{4} *Report on the Algorithmic Language Scheme.* Stk is written by Erik Gallesio.

TK is a powerful X11 graphical toolkit defined at the University of Berkeley by John Ousterhout [10]. This toolkit provides to the user with hight level widgets such as Buttons or Menus and is easily programmable. In particular, little knowledge of X fundamentals is needed to build an application. TK package relies on an interpretative language named TCL.

Stk is another implementation of SCHEME programming language. The main feature of Stk is to provide a full integration of the TK toolkit in SCHEME. In this implementation, SCHEME establishes the links between the user and the TK toolkit, since it substitutes the TCL language.

STKLOS is the object-oriented layer of Stk. Its implementation is derived from version 1.3 of the Gregor Kickales Tiny Clos Package [8]. However, it has been extended to be as close as possible to Clos, the Common Lisp Object System [7].

Briefly stated, the STKLOS extension gives the user a full object-oriented system with meta-classes, multiple inheritance, generic functions and multiple-methods [11]. Furthermore, the whole implementation relies on a true metaclass object protocol.

This model has also been used to embody the predefined TK widgets in the hierarchy of STKLOS classes. This set of classes permits to simplify the core TK usage by providing homogeneous accesses to widgets options and hiding the low level details of the TK widgets, such as naming conventions. Furthermore, as expected, using objects facilitates code reuse and definition of new widgets classes.

5 Implementation

In this section, the full code of the simulator is given. Its graphical interface is not describe. Files of this program can be obtained in a free public access way. Electronic addresses may be obtained from the authors.

5.1 Preliminary

The main principles of the object-oriented layer of Stk used here are inheritance (static feature sharing) and method specialization (dynamic feature sharing). These two kinds of feature sharing significantly reduce the code. Let us briefly introduce the defined objects.

The architecture consists of a global bus where chips are connected: Many controllers and one common memory.

The ⟨BUS⟩ has two functions: Receives user's clock signals and propagate them to the connected chips, checks accesses to the global memory and manages the potential conflicts. In addition, it deals with user's reset signals. Its main methods are **connect** (chips connection), **reset** (chips reset), **clock** (chips clock), **read** and **write!** (accesses to the common memory).

⟨Common⟩ memory is the global RAM plus I/Os management. In order to simulates I/Os, common object has one input file and one output file. These files are accessed when a controller read or write to the global address 0. Methods of this class are **read**, **write!**, and some additional initialization methods.

⟨Ram⟩ and ⟨Rom⟩ are a kind of ⟨MEMORY⟩. A memory is an array of cells that contain a value. This array is indexable with an address. In order to simplify the coding (especially the interface with visual objects), the class

⟨ROM⟩ also defines a **write!** method which allows program dumping with the **dump** method. The class ⟨MEMORY⟩ supports **read** and **write** methods.

A ⟨CONTROLLER⟩ has a RAM, a ROM, an ALU. Its registers are mapped into its RAM, as in the real little controllers. Most of the methods are relative to assembler instruction decoding. In addition, it can be found **reset**, **clock**, **microInstruction**, **macroInstruction** and some other methods.

In addition, two root classes are defined for inheritance: ⟨CONNECTABLE⟩ specifies an object that can be connected to a bus, and ⟨CASTABLE⟩ denotes an object which contain an integer coded with a limited number of bits.

The following sections presents the simulator code ordered from the more specialized to the simpler.

5.2 The main program: Card creation

The main program is presented. It consists on some object creations and a loop which send the clock signal to the main bus. Here, a 2-controllers card is modeled:

```
(define bs (make ⟨Bus⟩))

(define cm (make ⟨Common⟩
                 :bus bs
                 :size 1000
                 :in "input.dat"
                 :out "output.dat"))

(define c1 (make ⟨Controller⟩
                 :bus bs
                 :ram 200
                 :rom 200
                 :code "example-2-1.S"))

(define c2 (make ⟨Controller⟩
                 :bus bs
                 :ram 200
                 :rom 200
                 :code "example-2-2.S"))

(let mainLoop ()
  (clock bs)
  (mainLoop))
```

The global bus is created. Then the common memory is created with some options: Bus, RAM size, input file and output file. The two controllers are created with global bus, RAM and ROM sizes, and the file name where is the assembler program as options. Note that the number of created controllers is not limited.

The main bus is clocked by **mainLoop** tail-recursive named ***let***. Then the input file **input.dat** is read and the output file **output.dat** is written.

5.3 Bus object definitions

The global bus is an interface with the user: It offers some commands lines such as **clock** and **reset**, and an access to the transfered data with both data and address busses. The bus object must detect potential common memory access conflicts. This is accomplished with the **clock** code. When the bus is clocked, each controller is clocked too, and its current instruction is executed. This instruction may be a read/write access on the global bus: Such instructions are stored in a list. At the end of the bus clock call, bus accesses are checked in order to detect addressing conflicts.

As the model of architecture, class ⟨BUS⟩ has some attributes: A common memory, a list of connected controllers. The two last attributes **lread** and **lwrite** are lists used to check bus accesses:

```
(define-class ⟨Bus⟩ (⟨Castable⟩)
  ((common    :accessor common)
   (connected :initform '() :accessor connected)
   (lread     :initform '() :accessor lread)
   (lwrite    :initform '() :accessor lwrite)))
```

reset deletes the list of read/write accesses and it reinitializes each connected chip:

```
(define-method reset ((self ⟨Bus⟩))
  ; flushes the lists
  (set! (lread  self) '())
  (set! (lwrite self) '())
  ; resets connected chips
  (reset (common self))
  (map (lambda (controller) (reset controller))
       (connected self)))
```

connect allows a chip to be connected to the bus. Two kinds of chip can be connected: Common memory and other chips. First one alters the specific **common** attribute of the instance. Other chips are added into **connected** attribute list with the following function:

```
(define-method connect ((self ⟨Bus⟩) chip)
  (set! (connected self)
        (cons chip (connected self))))
```

Following **connect** function is more specialized than the previous: It connects especially the ⟨COMMON⟩ chips:

```
(define-method connect ((self ⟨Bus⟩)
                        (chip ⟨Common⟩))
  (set! (common self) chip))
```

The next method occurs when a controller wants to read the bus. It adds the list formed by **controller**, **address** and **procedure** arguments to the ⟨BUS⟩ instance **lread** list. The **procedure** will be called when all the controller bus accesses are registered.

```
(define-method read ((self ⟨Bus⟩)
                     (controller ⟨Controller⟩)
                     address
                     procedure)
  (set! (lread self)
        (cons (list controller address procedure)
              (lread self))))
```

write! function adds to the **lwrite** list the list formed by **controller**, **address** and **value** arguments:

```
(define-method write! ((self ⟨Bus⟩)
                       (controller ⟨Controller⟩)
                       address
                       value)
  (set! (lwrite self)
        (cons (list controller address (cast self value))
              (lwrite self))))
```

The following **clock** method is the main method of the system. Connected chips are clocked. Each clocked controller may access to the bus. These calls are stored into the **lread/lwrite** lists of the bus, as shown in the **read/write!** methods. Then all bus access controls can be accomplished:

```
(define-method clock ((self ⟨Bus⟩))
  ; checks if each address of the lread list matches
  (let ((callProcRead
         (lambda (address value)
           (map (lambda (ContAdProc)
                  (if (eq? address (cadr ContAdProc))
                    (apply (caddr ContAdProc)
                           (list value))))
                (lread self)))))
    ; flushes bus access lists
    (set! (lread  self) '())
    (set! (lwrite self) '())
    ; clocks each controller
    (map (lambda (controller) (clock controller))
         (connected self))
    ; bus accesses controls
    (case (length (lwrite self))
      ; no write access
      (0 (if (not (zero? (lread self)))
          (let* ((address (cadar (lread self)))
                 (value (cast self (read (common self) address))))
            (callProcRead address value))))
        ; one write access
        (1 (let ((address (cadar (lwrite self)))
                 (value (cast self (caddar (lwrite self)))))
             (callProcRead address value)
             (write! (common self) address value)))
        ; too many write accesses
        (else
          (error 'Too many writes accesses on the bus")))))
```

5.4 Controller object definitions

In this paper, the internal structure of the controllers is not described. We only focus on a realistic interface (assembler language). A controller is constituted by a RAM, a ROM, an ALU and some registers. In order to remain simple, registers are directly mapped onto the RAM memory. The first ten cells of the RAM are reserved. In addition, a controller has a list of next micro-instructions executed in more than one cycle. This list may be empty. The class declaration is:

```
(define-class ⟨Controller⟩ (⟨Connectable⟩)
  ((ram :accessor ram)
   (rom :accessor rom)
   (micro :accessor micro :initform '())
   ; the following register value is its address into the ram
   (A   :accessor A   :initform 0)
   (B   :accessor B   :initform 1)
   (CP  :accessor CP  :initform 2)
   (SB  :accessor SB  :initform 3)
   (SP  :accessor SP  :initform 4)
   (BP  :accessor BP  :initform 5)
   (W   :accessor W   :initform 6)
   (Z   :accessor Z   :initform 7)))
```

The initialization function of ⟨Controller⟩ deals with **:ram** option for the size of the RAM, with **:rom** for the size of the ROM and with **:code** for the filename of the program:

```
(define-method initialize ((self ⟨Controller⟩) arguments)
  (next-method)
  (set! (ram self)
        (make ⟨Ram⟩ :size (get-keyword :ram arguments 100)))
  (set! (rom self)
        (make ⟨Rom⟩ :size (get-keyword :rom arguments 100)))
  (dump self (get-keyword :code arguments '())))
```

The **dump** function loads assembler instructions from a file into the controller ROM. It uses the **dump** method of ⟨Rom⟩ which operates with a list of opcodes. This list is obtained with ***port->list*Stk** function and the reader ***read***:

```
(define-method dump ((self ⟨Controller⟩) fileName)
  (dump (rom self)
        (port->list read (open-intput-file fileName))))
```

The **reset** function applied onto ⟨Controller⟩ flushes the micro-instruction list and sets the first ten reserved cells of RAM to zero: These addresses are mapped to the internal registers.

```
(define-method reset ((self ⟨Controller⟩))
  (set! (micro self) '())
  (let loop ((ad 0))
    (if (not (eq? ad 10))
      (begin
        (write! (ram self) ad 0)
        (loop (+ 1 ad))))))
```

The two following functions allow to access in a named way to the controller registers. The **register** methods have two behaviors: Read access without a value, and write access, with a value:

```
; read access to the registers
(define-method register ((self ⟨Controller⟩) reg)
  (read (ram self) (reg self)))
```

```
; write access to the registers
(define-method register ((self ⟨Controller⟩) reg value)
  (write! (ram self) (reg self) value))
```

The **goto** method changes the value of the code pointer register (**CP**) to **address**. It is used by the jump functions:

```
(define-method goto ((self ⟨Controller⟩) address)
  (register self CP address))
```

The next method sets negative, zero and overflow flags according to **value**:

```
(define-method setFlags ((self ⟨Controller⟩) value)
  (register self N (if (< value 0) 1 0))
  (let ((casted (cast (ram self) value)))
    (register W self (if (eq? value casted) 0 1)))
  (register self Z (if (zero? value) 1 0)))
```

This **read** function returns the next instruction from the ROM at the address pointed by the code pointer **CP** and it increments the code pointer:

```
(define-method read ((self ⟨Controller⟩))
  (let ((opcode (read (rom self) (register self CP))))
    (goto self (+ (register self CP) 1))
    opcode))
```

The **clock** method deals with the next instruction to process. This instruction can be read either from the ROM, or from the head of the micro-instruction list if it is not empty:

```
(define-method clock ((self ⟨Controller⟩))
  (if (null? (micro self))
     (macroInstruction self)
     (microInstruction self)))
```

If the micro-instruction list is not empty, the next instruction to be processed is in the head of this list. In addition, the list is set to its tail:

```
(define-method microInstruction ((self ⟨Controller⟩))
  (eval (car (micro self)))
  (set! (micro self) (cdr (micro self))))
```

Macro-instructions are read from the ROM. According to the type of the instruction, an appropriate function is invoked:

```
(define-method macroInstruction ((self ⟨Controller⟩))
  (let ((opcode (read self)))
    (case opcode
      ((nop)
       'nothing)
      ((cmp
       (cmpInstruction self (read self))))
      ((jmp jn jnn jw jnw jz jnz)      ; jump instructions
       (jumpInstructions self opcode (read self)))
      ((mov get ldc lda sta exc)       ; move instructions
       (moveInstruction self opcode (read self)))
      ((ldsb ldbp push pop top)        ; stack instructions
       (stackInstruction self opcode (read self)))
                                       ; math unary instructions
      ((and or xor nor nand            ; logical instructions
        shr shl neg inc dec)           ; math 1  instructions
       (math1Instructions self opcode (read self)))
      ((add sub mul div mod)           ; math binary instructions
       (math2Instructions self opcode (read self)))
      (else
       (error "Unknown instruction: ~a~%" opcode)))
    ; the Z flag is altered with some instructions
    (case opcode
       ((neg inc dec add sub and or xor nor nand)
        (setFlags self (register self A))))))
```

Now, each instruction can be processed. The next function deals with the **cmp** instructions. The micro-code can be deduced from the table 2:

```
(define-method cmpInstruction ((self ⟨Controller⟩)
                               operand)
  ; set B with the value at the address operand
  (register self B (read (ram self) operand))
  ; set Z according to the A-B result
(setFlags self (- (register self A) (register self B))))
```

Jump instructions change on a condition the value of the code pointer **CP**:

```
(define-method jumpInstructions ((self ⟨Controller⟩)
                                opcode operand)
  (case opcode
    ; unconditional jump
    (jmp (goto self operand))
    ; jump if N == 1
    (jn  (if (zero? (register self N))
            (goto self operand)))
    ; jump if N != 1
    (jnn (if (not (zero? (register self N)))
            (goto self operand)))
    ; jump if W == 1
    (jw  (if (zero? (register self W))
            (goto self operand)))
    ; jump if W != 1
    (jnw (if (not (zero? (register self W)))
            (goto self operand)))
    ; jump if Z == 1
    (jz  (if (zero? (register self Z))
            (goto self operand)))
    ; jump if Z != 1
    (jnz (if (not (zero? (register self Z)))
            (goto self operand)))))
```

Move instructions deals with data transfers between registers and either local or global memories:

```
(define-method moveInstructions ((self ⟨Controller⟩)
                                opcode operand)
  (case opcode
    (mov
      ; write on the bus the value in A
      (write! (bus self) self operand (register self A)))
    (get
      ; put in A the value read from the bus
      (read (bus self) self operand
          (lambda (value)
              (register self A value))))
    (ldc
      ; load in A a constant
      (register self A operand))
    (lda
      ; load in A the value at the address operand
      (register self A (read (ram self) operand)))
    (sta
      ; stores at the address operand the value in A
      (write! (ram self) operand (register self A)))
    (exc
      ; exchange A and the value at the address operand
      (let ((tmp (register self A)))
        (register self A (read (ram self) operand))
        (write! (ram self) operand tmp)))))
```

Stack instructions alters specific stack registers. Instructions are **push**, **pop** and **top**. Most of these instructions are complex and take more than one cycle. In order to simulate the multiple-cycle instructions, a list of *actions* is affected to the **micro** attribute. This is built with a quasi-quote. Each action will be evaluated latter:

```
(define-method stackInstructions ((self ⟨Controller⟩)
                                opcode operand)
  (case opcode
    (ldsb
      ; put in SB and SP the constant operand
      (register self SB operand)
      (register self SP (register self SB)))
    (ldbp
      ; put in BP the value at the address operand
      (register self BP (read (ram self) operand)))
    (push
      ; write at the address in SP ...
      ; ... the value at the address operand
      (write! (ram ,self) (register ,self SP)
            (read (ram ,self) ,operand))
      ; micro-coded
      (set! (micro self)
          `(; decrement SP
            (register ,self SP (dec (register ,self SP))))))
    (pop
      ; write at the address operand ...
      ; ... the value at the address SP
      (write! (ram ,self) ,operand
            (read (ram ,self) (register , self SP)))
      ; micro-coded
      (set! (micro self)
          `(; decrement SP
            (register ,self SP (inc (register ,self SP))))))
    (top
      ; put in A the value at the address BP-operand
      (register ,self A
            (read (ram self) (sub (register self BP) operand))))))
```

Unary mathematical instructions have one argument. **opcode** is a symbol that corresponds to a SCHEME function. The function **eval** is used in order to evaluate the list formed by the opcode and the argument:

```
(define-method math1Instructions ((self ⟨Controller⟩)
                                opcode operand)
  (register self A (eval (list opcode (read (ram self) operand)))))
```

Binary mathematical operators have two arguments **A** and **B**. Some of these operators need more than one instructions-cycle, so, they are micro-coded:

```
(define-method math2Instructions ((self ⟨Controller⟩)
                                opcode operand)
  (case opcode
    ((mul div mod)
      (register ,self B (read (ram ,self) ,operand))
      ; micro-coded
      (set! (micro self)
          `((register ,self A (eval (list ,opcode
                                      (register ,self A)
                                      (register ,self B))))
            (setFlags ,self (register ,self A)))))

    (else
      (register self B (read (ram self) operand))
      (register self A (eval (list opcode
                                 (register self A)
                                 (register self B)))))))
```

In addition, all the operators must be defined, such as **add** with (***define*** **add** +).

5.5 Common memory and I/Os object definitions

Common memory is a RAM which decodes address **0** accesses for I/Os. In order to simulate the outer world, input values are read from a file stored in attribute **in** and output values are written into a file stored in attribute **out**. **CurrentIn/Out** values are used by the graphical interface:

```
(define-class ⟨Common⟩ (⟨Ram⟩)
  ((in  :accessor in)
   (out :accessor out))
   (cin :accessor currentIn)
   (cout:accessor currentOut)))
```

The **initialize** method scans the command line (**arguments**) for **:in** option which initializes the input the input file and for **:out** for the output file:

```
(define-method initialize ((self ⟨Common⟩) arguments)
  (let ((port (get-keyword :in arguments '())))
    (set! (in self)
        (if (null? port) '()
            (open-intput-file port))))
  (let ((port (get-keyword :out arguments '())))
    (set! (out self)
        (if (null? port) '()
            (open-output-file port))))
  (next-method))
```

When the **read** method is invoked, the address is decoded. If the address is not 0, then the **next-method** is called: This call invokes the ⟨Ram⟩ read method. If the address 0 is decoded, the input file is read (if it is possible):

```
(define-method read ((self ⟨Common⟩)
                     address)
  (if (zero? address)
    (let ((value 0))
      (if (not (null? (in self)))
        (begin
          ; uses the SCHEME read function
          (set! value (read (in self)))
          (if (eof-object? value)
           (begin
             (close-input-port (in self))
             (set! value 0)
             (set! (in self) '())))))
      (set! (currentIn self) (cast self val))
      (currentIn self))
    (next-method)))
```

The **write!** method has the same structure than the **read** one, but if the address 0 is decoded, the value is simply written on the output file:

```
(define-method write! ((self ⟨COMMON⟩)
                       address value)
  (if (zero? address)
    (if (not (null? (out self)))
      (begin
        (set! (currentOut self) (cast self value))
        (write (currentOut self) (out self))))
    (next-method)))
```

5.6 Memory object definitions

A memory is an array of cells (vector). ⟨MEMORY⟩ inherits from ⟨CONNECTABLE⟩ (it can be connected to a bus) and ⟨CASTABLE⟩ (stored value may be casted):

```
(define-class ⟨MEMORY⟩(⟨CONNECTABLE⟩⟨CASTABLE⟩)
  ((array :accessor array)))
```

The classes ⟨RAM⟩ and ⟨ROM⟩ simply inherits from ⟨MEMORY⟩:

```
(define-class ⟨RAM⟩ (⟨MEMORY⟩) ())
(define-class ⟨ROM⟩ (⟨MEMORY⟩) ())
```

The **initialize** method creates **array** vector according to **:size** option:

```
(define-method initialize ((self ⟨MEMORY⟩) arguments)
  (next-method)
  (set! (array self)
      (make-vector (get-keyword :size  arguments 0) 0)))
```

The memory size is the size of its internal vector:

```
(define-method size ((self ⟨MEMORY⟩))
  (vector-length (array self)))
```

The **read** function accesses to the **array** vector in order to retrieve the value at the index **address**. If the address is to big, **0** is returned:

```
(define-method read ((self ⟨MEMORY⟩) address)
  (if (address (size self))
      (vector-ref (array self) address)
    0))
```

The **write!** function puts **value** in the vector at index **address**:

```
(define-method write! ((self ⟨MEMORY⟩)
                       address value)
  (if ( address (size self))
    (vector-set! (array self) address (cast self value))))
```

The next **write!** method is specialized for the ⟨ROM⟩. It allows to dump a program into a ROM. This function is especially used by the graphical interface. So, the written values are not casted because they may be assembler instructions:

```
(define-method write! ((self ⟨ROM⟩)
                       address value)
  (if ( address (size self))
    (vector-set! (array self) address value)))
```

The **dump** method is reserved for a ROM. It copies a list of value in the vector with the use of classical SCHEME named **let** construct:

```
(define-method dump ((self ⟨ROM⟩) list)
  (let loop ((ad 0)(l list))
    (if (not (null? l))
      (begin
        (write! self ad (car l))
        (loop (+ ad 1) (cdr l))))))
```

5.7 Connectable object definitions

A connectable object can be connected to a bus. So, it support methods **reset** and **clock** in addition to the **initialize** method which deals with the creation options:

```
(define-class ⟨CONNECTABLE⟩ ()
  ((bus :accessor bus)))
```

The standard initialization method scans optional arguments (**arguments**) to find a **:bus** option in order to allocates **bus** attribute of the instance. By default, an instance of ⟨NULLBUS⟩ is created:

```
(define-method initialize ((self ⟨CONNECTABLE⟩) arguments)
  (next-method)
  (let ((bs (get-keyword :bus arguments (make ⟨NULLBUS⟩))))
    (set! (bus self) bs)
    (connect bs self)))
```

The **reset** function is an user-call. It has no-effect on a ⟨CONNECTABLE⟩ object:

```
(define-method reset ((self ⟨CONNECTABLE⟩))
  'nothing)
```

The **clock** function simulates the global clock of the system. It has no-effect on a ⟨CONNECTABLE⟩ object:

```
(define-method clock ((self ⟨CONNECTABLE⟩))
  'nothing)
```

5.8 Castable object definitions

Castable object are associated to a number of bit for integer coding. **cast** method alter a given number according to this number of bits. The class defines both an upper and a lower limits for the given value. When a value given to the **cast** method is not comprised between these two limits, the nearest limits is returned.

```
(define-class ⟨CASTABLE⟩ ()
  ((width :accessor width :initform 16 :init-keyword :width)
   (max   :accessor maximum)
   (min   :accessor minimum)))
```

Both the upper and lower limits are computed at init-time for computation improvement:

```
(define-method initialize ((self ⟨CASTABLE⟩) arguments)
  (next-method)
  (set! (maximum self)    (expt 2 (width self)))
  (set! (minimum self) (- (expt 2 (width self)))))
```

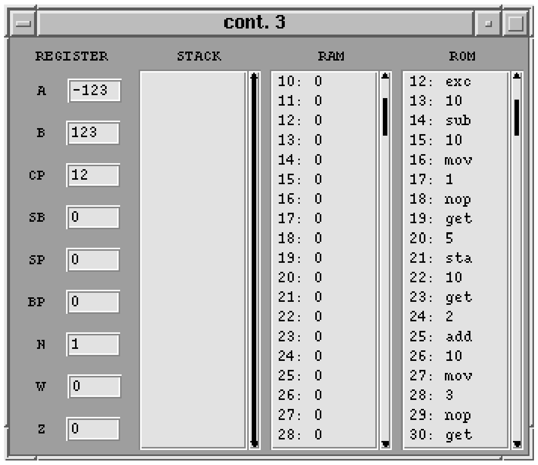
The **cast** method transform a number according to the width of the object:

```
(define-method cast ((cast ⟨CASTABLE⟩) value)
  (if   (< value (minimum self)) (minimum self)
    (if (> value (maximum self)) (manimum self)
      value))
```

6 Graphical interface

This section deals with the graphic interface layout of the simulator. Implementation details are not discussed in this paper.

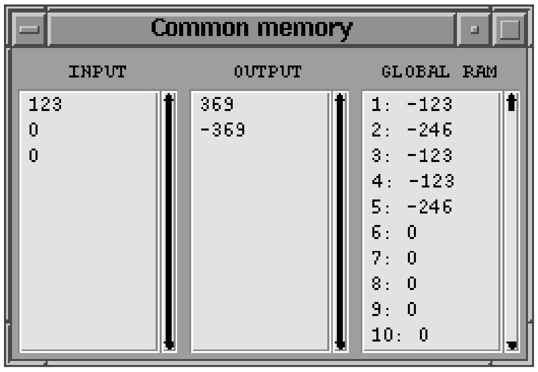
As shown in figure 4, **cont. 1** is an instance of the class ⟨VisualController⟩. It can be seen the graphic representation of registers, stack (as ⟨VisualStack⟩), RAM (as ⟨VisualRam⟩) and ROM (as ⟨VisualRom⟩). The interface allows dynamic changes of the memory contents.



```
A   -123        10: 0     12: exc
B   123         11: 0     13: 10
CP  12          12: 0     14: sub
SB  0           13: 0     15: 10
SP  0           14: 0     16: mov
BP  0           15: 0     17: 1
N   1           16: 0     18: nop
W   0           17: 0     19: get
Z   0           18: 0     20: 5
                19: 0     21: sta
                20: 0     22: 10
                21: 0     23: get
                22: 0     24: 2
                23: 0     25: add
                24: 0     26: 10
                25: 0     27: mov
                26: 0     28: 3
                27: 0     29: nop
                28: 0     30: get
```

Figure 4: *Graphical interface of a controller.*

The graphic representation of the common memory can be seen in figure 5. It can be seen the global RAM which begins at the address **1**, because the address **0** is reserver for I/Os. In addition, there are two lists of values: Input values read from the input file, and output values written into the output file.



```
123     369     1: -123
0       -369    2: -246
0               3: -123
                4: -123
                5: -246
                6: 0
                7: 0
                8: 0
                9: 0
                10: 0
```

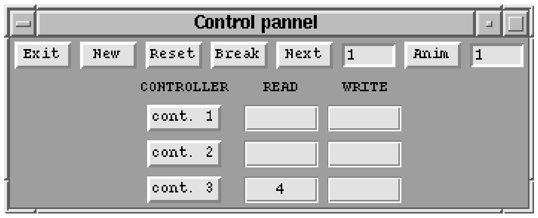
Figure 5: *Graphical interface of the common memory.*

Control panel is the graphical representation of the main bus, as in figure 6. Global memory accesses can be seen for each controllers. The system can be clocked either step by step or in an animation way. Breakpoints can be instantiated for each controller.

The complete graphic interface code takes about 600 lines because the TK interface and its object-oriented layer is particularly efficient in Stk.

7 Conclusion

In this paper, a practical use of a dynamic object-oriented language is shown. The efficiency of such languages permits to put in the pages of this article the specificatioos



```
Next 1    Anim 1
cont. 3   READ: 4
```

Figure 6: *Graphical interface of the main bus.*

of the problem, a short object-oriented analysis, and the totality of the code.

A parallel architecture is modeled here. The description may be as simple as possible. It is the preliminary work which leads an hardware implementation of the architecture and relative softwares such as an assembler/debugger, a profiler, ...

The analysis shows how to use the well known database object-analysis in other kind of applications. This model focuses on the leaf objects (that can be instantiated) and on their inner relationships. From this first step of analysis, the inheritance graph can be deduced: It shows how classes of objects are built.

The coding of this model has two phases: Classes description and their associated methods. Classes descriptions use inheritance mechanism which deals with a static sharing of features. Methods uses the method-specialization mechanism which deals with dynamic sharing of features. These mechanisms greatly increase the reuse of pieces of code.

At last, the graphic interface of the simulator is build as an independent applications. Only the layout of this interface is shown in this article.

The resulting code is actually short and efficient. Anyone can rapidly modify it. Comments are not needed in each line because the Stk language acts such as a specification language.

References

[1] H. Abelson, G.J. Susman, and J. Susman. *Structure and Interpretation of Computer Programs.* MIT Press, 1987.

[2] M. Auguin and F. Boéri. The opsila computer. *in Parallel Algorithms and architectures, Eds. M. Cosnard et coll., North Holland*, pages 143–154, 1986.

[3] W. Clinger and J. Rees. Revised^4 report on the algoritmic language scheme. Technical report, MIT Artificial Intelligence Laboratory, CSDTR 174, october 1990.

[4] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles.* InterÉdition, 1993.

[5] E. Gallesio. Stk reference manual, version 2.2. Technical report, Laboratoire I3S-CNRS URA 1376 - ESSI, e-mail:kaolin.unice.fr:/pub/, october 1995.

[6] A. Jaffer. Scm : A scheme implementation 4. Technical report, jaffer@ai.mit.edu, 1990.

[7] GL. Steele JR. *Common Lisp: The Language, 2nd Edition.* Digital Press (Bedford, MA), 1990.

[8] G. Kickzales. Tiny-clos. Source available on ftp.xerox.com in directory /pub/mops, December 1992.

[9] S. Miranda. *L'art des bases de données - Tome I : Introduction aux bases de données.* Eyrolles, 1988.

[10] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling.* Prentice-Hall, 1991.

[12] A. Tanenbaum. *Structured Computer Organisation (Third Edition).* Prentice-Hall, Incorporated, 1990.

[13] A. Valencia. A tutorial of GNU SMALLTALK. Technical report, Valencia Consulting, november 1993.