TR-561

Abstract A'UM Machine

by

K. Yoshida & T. Chikayama

May, 1990

©1990, ICOT

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Abstract $\mathcal{A}'\mathcal{UM}$ Machine

Kaoru Yoshida * and Takashi Chikayama †

Institute of New Generation Computer Technology (ICOT)
1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

6 March, 1990

Abstract

$\mathcal{A}'\mathcal{UM}$ is a stream-based concurrent object-oriented programming language. This report describes an abstract $\mathcal{A}'\mathcal{UM}$ instruction set, called $\mathcal{A}'\mathcal{UM}$-α , and its interpreter, called the *abstract $\mathcal{A}'\mathcal{UM}$ machine*. The $\mathcal{A}'\mathcal{UM}$-α instruction set does not require any special hardware: it can be implemented in any form of software, firmware or hardware. Key features of the $\mathcal{A}'\mathcal{UM}$-α instruction set are its (1) merger-intensive design, (2) sequential control, (3) sender-subjective transmission, (4) implicit argument handling, (5) implicit freezing/melting of every built-in operation, and (6) incremental garbage collection. It is shown that a software-naive implementation of the abstract $\mathcal{A}'\mathcal{UM}$ machine on a conventional von Neumann machine has attained reasonable performance. With optimization and improvement, higher performance can be expected.

1 Introduction

$\mathcal{A}'\mathcal{UM}$ is a stream-based concurrent object-oriented programming language [Yoshida and Chikayama 88B, Yoshida 90A].

This report describes an abstract $\mathcal{A}'\mathcal{UM}$ instruction set, called $\mathcal{A}'\mathcal{UM}$-α , and its interpreter, called the *abstract $\mathcal{A}'\mathcal{UM}$ machine.*

*email address: yoshida{@icot.jp, %icot.jp@relay.cs.net}

†email address: chikayama{@icot.jp, %icot.jp@relay.cs.net}

1.1 Objectives

The abstract $\mathcal{A}'\mathcal{UM}$ machine has been designed with the following aims:

1. *Abstractness and Portability.*
   The $\mathcal{A}'\mathcal{UM}$-α instruction set should be abstract enough for it to be implemented in any form of software, firmware or hardware. The design should exclude machine-dependent features as much as possible.

2. *Efficient Sequential Implementation.*
   The abstract $\mathcal{A}'\mathcal{UM}$ machine should run efficiently, especially on conventional von Neumann machines in which efficient sequential control can be exploited. For efficiency, some restrictions and extensions are introduced to the language.

   The present design described in this report assumes the use of a single processor. Just one schedule table is prepared. Considerations for parallel execution control, such as load-balancing, have not been made yet. Parallel implementation will be our future work.

3. *Extensibility to Shared Memory Parallel Implementations.*
   Although the abstract $\mathcal{A}'\mathcal{UM}$ machine is currently implemented on a single processor, it should be possible to extend the design for future parallel implementation on a shared-memory multiprocessor.

1.2 The $\mathcal{A}'\mathcal{UM}$ Computation Model

We briefly summarize the $\mathcal{A}'\mathcal{UM}$ computation model so that this report will be self-contained. For more detail, see [Yoshida 90A].

The $\mathcal{A}'\mathcal{UM}$ computation model is the reduction of a directed graph which consists of a finite set of arcs and a finite set of nodes. Each arc is a pair of *terminals*, called the *inlet* (head) and the *outlet* (tail), and represents the relation that the nodes ahead of the inlet should *precede* those behind the outlet.

When an arc is created, its two terminals, inlet and outlet, are both *undefined*. Each terminal is later *instantiated* to one of the four kinds of concrete nodes:

1. *objects*, each of which has an incoming arc

2. *messages*, each of which has an outgoing arc and an incoming arc

3. *joints*, each of which has an outgoing arc and two incoming arcs

4. *nils*, each of which has an outgoing arc

An *incoming arc* to a node is an arc whose head designates the node; an *outgoing arc* to a node is an arc whose tail designates the node.

(a) object (b) message (c) joint (d) nil

We call a sequence of messages each connected by arcs a *stream*, and a structure composed of streams and joints a *tree of streams*. A tree of streams is formed toward each object.

Objects and messages may contain terminals of arcs. These terminals are undefined. The terminals in a message may be instantiated to concrete nodes when the message arrives at some object. The terminals in an object may be instantiated to concrete nodes when the object is activated. These arcs are used for connecting one tree of streams toward an object, with another three of streams toward another object. Hence, as a whole, a directed graph, which is composed of trees of streams each toward an object, exists during computation.

Each joint represents an ordering relation of two incoming streams. There are two kinds of joints: *append joints* and *merge joints*.

- An append joint is a relation in which the messages from the first incoming stream should precede those from the second incoming stream and should follow the outgoing stream.

- A merge joint is a relation in which the messages from the two incoming streams should follow the outgoing stream in any order.

Instantiating an undefined terminal to a concrete node is called *chaining*; releasing a concrete node from a terminal of an arc is called *unchaining*.

Operationally, each joint unchains the two incoming streams, chains a stream that should keep the above ordering relation, and connects it to the outgoing stream.

When the inlet of the outgoing arc of a message or nil is instantiated to an object, the object may be *activated*. When an object is activated, it unchains the message or nil, creates new nodes and arcs, and chains them.

Thus, the reduction consists of create, chain and unchain operations. Given a graph, the graph is reduced until it contains no more arcs with undefined terminals.

For intuitive understanding, we name the basic chaining and unchaining operations as follows:

- *send:* chaining the outlet of an arc and the inlet of another arc to a message
- *close:* chaining an arc to a nil
- *receive:* unchaining a message from an arc
- *is_closed:* unchaining a nil from an arc

Hereafter, we use the word, *object*, in a different meaning. We will refer to what has been called an object as a *generation*. Among those nodes which a generation may create, there is at most one representing its *next generation*, called *self*. A sequence of generations will be called an *object*.

1.3 Characteristic Features of $\mathcal{A}'\mathcal{UM}$

The $\mathcal{A}'\mathcal{UM}$ computation model is characterized by three features:

1. *Object Orientation.*
   - *Objects* communicate with each other via message-passing.
   - Each object consists of *generations*.
   - Objects may hold terminals of streams in their slots.
   - Primitive objects such as integers are treated in the same way as abstract objects.
   - Conditional branching and looping are both realized in the same framework of an object, called a *volatile object*.
   - Multiple class inheritance is supported for the purpose of minimizing the source program code size.
2. *Stream Computation.*
   - Transmission of messages between the sender object and the receiver object are explicitly represented as a sequence of messages, called a *stream*. For each object, the stream toward it is split as the number of senders increases. As a result, a tree of streams is formed toward each object.

- Every time a stream is split into two, the *joint* of the two branch streams represents either of the two kinds of binary stream operations: *append* and *merge*. If the joint is append, the messages from the first incoming stream arrive at the destination object earlier than those from the second incoming stream. If the joint is merge, the messages from from the two incoming streams arrive at the destination object in a nondeterministic order.

  Note that in either joint, the order of the messages in each incoming stream is preserved in the order of their arrival at the destination object.

- Messages may contain terminals of streams as their arguments. Such a message that contains terminals of streams as its arguments works as a stream connector. When a message arrives at an object, each argument terminal is connected to a stream in the receiver's scope.

3. *Relational Programming.*

   - Programs can be declaratively read. How to execute a program is independent of how to write a program. A tree of streams can be constructed from any part, whether from the part closer to the destination object or from the part farther from the destination object. The destination object may be created at any time, whether earlier or later than the construction of the tree of streams toward it.

In the implementation of $\mathcal{A'UM}$, we should consider how to implement the following efficiently:

- Generations
- Slot access
- Primitive objects and built-in operations
- Volatile objects
- Class inheritance and class management
- Streams and joints (especially mergers)
- Relay and transmission mechanisms

1.4 Key Features of $\mathcal{A}'\mathcal{UM}$-α

The $\mathcal{A}'\mathcal{UM}$-α instruction set is characterized by the following features:

1. *Merger-Intensive Design.*
   Each communication path between the sender and the receiver is constructed of mergers. Each merger is a relay which buffers messages while the destination is undefined, and forwards them when the destination is determined. The merger function is also embedded in each object.

2. *Sequential Control.*
   The $\mathcal{A}'\mathcal{UM}$-α instruction exploits sequential and efficient control, which imposes some restrictions on the language specification.

   (a) *Execution Order Dependent.*
   The order of messages which should be sent to the same destination is determined by the execution order of sending instructions.

   (b) *Sequential Execution of Generations.*
   For each object, no more than one generation is created. Concurrency assuming the existence of multiple generations is restricted. Those programs in which a past generation of an object waits for a future generation of the object to return some result will fall into deadlock (Figure 1).

   (c) *Sequential Execution of Conditional Objects.*
   Conditional (volatile immutable) objects appearing in a method are executed sequentially in the order in which they occur in the method. Concurrency assuming the parallel execution of creator objects and their conditional objects is restricted. Those programs in which earlier appearing conditional objects wait for the result of later appearing conditional objects will fall into deadlock (Figure 2).

3. *Sender-Subjective Transmission.*
   The sender actively delivers each message to the receiver, rather than the receiver itself trying to draw messages toward it.

4. *Implicit Argument Handling.*
   When an object is activated by the arrival of a message, the arguments of the message are already loaded on registers. Creation and setting-up of a message is embedded in message-sending instructions. There are no instructions to explicitly handle the arguments of messages.

5. *Implicit Freezing/Melting of Every Built-in Operation.*
   Any built-in instruction is implicitly frozen (converted to a message) while the destination is undefined, and melted (converted to the built-in instruction) and executed when it is determined.

6. *Incremental Garbage Collection.*
   The garbage collection function is embedded in each communication instruction.

```
class  sorry1.

   :do1(^X, ^Y, Z) ->
       :do2(X, R, ^Z)            % 'do2' is executed earlier than do3
       :do3(Y, ^R).

   :do2(^X, ^R, Z) ->
       (X > 0) /\ (R == ok) ? ( % waiting for R to be determined
         :'true  -> ok = ^Z;
         :'false -> no = ^Z;
       ).

   :do3(^Y, R) ->
       Y = ^R.                   % R is connected to Y here in 'do3'.
end.
```

Figure 1: Deadlock caused by restricted concurrency (case 1)

```
class  sorry2.

   :do(^X, ^Y, ^Z, R) ->
       (X == Q) ? (            % runnable after Q is determined.
         :'true  -> Z = ^R ;
         :'false -> Z = ^R
       ),
       (Y == ok) ? (           % Q is determined here.
         :'true  -> ok = ^Q ;
         :'false -> no = ^Q
       ).
end.
```

Figure 2: Deadlock caused by restricted concurrency (case 2)

1.5 Organization

This report is organized as follows:

Section 2 describes the system architecture, including the system resource management and the top level of the abstract $\mathcal{A}'\mathcal{UM}$ machine.

Section 3 describes the data representation: what kinds of data may reside in the memory and how they are represented.

Section 4 describes the $\mathcal{A}'\mathcal{UM}$-α instruction set, focusing on how the basic instructions are executed.

We have been implementing a software emulator of the abstract $\mathcal{A}'\mathcal{UM}$ machine, which is written in C++ and runs on the Sequent Symmetry S81 (CPU 80386, write-back cache) machine. Section 6 shows the basic performance of the latest version of the implementation.

Finally, Section 7 concludes this report, listing several problems left unsolved or under consideration, and stating our future research plans.

2 System Architecture

2.1 System Overview

A runtime environment of the abstract $\mathcal{A}'\mathcal{UM}$ machine is illustrated in Figure 3.

```
================================================================================
      ClassTable                <registers>                     ProtocolTable
     +------------+                                            +------------+
     |        *   |                                            |         *  |
     +--------|---+                                            +---------|--+
ScheduleTable | CurrentObject      CurrentClass         CurrentMessage   |
+-----------+ | +------------+     +------------+       +------------+   |
|        *  | | |         *  |     |         *  |       |         *  |   |
+--------|--+ | +---------|--+     +---------|--+       +---------|--+   |
=========|====|===========|==================|====================|======|======
 +--------+   |           |    +-------------+         +----------+      |
 | +----------+           |    |   class template    |                 |
 | |    class table       | +--+->[ HEADER          ]  |   +-------------+
 | +->[  size         ]  |  |  |  [ name            ]  |   |   protocol table
 |    [  fullness     ]  |  |  |  [ version         ]  |   +->[  size           ]
 |    [      :        ]  |  |  |  [ supers          ]  |      [        :        ]
 |    [  entry        ]-+|  |  |  [ #ins / #outs    ]  | PID*2[  arity/name     ]
 |    [      :        ] ||  |  |  [ references      ]  |      [  modes          ]
 | +---------------------+|  |  |  [ slot dict.      ]  |      [        :        ]
 | |   class information  |  |  |  [ method dict.    ]  |
 | |      table           |  |  |  [ code module     ]  |
 | +->[ HEADER        ]  |  |  |                       |
 |    [ name          ]  |  |  +-----------------+     |
 |    [ next          ]  |  |                    |     |
 |    [ class         ]--|-+ object             |     |
 |    [ subs          ]  +->[ HEADER          ] |     |    message
 |    [back references ]  |   [MSG  | first     ]-|-----+-->[MSG  | next    ]--->
 |                        |   [MSG  | last      ]-|---+      [       PID    ]
 |    schedule table      |   [OBJ  | next      ]-|-+ |    [       arg    ]
 +-->[     size       ]  |   [       class     ]-+ | |     [        :     ]
     [        :       ]  |   [       priority  ]   | |
entry[OBJ   | first   ]--+   [       global    ]   | +---> last message
 [i] [OBJ   | last    ]--+   [       creator   ]   |
     [        :       ]  |   [       slots     ]   +---> next object
                          |
                          +-> last object
================================================================================
```

Figure 3: A runtime environment of the abstract $\mathcal{A}'\mathcal{UM}$ machine

Table 1: Special Registers

Local Registers	*Content*
ScheduleTable	pointer to the schedule table
CurrentPriority	the priority of the currently executed object
MaximumPriority	the highest priority of the executable objects
CurrentObject	pointer to the currently executed object
CurrentMessage	pointer to the currently executed message
CurrentClass	pointer to the class template of the currently executed object
CurrentClassName	class identifier of the currently executed object
CurrentCode	pointer to the top of the currently executed method
InstructionPointer	pointer of the currently executed instruction
LocalHeapTop	pointer to the top of the local heap
LocalHeapBottom	pointer to the top of the local heap
FreeBlockTable	pointer to the free block table
Global Registers	*Content*
ClassTable	pointer to the class table
ProtocolTable	pointer to the message protocol table
GlobalHeapTop	pointer to the global heap
GlobalHeapBottom	pointer to the global heap

2.2 Registers

The following set of general registers and special registers, each of which is 32 bit long, are prepared.

(1) General Registers

32 registers, `R0`, `R1`, ... `R31`, hold temporary information during the execution of a method of an object.

(2) Special Registers

The following set of registers holds the current execution environment and some global information used beyond methods and objects.

2.3 Memory Management

2.3.1 Memory Space

The following memory space is assumed:

- **Byte Address:** The maximum (or implementable) memory space is 4G bytes in which a 32-bit address is given to each byte.
- **Word Access:** Memory access is done for every 32-bit word. The least significant two bits of each word are masked.
- **Word Alignment Allocation:** As for memory management, there is no constraint except that memory area is allocated with word alignment. An allocated block of memory may be of any size. The most significant 28 bits are reserved for the effective address.
- **Full Address Space:** The entire address space is accessible, that is, the whole 32-bit address is available.

2.3.2 Memory Allocation

Memory space is managed in a heap-based manner.

(1) Global Heap

All processors share a single memory space, called the *global heap*, composed of pages of fixed length. The top and bottom of the global heap are kept in registers, `GlobalHeapTop` and `GlobalHeapBottom`.

(2) Local Heap

For each processor, the memory space it can privately consume is called the *local heap*, whose top and bottom are kept in registers, `LocalHeapTop` and `LocalHeapBottom`.

1. *Initial Allocation:* At start-up time, a certain number of pages are given to each processor for its local information such as a schedule table.
2. *Dynamic Allocation:* During execution, when a processor attempts to take some area from the local heap, if the following condition holds:

$$\texttt{LocalHeapTop} + (\mathit{RequiredAmount}) > \texttt{LocalHeapBottom}$$

then a sufficient number of pages are taken from the global heap.

```
+--------------------+
| LocalHeapTop     * |
+-----------------|--+
                  |     | ////////////////// |
                  |     | ////// used ////// |
                  |     | ////////////////// |
                  |     +--------------------+
                  +---> |                    |
                        |       unused       |
                        .                    .
                  +---> |                    |
                  |     +--------------------+
+-----------------|--+
| LocalHeapBottom *  |
+--------------------+
```

2.3.3 Free Block Management

The free block table is a table to manage free (or garbage) memory blocks, listed according to size. Register `FreeBlockTable` points to this table.

```
+------------------+
| FreeBlockTable * |
+---------------|--+
                |      +--+----+------------+
                +----->|RC|FTB | block size |
                       +--+----+------------+
                       | entry[1]      *------->
                       +--------------------+
                       |                    |
                       .                    .
                       +--------------------+
                       | entry[i]      *------->
                       +--------------------+
                       .                    .
                       |                    |
                       +--------------------+
                       | entry[m]      *------->
                       +--------+-----------+
                       | others | first  *------>
                       |        +-----------+
                       |        | last   *------>
                       |        +-----------+
                       |        | min size  |
                       |        +-----------+
                       |        | max size  |
                       +--------+-----------+
```

where `entry[n]` points the first one of 2^i word free blocks
`others` keep a list of free blocks longer than 2^m words
`first` points to the first element
`last` points to the last element
`min` the minimum of the listed free blocks sizes
`min` the maximum of the listed free blocks sizes

2.4 Scheduling Control

2.4.1 Schedule Table

Executable objects make a queue according to their priority. A *schedule table* is a table to manage this queue. Register `ScheduleTable` points to this table.

```
+-----------------+
| ScheduleTable * |
+---------------|-+
                |      +--+----+------------+
                +----->|RC|STB | block size |
                       +--+----+------------+
                       | entry[0]           |
                       +--------------------+
                       |                    |
                       -                    -

                       +----------+---------+
                       |          | first *------>
                       | entry[i] +---------+
                       |          | last  *----->
                       +----------+---------+
                       -                    -
                       |                    |
                       +--------------------+
                       | entry[MAXOPR]      |
                       +--------------------+
```

where each entry, `entry[i]`, consists of two fields:

`first` the first one of the executable objects with priority i.

`last` the `next` field of the last one of the executable objects with priority i, where the next object will be placed.

2.4.2 Priorities

The following two kinds of priority control are provided:

- Object priority
- Message priority

2.4.3 Object Priority

Object priority decides how an object will be scheduled, and is of the range:

$$0 \leq P_o \leq \texttt{MAXOPR}$$

Executable objects are put in an appropriate queue in the schedule table according to their priority. The higher the priority an object has, the earlier it is scheduled. An object priority is given to an object when it is created.

2.4.4 Message Priority

Message priority P_m is a priority given to a message:

$$P_m = \begin{cases} 0 & \text{for normal messages} \\ 1 & \text{for express messages} \end{cases}$$

The message priorities are identified by message tags.

(1) Normal Message

When a normal message is sent to an object or a joint, the message is enqueued *at the end* of the message queue of the object or joint, that is, the `last` field of the object or joint points to this message.

(2) Express Message

When an express message is sent to an object or a joint, the message is enqueued *at the beginning* of the message queue of the object or joint, that is, the `first` field of the object/joint points to this message.

2.5 $\mathcal{A}'\mathcal{UM}$-α Interpreter

An $\mathcal{A}'\mathcal{UM}$ program is translated into a sequence of $\mathcal{A}'\mathcal{UM}$-α instructions. The $\mathcal{A}'\mathcal{UM}$-α interpreter, that is the top level of the abstract $\mathcal{A}'\mathcal{UM}$ machine, interprets and executes the sequence of the $\mathcal{A}'\mathcal{UM}$-α instructions according to the following algorithm.

Algorithm 2.1 ($\mathcal{A}'\mathcal{UM}$-α **Interpreter)**

Step0 (Initialization)
Make the maximum priority the *current priority*.

Loop1 (Top Loop)

Loop2 (Execute Objects with the Current Priority)

Step21 (Set Current Object)
Try to take one object from the object queue with the current priority.
If there is an object, then make it the *current object*;
otherwise, exit Loop1.

Step22 (Set Current Class)
If the class of the current object differs from that of the last executed object, then make it the *current class*.

Loop3 (Execute Messages of the Current Object)

Step31 (Set Current Message)
Try to take the first message from the message queue of the current object.
If there is a message, then make it the *current message*;
otherwise, exit Loop2.

Step32 (Detect Closing)
If the interface stream is already closed, then regard the closing (`NIL`) as the current message.

Step33 (Search for Method)
Search for a method corresponding to the current message and execute it.

Step34 (Check Maximum Priority)
If the maximum priority is higher than the current priority,

- make the maximum priority the current priority, and
- if there is one or more message enqueued to the current object or if the message queue is closed, then put the current object back on the schedule table.

(end of Loop3)

(end of Loop2)

Step11 (Find Next Highest Priority)
Find the next highest priority after the current maximum priority.

(end of Loop1)

□

3 Data Representation

In designing the data representation described in this section, the following considerations have been made:

(1) Meanings and Representations

1. *Meanings.*

   There are two kinds of entities to represent:

   - objects/joints
   - messages

   They are exclusive: objects/joints do not appear where message do and vice versa. For instance, as will be mentioned later, objects may be put in the schedule table or pointed from the `destination` field of joints, but messages may not; messages may be put in the message queue of objects, but objects may not. Those which are exclusive can share the same representation for different meanings.

2. *Representations.*

   Among the entities,

   - some can be represented in a single word, and
   - others make a structure.

   The former is called a *constant entity* and the latter a *structured entity*. A structured entity can be represented as a pair of a *structure* and a *pointer* to it.

(2) Structure Tags

For a structured entity, there are two ways of putting tags:

1. *Pointer Tag Method:* to put a tag in a pointer.

```
+-----+---------+        +--------------+
| Tag |     *---------> |              |
+-----+---------+        |              |
                         |              |
                         +--------------+
```

2. *Object Tag Method:* to put a tag in a structure.

```
+---------------+        +-----+---------+
|               *------->| Tag |         |
+---------------+        +-----+---------+
                         |               |
                         +---------------+
```

The pointer tag method is faster by one reference in detecting the type of the pointed structure, though the tag size is more limited.

Placing more importance on quick access, we have adopted the pointer tag method.

3.1 Cells

A *cell* is a word consisting of a *value part* and a *tag part*, both of flexible length.

```
+-----------------------------+-------+
| value                       | tag   |
+-----------------------------+-------+
32                            n       0
```

where **value** constant data, address (pointer)
tag object type, message type, etc.

3.2 Pointers

Those cells whose least significant two bits are either 01, 10 or 11 are *pointers*. Taking into account the byte address, word access property of memory space, the least significant two bits are used to represent pointers. There are pointers to object structures and pointers to message structures.

```
+----------------------------------+--+
| pointer                          |XX|   XX = 01, 10, 11
+----------------------------------+--+
32                                 2  0
```

3.2.1 Object Pointers

There are three kinds of pointers to objects:

1. pointers to merge joints
2. pointers to general objects
3. pointers to built-in structured objects

```
                                            mnemonic
+---------------------------------+--+
| merge joint                     |01|     [MJNT |  *  ]
+---------------------------------+--+

+---------------------------------+--+
| object                          |10|     [OBJ  |  *  ]
+---------------------------------+--+

+---------------------------------+--+
| built-in object                 |11|     [BOBJ |  *  ]
+---------------------------------+--+
32                                2  0
```

Note that it is assumed that append joints will not be used often, thus they are implemented as built-in objects.

3.2.2 Message Pointers

There are two kinds of messages according to their priority:

1. normal messages
2. express messages

A message is identified as normal or express by the tag of a pointer to it, so it can be recognized quickly without accessing the message itself.

```
                                            mnemonic
+---------------------------------+--+
| normal message                  |11|     [MSG  |  *  ]
+---------------------------------+--+

+---------------------------------+--+
| express message                 |10|     [MSGX |  *  ]
+---------------------------------+--+
32                                2  0
```

3.3 Constants

Those cells with 00 in their least significant two bits are *constants*.

```
+-------------------------+-----+--+
| constant data           |m-tag|00|
+-------------------------+-----+--+
32                        n     2  0
```

where `m-tag` is a minor tag to categorize constant types.

(1) Integer

If the minor tag is 0, it represents an integer object. The data part `i` is an integer which is represented in a two's complement method using 28 bits, that is, $-2^{28} \leq i \leq 2^{28} - 1$.

```
                                                  mnemonic
+--------------------------------+---+
| integer                        |000|            [INT  |  i  ]
+--------------------------------+---+
32                               3   0
```

(2) Single-Precision Floating-Point Number

If the minor tag is `11`, it represents a single-precision floating-point number. The data part `f` is a hexadecimal normalized floating number with a 21-bit mantissa (1 bit for the sign and 20 bits for the absolute value) and a 7-bit exponent.

```
                                                  mnemonic
+-------------------------------+----+
| short floating                |1100|            [SFLT |  f  ]
+-------------------------------+----+
32                              4    0
```

(3) Atom

If the minor tag is `001`, it represents an atom (a symbol). The data part `a` is an atom number (a symbol identifier) which is zero or a positive integer represented in 27 bits, that is, $0 \leq a \leq 2^{27} - 1$, where `[ATOM | 0 ]` is reserved for `NIL`.

```
                                                  mnemonic
+------------------------------+-----+
| atom                         |00100|            [ATOM |  a  ]
+------------------------------+-----+
32                             5     0
```

(4) Bool

If the minor tag is `0101`, it represents a boolean object, either a `true` object or a `false` object.

1. `[BOOL | 0 ]` represents a **`true`** object.
2. `[BOOL | 1 ]` represents a **`false`** object.

```
                                                  mnemonic
+-----------------------------+------+
| bool                        |010100|            [BOOL |  b  ]
+-----------------------------+------+
32                            6      0
```

(5) System Objects

If the minor tag is `1101`, it represents a system object.

There are three kinds of system objects:

1. `[SOBJ | 0 ]` represents a *sink object* which works for garbage collection.

2. [SOBJ | 1] represents an *initial outlet*. When a message is sent to an initial outlet, it is collected as garbage or an error is raised.

3. [SOBJ | 2] represents an *initial inlet*. When an initial inlet is referred to, an error is raised.

```
                                              mnemonic
+-----------------------------+-------+
| system object               |110100|      [SOBJ |  s  ]
+-----------------------------+-------+
32                             6      0
```

3.4 Structures

As introduced in Section 3.2, there are two kinds of structures pointed by pointer cells:

1. objects

2. messages

3.4.1 Object Header

The following *header* is provided in the first cell of the structure for objects (including merge joints, general objects and built-in objects and class templates).

```
                                              mnemonic
+--------+--------+----------------+
| B-tag  | LOCK   |  RC            |       [ HEADER  ]
+--------+--------+----------------+
32       24       16               0
```

where B-tag built-in object tag to detail built-in types, or 0 for those other than built-in objects.
LOCK lock field prepared for parallel implementation
RC reference count

(1) Lock Field for Parallel Implementation

This system is assumed to be extended for parallel implementations using multiprocessors. In the parallel implementation, more than one processor may access the same object. To control mutual exclusion between processors, the lock/unlock mechanism is assumed; the LOCK field is prepared for this purpose.

(2) Reference Count Management for Stream Merging

The abstract $\mathcal{A}'\mathcal{UM}$ machine adopts the following reference count method for both stream-merging and garbage collection:

1. *Initiation.* At creation of a general object or a merge joint, its reference count is set as follows:

$$\mathtt{RC} = \begin{cases} 1 & \text{for a general object} \\ 2 & \text{for a merge joint} \end{cases}$$

2. *Merging.* When merging a stream, the reference count of its destination object or merge joint is incremented.

3. *Closing.* When closing a stream, the reference count of its destination object or merge joint is decremented. When the reference count of an object or a merge joint becomes `0`, that is, `RC = 0`, the object or merge joint may be collected as garbage.

3.4.2 General Objects

General objects, including external, immutable volatile and mutable volatile objects, are those of user-defined classes. They are concurrent computation units to be scheduled. When they are executable, they are enqueued in the schedule table.

```
[OBJ  |  *  ] -----> [    0|LOCK| RC    ]
                     [MSG  | first      ] -----> first message
                     [       last       ]
                     [OBJ  | next       ] -----> next object
                     [       class      ] -----> class template
                     [       priority   ]
                     [       global     ]
                     [       creator    ]
                     [       slots      ] -----> slot table
```

where

1. Communication Part

`first`	pointer to the first message of the message queue
`last`	pointer to either the **`first`** message when no message is coming, or the **`next`** field of the last message

2. Scheduling Part

`next`	pointer to the next object enqueued with the same priority
`class`	pointer to its class template
`priority`	object priority which is a positive number. Enqueued with this priority in the schedule table

3. Slots

`global`	a global object or joint
`creator`	its creator object if it is a volatile object
`slots`	pointer to a slot vector

3.4.3 Built-in Structured Objects

The following built-in structured objects are provided:

1. strings
2. lists
3. vectors
4. double precision floating numbers
5. class objects
6. message objects

(1) String

```
[BOBJ | * ] -----> [*STR|LOCK| RC ]
                   [    size      ]
                   [    element   ]
                         :
                   [     :        ]
```

where `*STR` = `BSTR` bit string
`CSTR` byte string
`WSTR` double byte string
`size` number of elements
`element` packed representation of the string content

(2) List

```
[BOBJ | * ] -----> [VECT|LOCK| RC ]
                   [    car       ]
                   [    cdr       ]
```

where **car** an object or a merger for the car part
cdr an object or a merger for the cdr part

(3) Vector

```
[BOBJ | * ] -----> [VECT|LOCK| RC ]
                   [    size      ]
                   [    element   ]
                         :
                   [     :        ]
```

where **size** number of elements
element an object or a merger for each element

(4) Double Precision Floating Number

```
[BOBJ |  *  ] -----> [BFLT|LOCK|  RC  ]
                     [       word1    ]
                     [       word2    ]
```

where `word1,2` packed floating number

(5) Class Object

A class object exists only when it must be an object, such as when it is passed as an argument of a message to an object.

```
[BOBJ |  *  ] -----> [CLS |LOCK|  RC  ]
                     [       class    ] -----> class template
```

where `class` pointer to a class template

(6) Message Object

A *message object* is created only when it has to be an object, such as when a default message is specified in a method.

```
[BOBJ |  *  ] -----> [MSG |LOCK|  RC  ]
                     [    message     ] -----> message
```

where `message` pointer to a message

3.4.4 Joints

A joint is a relay to hold messages whose destination is not determined yet. As soon as the destination is determined to be an object or another joint, these buffered messages are forwarded to the destination object or joint.

There are two kinds of joints:

- merge joints (or mergers)
- append joints (or appenders)

(1) Mergers

A merger accepts messages from two inlets and forwards them in a nondeterministic order to the destination.

```
[MJNT |  *  ] -----> [    0|LOCK|  RC       ]
                     [       destination ] -----> object/merger
                     [MSG   | first        ] -----> first message
                     [       last          ]
```

where	RC	reference count
	destination	pointer to a destination object/merger
	first	pointer to first message of those buffered, or 0 at initiation
	last	pointer to the **next** field of the last message, or the above **first** field at initiation

(2) Appenders

An appender accepts messages from two incoming streams and forwards them to the destination. All messages from the first incoming stream are forwarded before any message from the second incoming stream.

An appender is implemented as a built-in object, for the sake of tag capacity, which takes the first inlet as its interface stream and holds the second inlet and the destination. When the first incoming stream is closed, it connects the `second stream` to the `destination`.

```
[BOBJ |  *  ] -----> [AJNT|LOCK|  RC      ]
                     [    destination    ] -----> object/merger
                     [       first        ] -----> first message
                     [       last         ]
                     [   second stream   ] -----> merger
```

where	**destination**	pointer to a destination object/merger.
	first	pointer to first message of those buffered, or 0 at initiation.
	last	pointer to the **next** field of the last message, or the above **first** field at initiation.
	second stream	pointer to a merger for the second incoming stream.

3.5 Messages

According to their structural differences, there are two kinds of messages:

- atomic messages
- compound messages

According to their priority, there are two kinds of messages:

- normal messages with tag MSG
- express messages with tag MSGX

(1) Atomic Messages

Atomic messages are those which contain no arguments, such as integer, atom, bool, vector messages. They are also registered in the message protocol table.

```
[MSG* |  *  ] -----> [MSG  | next ] -----> next message
                     [    PID     ]
                     [    value   ]
```

where	next	when a message is enqueued in the message queue of an object or a merger, it points to the next message.
	PID	message protocol identifier given to an atomic message
	value	constant value representing an atomic message

(2) Compound Messages

Compound messages are those which contain a message name and arguments.

```
[MSG* |  *  ] -----> [MSG  | next ] -----> to the next message
                     [    PID     ]
                     [    arg     ]
                          :
                     [    :       ]
```

where	next	when the message is enqueued in the message queue of an object or a merger, it points to the next message.
	PID	message protocol identifier which is an offset in the message protocol table.
	arg	arguments of the message

3.6 Classes

A class object has a class template as its value, which contains inheritance information, slot information and method information.

A built-in class object is created only when a class has to exist as an object, for instance, when an object creates a class object and passes a stream to the class object as an argument of a message to another object. Otherwise, a class template is directly pointed to from the `class` field of each instance.

3.6.1 Class Table

The class table is a hash table to search a class template from a class name. The open-hash method is used for searching; when the number of registrations gets larger than twice of the hash table size, then a new table of double size is created and reconfigured.

```
[ClassTable] -----> [    size     ]
                    [  fullness   ]
                          :
                    [ class chain ] -----> class information chain
                          :
                    [     :       ]
```

where	`size`	the table size
	`fullness`	the number of registered class templates, which is used as a guide for reconfiguration
	`class chain`	pointer to the first class information of a chain of class information

3.6.2 Class Information

A *class information* keeps the information which is necessary for updating a class but not for execution, as follows:

- *List of subclasses which inherit this class.*
  When a class is relinked, all the classes which inherit this class must be updated. As long as there is an instance of these subclasses, it might traverse its inheritance list to reach this class. To keep them running correctly, the current version of these subclasses should be kept without destruction. Hence, when a parent class is relinked, its children classes are copied, so that the execution of the new version does not interfere that of the old version.

- *List of reference classes which refer to this class.*
  When a class is relinked, all the classes which refer to this class must be copied for the same reason as the above.

```
[ClassTable]
    |       class table
    +----> [     :     ]
           [ class chain] -----> [    HEADER      ]
           [     :     ]         [     name       ]
                                 [     next       ] -----> next class information
                                 [    class       ] -----> class template
                                 [     subs       ] -----> subclass table
                                 [back references] -----> back-reference class table
```

where	`name`	class name (atom)
	`next`	pointer to the next entry of a chain of class information whose class names have the same hashed value. (0 for the last entry)
	`class`	pointer to a class template
	`subs`	pointer to a subclass table
	`back references`	pointer to a back-reference table

(1) Subclass Table (subs)

A *subclass table* is a table of sub classes which inherit this class. When this class is relinked, the contents of this list is copied.

```
[  subs   ] -----> [   size      ]
                        :
                   [   class     ] -----> class template
                        :
                   [     :       ]
```

where	`size`	table size
	`class`	pointer to a class template of each subclass which inherits this class

(2) Back Reference Class Table (back references)

A *back reference class table* is a table of classes which refer to this class. When this class is relinked, the contents of this list is also copied.

```
[back references] -----> [   size     ]
                              :
                         [   class    ] -----> class template
                              :
                         [     :      ]
```

where	`size`	table size
	`class`	pointer to a class template of each reference class which refers to this class

3.6.3 Class Templates

A *class template* contains only the class inheritance information, slot information, and method information which is defined in the class. Slots and methods are retrieved by traversing the inheritance tree; for quick access, slot caches and methods caches are provided.

```
[  class  ] -----> [   HEADER     ]
                   [ATOM|  name   ]
                   [   version    ]
                   [   supers     ] -----> super class table
                   [ #ins/#outs   ]
                   [  references  ] -----> reference class table
                   [ slot dict    ] -----> slot dictionary
                   [ method dict  ] -----> method dictionary
                   [ code module  ] -----> code module
```

where	`name`	class name (atom)
	`version`	version number of the class template (integer)
	`supers`	pointer to a super class table
	`#ins/#outs`	the number of inlets and the number of outlets, both of which are directly defined in this class
	`references`	pointer to a reference class table
	`slot dict`	pointer to a slot dictionary
	`method dict`	pointer to a method dictionary
	`code module`	pointer to a code module

(1) Super Class Table (supers)

A *super class table* is a table of class templates and slot bases of the super classes that this class inherits.

```
[ supers ] -----> [   size    ]
                  [ slot size ]
                       :
                  [   class   ] -----> class template
                  [   BASE    ]
                       :
                  [    :      ]
```

where	`size`	table size
	`slot size`	the number of slots defined in this class and its supers. This information is used when a slot vector is created.
	`class`	pointer to the class template of a super class which this class inherits.
	`BASE`	a slot base position in the slot table, which is an offset where the first slot of each super class is stored. Any slot is accessed with the base of its own class and the offset given to the slot within the class.

(2) Reference Class Table (references)

A *reference class table* is a table of all classes which refer to this class. When this class is relinked, the content of this list is copied, too.

```
[references] -----> [   size    ]
                         :
                    [   class   ] -----> class template
                         :
                    [    :      ]
```

where **size** table size
class pointer to the class template of a class to which this class refers

3.6.4 Slot Information

(1) Slot Dictionary

A slot dictionary is a hash table to obtain a slot offset in the slot table, using a slot name as a key.

```
  [ slot dict ] -----> [      size      ]
                                :
hash(name, size)*2 ==>[   slot name     ]
                       [     SENT       ]
                                :
                       [        :       ]
```

where **slot name** slot name
SENT slot offset from the base

(2) Slot Table (slots)

A slot table is a table of all the inlet and outlet slots that are defined for an object in its own class and super classes. For each slot, an offset (SENT) in its own class is uniquely given; for each super class, a base (BASE) in the slot table, which is where the first one of its slots is stored, is uniquely given. Hence, a slot is accessed with the base and offset at the position of BASE + SENT.

```
[  slots ] -----> [   size    ]
               |         :
               |  [      :    ]
               V  ----------------- slots for class x
      base(BASE) [      :    ]
               |         :
               V         :
   offset(SENT) [   slot    ] -----> class template
                         :
                  -----------------
                  [      :    ]
                         :
                  [      :    ]
```

where **size** table size
slot either an inlet or an outlet. At creation, each inlet is set to an *initial inlet*; each outlet is set to an *initial outlet*. At termination, each inlet is connected to a sink object; each outlet is closed.

3.6.5 Method Information

Method code defined in a class is grouped together to be a *code module*. Each method is identified with a *message protocol identifier* which is a unique number determined by the message name, arity and modes. Each method code entry is at an offset in the code module, which is determined from the message protocol identifier.

(1) Method Dictionary

A *method dictionary* is a hash table to obtain a method code offset in a code module, using a message protocol identifier, PID.

```
[ method dict ] -----> [      size     ]
                               :
 hash(PID, size)*2 ==> [      PID      ]
                       [      MENT     ]
                               :
                       [       :       ]
```

where PID message protocol identifier
MENT method entry offset in a code module

(2) Message Protocol Table

A *message protocol table* is a global table which keeps the message name, arity and modes (argument directions) of each message with its message protocol identifier, PID.

A message protocol identifier PID is an offset from the top of this table.

```
[ProtocolTable] -----> [      size        ]
                              :
         PID * 2 ==>[  name / arity  ]
                    [  modes         ]
                              :
                    [         :      ]
```

where name message name
arity number of arguments
modes a bit array whose every bit represents the stream direction of an argument

(3) Code Module

A *code module* is a block of method code which is defined in a class. For each method, its entry offset MENT is registered in the method dictionary.

```
[ code module ] -----> [      size      ]
                |      [                ]
                v             :
           MENT ==>[ method code  ]
                   [              ]
                          :
                   [              ]
```

4 $\mathcal{A}'\mathcal{UM}$-α Instruction Set

The $\mathcal{A}'\mathcal{UM}$-α instructions are categorized according to their derivations as follows:

1. Send/Close Instructions
   (a) General Send and Close Instructions (Table 2)
   (b) Built-in Function Instructions (Table 4)
   (c) Optimized Conditional Instructions (Table 5)
   (d) Slot Access Instructions (Table 8)
2. Connect Instructions (Table 3)
3. Create Instructions (Table 7)
4. Descend and Other Instructions (Table 6)

In addition, there are pseudo instructions which are issued to the assembler and linker, not the machine itself, as listed in Table 9.

We describe below the outline of the $\mathcal{A}'\mathcal{UM}$-α instructions: the execution mechanism of the abstract $\mathcal{A}'\mathcal{UM}$ machine.

4.1 Instruction Format

Each $\mathcal{A}'\mathcal{UM}$-α instruction is of flexible length, with any number of operands.

```
+---------+---------------------//-------------+
|OP code  | Operands                           |
+---------+---------------------//-------------+
```

where each operand may be one of the following:

`Ri`	general register
`Vx`	immediate value (constant)
`Lx`	immediate value (label)

Table 2: $\mathcal{A}'\mathcal{UM}$-α Instructions (general send & close)

Op code	*operands*	*function*
send instructions		
`send`	`Robj, Vpid, R0, ..., Rn`	creates a message with `Vpid` as its PID and `R0` to `Rn` as its arguments, and sends it to the destination, `Ri`, in *normal mode*: the message is appended at the end of the message queue of the destination.
`send_express`	`Robj, Vpid, R0, ..., Rn`	creates a message with `Vpid` as its PID and `R0` to `Rn` as its arguments, and sends it to a stream, `Ri`, in *express mode*: the message is added to the beginning of the message queue of the destination.
`send_self`	`Vpid, R0, ..., Rn`	creates a message with `Vpid` as its PID and `R0` to `Rn` as its arguments, and sends it to the current object (*self*). The message is prepended at the beginning of normal messages of the message queue of the current object.
close instructions		
`close`	`Ri`	closes stream `Ri`. If `Ri` points to an object or a joint, this instruction decrements the reference count of `Ri` by `1`.

Table 3: $\mathcal{A}'\mathcal{UM}$-α Instructions (connect & split)

Op code	*operands*	*function*
connect instructions		
`connect`	`Rout, Rin`	connects the inlet, `Rin`, of a joint to the outlet, `Rout`, of an object or a joint. When `Rout` points to either an object, a volatile object or a joint, the messages buffered in `Rin` are forwarded to `Rout`. When `Rout` points to a built-in object, the messages buffered in `Rin` are immediately executed.
`split`	`Ri`	increments the reference count (`RC`) of `Ri` by `1`
`split_n`	`Ri, Vn`	increments the reference count (`RC`) of `Ri` by `Vn`

Table 4: $\mathcal{A}'\mathcal{UM}$-α Instructions (built-in operations)

Op code	*operands*	*function*
arithmetic/logical operations		
`minus`	`Robj, Rres`	takes the 2's complement of a number `Robj`; `Rres` holds the outlet of a stream toward the resulting number (common to each below).
`add`	`Robj1, Robj2, Rres`	adds number `Robj2` to number `Robj1`.
`sub`	`Robj1, Robj2, Rres`	subtracts number `Robj2` from number `Robj1`.
`mul`	`Robj1, Robj2, Rres`	multiplies number `Robj1` by number `Robj2`.
`div`	`Robj1, Robj2, Rres`	divides number `Robj1` by number `Robj2`.
`mod`	`Robj1, Robj2, Rres`	divides number `Robj1` by number `Robj2`, and takes the residue.
`shtr`	`Robj1, Robj2, Rres`	shifts number `Robj1` to the right by the number of bits `Robj2`.
`shtl`	`Robj1, Robj2, Rres`	shift number `Robj1` to the left by the number of bits `Robj2`.
`not`	`Robj, Rres`	negates `Robj`.
`and`	`Robj1, Robj2, Rres`	takes conjunction of `Robj1` and `Robj2`.
`or`	`Robj1, Robj2, Rres`	takes disjunction of `Robj1` and `Robj2`.
`xor`	`Robj1, Robj2, Rres`	takes exclusive disjunction of `Robj1` and `Robj2`.
`eq`	`Robj1, Robj2, Rres`	tests whether the value of `Robj1` is equal to that of `Robj2`; if so, `Rres` holds a `true` object; otherwise, a `false` object (common to each below).
`neq`	`Robj1, Robj2, Rres`	tests whether the value of `Robj1` is not equal to that of `Robj2`.
`lt`	`Robj1, Robj2, Rres`	tests whether the value of `Robj1` is less than that of `Robj2`.
`gt`	`Robj, Robj2, Rres`	tests whether the value of `Robj1` is greater than that of `Robj2`.
(many others)		
list operations		
`car`	`Rlist, Rres`	refers to the car of a list `Rlist`.
`cdr`	`Rlist, Rres`	refers to the cdr of a list `Rlist`.
universal built-in operations		
`class`	`Robj, Rres`	takes the class of object `Robj`; `Rres` holds the class
`who`	`Robj, Rres`	asks `Robj` "who are you"; `Rres` holds the inlet of the incoming stream.

Table 5: $\mathcal{A}'\mathcal{UM}$-α Instructions (conditional)

Op code	*operands*	*function*
optimized conditional instructions		
`if_equal`	`Robj1, Robj2, Ltrue, Lfalse`	When it is already know that `Robj1` is equal to `Robj2`, jump to `Ltrue`; otherwise jump to `Lfalse`.
`if_lt`	`Robj1, Robj2, Ltrue, Lfalse`	When it is already known that `Robj1` is less than `Robj2`, jump to `Ltrue`; otherwise jump to `Lfalse`.
`if_gt`	`Robj1, Robj2, Ltrue, Lfalse`	When it is already known that `Robj1` is greater than `Robj2`, jump to `Ltrue`; otherwise jump to `Lfalse`
`if_true`	`Robj, Ltrue, Lfalse`	When it is already known that `Robj` is true, jump to `Ltrue`; otherwise jump to `Lfalse`.

`branch_on_who`
`entry`
⋮
`entry`
	`Robj, Vno`
	`V0, L0`
	
	`Vn, Ln`
		When `Robj` is already determined to be an integer or atom, jump to one of the labels, `Li` corresponding to the content of `Robj`, where `Vno` is the number of entries.

Table 6: $\mathcal{A}'\mathcal{UM}$ α Instructions (descend/control)

Op code	*operands*	*function*
descend instructions		
`descend`		(end of method) enters a new scheduling cycle (returns control to the top-level interpreter).
`terminate`		(end of object) completes all the slots, releases the object, and returns control to the top-level interpreter. This instruction is issued when the internal termination message is sent to the object.
other instructions		
`jump`	`Vlabel`	jumps to the label `Vlabel`
`wait`		suspends the execution of the current object. With this instruction, the object becomes dormant; it is awakened when the activate instruction is executed.
`activate`	`Robj`	puts the object, `Robj`, on the schedule table. This instruction is used when a volatile object resumes the execution of its creator object.
`move`	`Ri, Rj`	copies the content of `Ri` to `Rj`

Table 7: $\mathcal{A}'\mathcal{UM}$-α Instructions (create)

Op code	*operands*	*function*
primitive/built-in object creation instructions		
`create_integer`	`Robj, Vinteger`	loads an integer `Vinteger` in `Robj`
`create_atom`	`Robj, Vatom`	loads an atom `Vatom` in `Robj`
`create_bool`	`Robj, Vbool`	loads a boolean `Vbool` in `Robj`
`create_sfloat`	`Robj, Vfloat`	loads a single-precision floating number `Vfloat` in `Robj`
`create_sink`	`Robj`	loads a sink cell in `Robj`
`create_error`	`Robj`	loads an error cell (an initial outlet) in `Robj`
`create_string`	`Robj, Vbits, Vno, Vstring`	creates a string with the content, `Vstring`, which consists of number `Rno` of elements of `Rbits` bits; put the pointer to it in `Robj`
`create_dfloat`	`Robj, Vfloat1, Vfloat2`	creates a double-precisioned floating number; puts the pointer to it in `Robj`
`create_list`	`Robj, Rcar, Rcdr`	creates a list whose car holds `Rcar` and cdr holds `Rcdr`, puts the pointer to it in `Robj`
`create vector`	`Robj, Vsize`	creates a vector of size `Vsize`; puts the pointer to it in `Robj`
`create_class`	`Robj, Vclass`	retrieves a class template for class name `Vclass` from the class table; puts the pointer to it in `Robj`
general object creation instructions		
`create_instance`	`Robj, Vclass`	creates an instance of a class with class name, `Vclass`; puts the pointer to it in `Robj`
`create_instance_of`	`Robj, Rclstmp`	creates an instance of class template designated by `Rclstmp`; puts the pointer to it in `Robj`
volatile object creation instruction		
`create_volatile`	`Robj, Rmjnt, Vclass`	creates a volatile object of class `Vclass`; puts the pointer to it in `Robj`. Every volatile object has two slots: one is an outlet slot toward its creator object; the other is an inlet slot. When the volatile object receives a message, it creates a built-in object representing the received message, connects the inlet slot to that object, then activates its creator object.
joint creation instructions		
`create_mjoint`	`Rmjnt`	creates a merge joint; puts the pointer to it in `Rmjnt`
`create_ajoint`	`Rajnt, Rmjnt, Rdst`	creates an append joint with `Rmjnt` as the second stream and `Rdst` as the destination; puts the pointer to it in `Rajnt`

Table 8: $\mathcal{A}'\mathcal{UM}$-α Instructions (slot access)

Op code	*operands*	*function*
slot access instructions		
get_inlet	Ri, Vclass!Voffset	retrieves the content of an inlet slot at Voffset, in Ri. By this instruction, the inlet slot is initialized to be an *error* state, so that no more than one continuous retrieval is possible.
set_inlet	Ri, Vclass!Voffset	updates an inlet slot Ri at Voffset with Ri.
get_outlet	Ri, Vclass!Voffset	retrieves the content of an outlet slot at Voffset, in Ri. This instruction issues the split instruction to the outlet slot.
set_outlet	Ri, Vclass!Voffset	updates an outlet slot at Voffset with Ri.
get_inlet_by_name	Ri, Vclass, Vname	retrieves the content of an inlet slot named Vname, in Ri. By this instruction, the inlet slot is initialized to be an *error* state, so that no more than one continuous retrieval is possible.
set_inlet_by_name	Ri, Vclass, Vname	updates an inlet slot Ri named Vname with Ri.
get_outlet_by_name	Ri, Vclass, Vname	retrieves the content of an outlet slot named Vname, in Ri. This instruction issues the split instruction to the outlet slot.
set_outlet_by_name	Ri, Vclass, Vname	updates an outlet slot named Vname with Ri.

Table 9: $\mathcal{A}'\mathcal{UM}$-α Pseudo Instructions

pseudo op	*arguments*	*meaning*
.class	ClassName	declares the beginning of class definition.
.super	SuperClassName	defines a direct super class name to be inherited.
.clsref	ReferredClassName	notifies a class name which is referred in the class.
.outlet	OutletSlotName	defines a outlet slot name.
.inlet	InletSlotName	defines an inlet slot name.
.pid	PIDlabel, PrintName, #args, mode	defines a protocol identifier.
.method	MethodName, PrintName, #args, mode	declares the beginning of a method.
.end	ClassName	declares the end of class definition.

4.2 Stream Manipulation

Merge joints play the central role of stream manipulation.

- **Execution Based on Merge Joints:** Merge joints work as relays. Messages which are sent to a merge joint are buffered until the destination of the merge joint is determined. The merge joint function is also embedded in objects.

- **Creation of Merge Joint:** A merge joint is created at the first occurrence of a channel variable. For later occurrence, the split instruction is issued instead of the merge joint creation instruction. This instruction increments the reference count of the merge joint or object.

- **Sequencing of Messages:** Sequencing of messages (production of a stream) is performed by issuing a sequence of send instructions to the same merge joint. The order of messages to the same merge joint follow the order of send instructions.

- **Forwarding of Messages:** Those messages buffered in a merge joint are forwarded when the connect instruction is issued to the merge joint.

- **Incremental Garbage Collection:** Incremental garbage collection is embedded in the send, close and connect instructions. It is performed based on the reference count scheme. The reference count is incremented by the split and connect instructions; it is decremented by the close and connect instructions.

  When the close instruction is issued to an object or a merge joint, its reference count is decremented. When its reference count reaches 0, the object or joint is scavenged. If the reference count reaches 0, the reference count of its destination is decremented; so the closing is propagated as far as possible.

  When the connect instruction is issued, the reference count of the source joint is decremented, while the reference count of the destination joint or object is incremented.

- **Append Joints as Built-in Objects:** Append joints are created as built-in objects.

4.3 Message Sending

- **Message Identification by PID:** Each message is given a protocol identifier (PID) from its message name and the number and mode of its terminal arguments. The closing is also given a special PID during the method search phase.

- **Handling Normal/Express Messages:** According to their priorities, messages are divided into two: normal messages and express messages. When a normal message is sent to a merge joint or object, it is appended at the end of the message queue of the joint or object. When an express message is sent to a merge joint or object, it is added at the beginning of the message queue of the joint or object.

4.4 Built-in Operations

A variety of built-in operations are provided.

- **Binary and Unary Operations:** According to the number of arguments, there are binary operations, such as `add`, and unary operations, such as `minus`.

- **Commutative and Non-Commutative Operations:** According to whether it is commutative or not, binary operations are divided into two: commutative operations, such as `add`, and non-commutative operations, such as `sub`.

- **Freezing and Melting of Built-in Operations:** The most characteristic feature of the $\mathcal{A}'\mathcal{UM}$ built-in operations, compared to those of other concurrent object-oriented languages, is that the *freezing* and *melting* of built-in operations is required in their execution, since operands may be *undefined* when built-in operations are issued.

  Step1. When a built-in operation is executed, if the operands are already determined, the built-in operation is immediately executed, producing the result.

  Step2. If the first operand is not determined yet (if it is a joint whose destination is undefined), the built-in operation is *frozen* to be a message with the same name as Op-code, and is sent to the joint.

  Step3. When the joint is connected finally to an object, the built-in operation message is *melted* to be a built-in operation, and executed against the destination object.

  Step4. When the second operand is undefined, the built-in operation is frozen again and sent to the second operand, though there is a difference in the treatment of the built-in operation. If it is a commutative operation (`add/+-` for example), the same message (`add/+-`) as the above one is sent. If it is a non-commutative operation (`sub/+-`), a message which represents a reverse function (`rev_sub/+-`) is sent.

4.5 Optimized Condition Handling

Optimized conditional instructions are provided.

- **When Condition is Determined, No Volatile Object is Created:** When the conditions are already determined, no volatile object is created. Like conventional conditioning, such as `if` or `switch`, the current object only jumps to an appropriate label.

- **Even When Created, Volatile Object is a Synchronizer:**
  Step1. When the condition is not determined (for example, either `X` or `Y` is not determined in `X > Y`), an object creates a volatile object, and passes two pointers to the volatile object: one is a pointer (A) to itself, and the other is a pointer (B) to a result. Then the creator object becomes dormant.

  Step2. The created volatile object works as a synchronizer. A volatile object has two slots:

  - an outlet slot to hold the pointer toward the creator object (A)
  - an inlet slot which will be connected to a built-in object (B) representing the received message

  Step3. When the volatile object receives a message or detects its interface stream closing, it creates an appropriate primitive or built-in object (for example, for an integer message, `1`, it creates an integer object, `1`), and connects the inlet slot (B) to this object. Then, it activates the creator object designated from the outlet slot (A), and is scavenged.

  Step4. When the creator object is awakened, it is sure that the result is determined. The creator checks what the result is, and jumps to a label appropriate for the result.

4.6 Slot Access

Slot access is done by the following slot access instructions.

- **Immediate Execution, No Message Sending:** Because of the sequential execution strategy, slot access instructions are immediately executed to the current object, rather than sending slot access messages being sent to the object itself.

4.7 Object Creation

The following object creation instructions are provided.

- **Only General and Volatile Objects are to be Scheduled:** Only general and volatile objects are put on the schedule table. The others, including primitive objects, built-in objects and merge joints, are immediately executed in specific instructions.
- **Volatile Objects with the Same Structure as General Objects:** Volatile objects have the same structure as general objects.

5 Examples of $\mathcal{A}'\mathcal{UM}$-α Code Generation

Here are two examples of $\mathcal{A}'\mathcal{UM}$-α code generation: one is the code for *counter* and the other is the code for *append-list*.

(a) Counter

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% class  counter.
%     :up       -> !n + 1 = !n.
%     :down     -> !n - 1 = !n.
%     :set(^N) -> N = !n.
%     :show(N) -> !n = N.
% end.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        .class          #counter;
        .pid            up, "up", 0, 0x0;
        .pid            down, "down", 0, 0x0;
        .pid            'set/+', "set", 1, 0x1;
        .pid            'show/-', "show", 1, 0x0;

        .method         up, "up", 0, 0x0;
up:
        get_outlet      R1, #counter!n;
        create_integer  R2, 1;
        add             R1, R2, R4;
        set_outlet      R4, #counter!n;
        descend;

        .method         down, "down", 0, 0x0;
down:
        get_outlet      R1, #counter!n;
        create_integer  R2, 1;
        sub             R1, R2, R4;
        set_outlet      R4, #counter!n;
        descend;

        .method         'set/+', "set", 0, 0x1;
'set/+':
        set_outlet      R0, #counter!n;
        descend;

'show/-':
        get_outlet      R1, #counter!n;
        connect         R1, R0;                     % !n = ^N
        descend;

        .end            #counter;
%-----------------------------------------------------------------------
```

(b) Append-list30

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% class  test_list
%    :list30(X) ->
%        [1,2,3,4,5,6,7,8,9,10,
%         11,12,13,14,15,16,17,18,19,20,
%         21,22,23,24,25,26,27,28,29,30] = ^X .
%
%    :append_list30 ->
%        [] = ^Y,
%        #list_utility:append(X, Y, ^Z),
%         :list30(^X) .
% end.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        .class          #test_list;
        .classref       #list_utility;
        .pid            'list30/+', "list30", 1, 0x1;
        .pid            append_list30, "append_list30", 0, 0x0;

        .pid            'append/++-', "append", 3, 0x6;

        .method         'list30/+', "list30", 1, 0x1;
'list30/+':
        create_atom R3, '[]'; create_integer R2, 30; create_list R1, R2, R3;
        move        R1, R3;   create_integer R2, 29; create_list R1, R2, R3;
        move        R1, R3;   create_integer R2, 28; create_list R1, R2, R3;
                                 ... (abbr. from 27 to 4)
        move        R1, R3;   create_integer R2, 3; create_list R1, R2, R3;
        move        R1, R3;   create_integer R2, 2; create_list R1, R2, R3;
        move        R1, R3;   create_integer R2, 1; create_list R1, R2, R3;
        connect     R1, R0;              % [....] = ^X
        descend;

        .method         append_list30, "append_list30", 0, 0x0;
append_list30:
        create_atom     R1, '[]';               % [] = ^Y
        create_instance R2, #list_utility;      % #list_utility
        create_mjoint   R0;
        create_mjoint   R2;
        close           R2;
        send            R2, 'append/++-', R0, R1, R2;
        send_self       'list30/+', R0;
        descend;

        .end            #test_list;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% class  list_utility.
%     :append(^X, ^Y, Z) ->
%        ( class_of X == list ) ? (      % X:class(^Cls), Cls:eq(list, TorF)
%          :'true  ->  Y = ^Z ;
%          :'false ->  X:car(^E):cdr(^X1),
%                      [E|Z1] = ^Z,
%                      :append(X1, Y, ^Z1)
%        ).
% end.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        .class          #list_utility;
        .classref       #'if_then_else';
        .pid            'car/-', "car", 1, 0x1;
        .pid            'cdr/-', "cdr", 1, 0x1;

        .method         'append/++-', "append", 3, 0x6;
'append/++-':
```

```
        class           R0, R3;         % X:class(^Cls)
        create_atom     R4, list;
        if_eq           R3, R4, 'append/++-/i1_true', 'append/++-/i1_false';
%
        eq              R3, R4, R5;     % Cls:eq(list, ^TorF)
        who             R5, R6;         % TorF:who(Who)
        create_mjoint   R3;             % slot result
        create_volatile R4, R3, #if_then_else;
        connect         R4, R6;         % #if_then_else = ^Who
        send_continue   'append/++-/i1_cont', R0, R1, R2, R3;
        wait;

'append/++-/i1_cont':
        if_true         R3, 'append/++-/i1_true', 'append/++-/i1_false';
        raise_error;

'append/++-/i1_true':
        connect         R1, R2 ;
        descend;

'append/++-/i1_false':
        car             R0, R3;         % X:car(^E)
        cdr             R0, R4;         % X:cdr(^X1)
        create_mjoint   R0;
        create_list     R5, R3, R0;     % [E|Z1]
        connect         R5, R2;         % [E|Z1] = ^Z
        send_self       'append/++-', R4, R1, R0;
        descend;

        .end            #list_utility;
%------------------------------------------------------------------------
        .class #if_then_else;
        .inlet          result;
        .outlet         creator;
        .pid            ''true', "'true", 0, 0x0;
        .pid            ''false', "'false", 0, 0x0;

''true':
        create_boolean  R0, 'true;
        get_inlet       R1, #if_then_else!result;
        connect         R0, R1;
        get_outlet      R0, #if_then_else!creator;
        activate        R0;
        terminate;

''false':
        create_boolean  R0, 'false;
        get_inlet       R1, #if_then_else!result;
        connect         R0, R1;
        get_outlet      R0, #if_then_else!creator;
        activate        R0;
        terminate;

        .end            #if_then_else;
%------------------------------------------------------------------------
```

6 Performance Measurement

We have been implementing a software emulator of the abstract $\mathcal{A}'\mathcal{UM}$ machine, which is written in C++ and runs on the Sequent Symmetry S81 (CPU 80386, write-back cache).

We measured the basic performance (the cost of basic operations) of the current implementation of the abstract $\mathcal{A}'\mathcal{UM}$ machine against several benchmark programs.

6.1 Benchmark Programs

We measured the performance against the following five kinds of benchmark programs:

1. *Counter Program*

   (a) *Up100*

   This program creates a counter object, sets up the counter value to 0, and increments the counter value 100 times.

   This benchmark program has been chosen to measure the speed of method invocation and slot access.

2. *Stream Manipulation Program*

   This contains two benchmark programs on stream manipulation:

   (a) *Append-stream30*

   This program appends an empty stream to a stream of 30 messages.

   (b) *Reverse-stream30*

   This program reverses a stream of 30 messages.

   This set of benchmark programs has been chosen to measure the speed of stream manipulation.

3. *List Processing Program*

   This contains two benchmark programs on list processing:

   (a) *Append-list30*

   This program appends an empty list ([]) to a list of 30 elements (see the previous section).

   (b) *Reverse-list30*

   This program reverses a list of 30 elements.

Note that each list is an $\mathcal{A}'\mathcal{UM}$ built-in list object. This set of benchmark programs has been chosen to measure the speed of list processing.

4. *Message Transmission Program*

   This contains two benchmark programs on message transmission:

   (a) *Forward-transmission*

   This program first creates an integer 3, connects to this integer 100 streams one after another, and sends an addition message, `add(4, ^Sum)`, at the end.

   (b) *Backward-transmission*

   This program executes an addition operation, $X+4$, connects 100 streams one by one backward from the X, and finally creates an integer 3 ahead of the inlet of the last connected stream.

   This set of benchmark programs has been chosen to measure the speed of stream connection and message transmission.

5. *Arithmetic Operation Program*

   This contains two benchmark programs on arithmetic operation:

   (a) *3 + 4 (simple addition)*

   This program executes an addition, $3+4$, 100 times.

   (b) *X + Y (stream-indirect addition)*

   This program executes an addition, $X+Y$, 100 times, where X is a stream toward an integer 3 and Y is a stream toward an integer 4.

   This set of benchmark programs has been chosen to measure the speed of built-in arithmetic operations.

6.2 Measurement Procedure

We measured the performance as follows:

1. For each benchmark program, two kinds of methods in its class definition are prepared: one is an *original method*; the other is a *dummy method*.
2. The original method and the dummy method are executed independently.
3. The difference between the execution time spent for the original method and that spent for the dummy method is regarded as the effective time.

```
class  benchmark1.
   :doit  ->
       #counter:set(0)
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up
       :up:up:up:up:up:up:up:up:up:up.
   :dummy ->
       #counter:set(0).
```

Figure 4: Benchmark-1: counter program

Note that the *execution time* is the time which is spent from when the abstract $\mathcal{A}'\mathcal{UM}$ machine is initialized until all objects are terminated.

For example, for the counter program, we prepare the class definition as shown in Figure 4.

Let $T1$ be the time which is spent for the execution of the original method:

```
#benchmark1:doit.
```

Let $T2$ be the time which is spent for the execution of the dummy method:

```
#benchmark1:dummy.
```

Let T be the time spent for the execution of sending 100 **up** messages, then it is expressed as follows:

$$T = T1 - T2.$$

In the actual measurement, we have executed a number of invocations of the above test methods, so that the figures, $T1$ and $T2$, should be big enough to be measured.

Let n be the number of invocations of a test method. Then the time measured for testing the original method is $T1' = n \times T1$; the time measured for testing the dummy method is $T2' = n \times T2$. The difference between these figures is:

$$T' = T2' - T1' = n \times T.$$

Let m be the number of methods invoked during the time, T. Then the average time, t, spent for the execution of each method is expressed as follows:

$$t = T' \times \frac{1}{m} \times \frac{1}{n}.$$

The number of methods which are executed in one second, μ, is the reciprocal of t,

$$\mu = \frac{1}{t}$$

We use this figure, μ, as a unit, MPS (Methods Per Second), and refer to $1000 MPS$ as $kMPS$ (kilo MPS).

Note that we have performed three trials for each measurement and have taken the average time of the three results. Every other benchmark program has been measured similarly.

6.3 Measurement Result

We shows the result of the performance measurement in Figure 10.

Note that for the number, m, we count only user-defined methods. Built-in operations, such as `car(X)` in the list processing programs and `add(Y, ^Z)` in the arithmetic operation programs, are not included.

Table 10: Performance of Benchmark Programs (time unit: ms)

benchmark program	*original* $T1'$	*dummy* $T2'$	*difference* $T' = T1' - T2'$	*# invocations* n	*# methods* m	*(kMPS)* μ
Counter program						
Up100	3918	164	3754	200	100	5.33
Stream manipulation						
Append-stream30	3414	188	3226	400	30	3.72
Reverse-stream30	4831	196	4635	400	30	2.59
List processing						
Append-list30	4371	831	3540	200	31	1.75
Reverse-list30	3185	38	3147	10	496	1.58
Message transmission						
Forward-transmission	4674	224	4450	200	100	4.50
Backward-transmission	4698	218	4480	200	100	4.46
Arithmetic operation						
3 + 4	3535	191	3344	400	1	0.120
X + Y	4174	186	3988	400	1	0.100

7 Conclusions and Future Work

The work we have completed to date shows that a software implementation of the abstract $\mathcal{A}'\mathcal{UM}$ machine on a conventional von-Neumann hardware machine has attained reasonable performance. The implementation of some parts of the described design are still underway. Among them are:

1. *Alteration from Bytecode of Threaded Code.*
   In the current implementation, we adopted the bytecode emulation scheme. We are revising the implementation based on the threaded code scheme, so that the instruction dispatching cost should be reduced.

2. *Implementation of Method and Slot Caches.* For quick access to method code and slots, we will implement a method cache and a slot cache.

3. *Implementation of Foreign Language Interface.*
   We will introduce another built-in object to the design, a *foreign object* which should work as the interface to a foreign language.

In parallel, we have found several problems in the current implementation. We are now revising the design to solve these problems.

1. *Introduction of Message Objects.*

   In the $\mathcal{A}'\mathcal{UM}$ computation model described in this report, messages and objects are at different levels. For each message, its exact message pattern has to be specified in the program.

   Mainly for the purpose of message delegation, it is expected to allow a message to be designated using a variable, that is, to treat a message as an object.

   A *message object* is now under design. This object is envisioned as a *higher-order message* that encapsulates a first-order (or internal) message and allows retrieval of the message name and arguments of the first-order message. Along with the introduction of message objects, a *object-message conversion mechanism*, providing for the construction of higher-order messages and conversion to first-order messages will be introduced.

2. *Introduction of Object Groups.*
   In the $\mathcal{A}'\mathcal{UM}$ computation model, each tree of streams connects to a single object, so one-to-one communication and many-to-one communication are available.

For one-to-many communication, it must be expanded to a number of one-to-one communications.

In order to deal with one-to-many communication as easily as one-to-one communication, the notion of an *object group* will have to be introduced.

Besides the above implementation and revision, two types of parallel implementations and a full implementation will be our future work.

1. *Shared Memory Parallel Implementation.*

   The abstract $\mathcal{A}'\mathcal{UM}$ machine described in this report is a sequential implementation using a single processor, though it has been designed assuming extensibility to parallel implementation.

   The extension of the design for parallel implementation, which concerns mainly memory management, is under progress; the parallel implementation will be realized in the near future.

   Also, object priority and message priority have been introduced in the abstract $\mathcal{A}'\mathcal{UM}$ machine. Along with their manipulation, we will have to study load balancing schemes and parallel debugging schemes.

2. *Full Implementation.*

   The $\mathcal{A}'\mathcal{UM}$-α instruction set has been designed for a subset of $\mathcal{A}'\mathcal{UM}$, which has some sequentiality restrictions placed on conditionals and sending messages to self.

   Also, more specifically, the adoption of the pointer-tag method rather than the object-tag method as a method to identify structure types resulted in limiting the ability of object mutation.

   Removing these restrictions and redesigning the abstract instruction set which will deal with the full set of $\mathcal{A}'\mathcal{UM}$ and make available the full parallelism without loss of efficiency will be a theme for future work.

3. *Local Memory Distributed Implementation.*

   The abstract $\mathcal{A}'\mathcal{UM}$ machine has been designed assuming a shared-memory multiprocessor as a target machine. There is a limit on the number of processors which can be connected to a shared-memory.

   The design and implementation of $\mathcal{A}'\mathcal{UM}$ on distributed processors with local memory and a common communication line will be a big future theme.

Acknowledgments

Akihiko Konagaya, Tsutomu Maruyama, Kouichi Konishi, Shinji Yanagida and Satoshi Oyanagi participated in the design and implementation of the abstract $\mathcal{A}'\mathcal{UM}$ machine. We would also like to express our thanks to Kazuhiro Fuchi, the director of ICOT, and Shun'ichi Uchida, the manager of the fourth research laboratory, for their valuable suggestions and encouragement.

References

[Yoshida and Chikayama 88B] Kaoru Yoshida and Takashi Chikayama: "$\mathcal{A}'\mathcal{UM}$ - A Stream-Based Concurrent Object-Oriented Language -", Technical Report TR-388, ICOT; In *Proceedings of International Conference on Fifth Generation Computer Systems (FGCS'88)*, pp.638-649, ICOT(OHM/Springer-Verlag), November 1988.

[Yoshida 90A] Kaoru Yoshida: "$\mathcal{A}'\mathcal{UM}$: A Stream-Based Concurrent Object-Oriented Programming Language", Ph.D thesis, Keio University, March 1990.