Instruction Set Architecture of Mamba, a New Virtual Machine for Python

David Pereira and John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4

{pereira,aycock}@cpsc.ucalgary.ca

TR 2002−706−09

Abstract: Python programs are implemented by compiling them into code for a virtual machine. Mamba is a new virtual machine for Python whose design is minimalist and register−based. In contrast, the current Python virtual machine is stack−based and contains almost six times as many instructions as Mamba. We anticipate that our smaller instruction set will greatly simplify optimization implementation. In this paper we detail Mamba's instruction set and execution model.

Programming Model

Mamba is a new virtual machine for the Python language. Its instruction set is quite different from the instruction set of the standard Python virtual machine; there are 18 register–based instructions instead of 103 stack–based instructions. This reduction in instruction set size is accomplished by removing functionality from instructions and relocating that functionality into objects that the instructions act upon.

By moving functionality into objects we can remove many instructions, as long as:

1. All object types present the functionality they support via an interface.
2. The virtual machine provides a set of basic instructions to access that common interface.

For example, the standard Python virtual machine has an instruction named `BINARY_SUBSCR`. This instruction uses the top object on the stack as an index into the object below it. The object returned by this subscription operation then replaces the two top items on the stack. The implementation of this instruction contains special cases to subscript different types of objects such as strings, tuples, lists, and dictionaries.

However, in the Mamba virtual machine there is no `BINARY_SUBSCR` instruction. Instead, each type which supports the subscription operation provides that functionality via the `__getitem__` method. In this case,

1. The `__getitem__` method for tuples will check if the given index is an integer. If not, it will raise a `TypeError` exception. Then it will check if the index is out of range. If so, it will raise an `IndexError` exception. Otherwise, it will return the object at that position in the tuple.

2. The `__getitem__` method for lists would work similarly, as would the `__getitem__` method for strings.

3. The `__getitem__` method for dictionaries would use the given parameter as a key. It would then search for a key–value mapping containing the given key, and, if there exists such a mapping, it will return the corresponding value. Otherwise it will raise a `KeyError` exception.

4. Objects that do not support that this functionality will not have a `__getitem__` method.

The virtual machine provides an instruction to retrieve methods such as `__getitem__` from objects that provide them. It also provides an instruction to call a method with a set of arguments. The following example, written in Mamba pseudo–assembler, does this.

```
GETATTR     R_container, "__getitem__", R_getitem
CALL        R_getitem (R_key) -> R_result
```

In the above example, $R_{container}$, and R_{key} are registers which contain references to the indexable object (such as a string, tuple, list or dictionary) and the indexing object, respectively. R_{result} is the register that will hold the result of the computation. The `GETATTR` instruction extracts the `__getitem__` method for the object referenced in $R_{container}$ and puts it into register $R_{getitem}$. Then the `CALL` instruction calls the method object referenced in register $R_{getitem}$ with the object referenced in register R_{key} as its parameter and then places the return value in register R_{result}. These two Mamba instructions perform the equivalent of the `BINARY_SUBSCR` instruction. *Note:* Instructions in the Mamba virtual machine are orthogonal in that any register may be used for any operand. In the above code, any register in the register set can replace a symbolic register name such as $R_{container}$.

As a general rule, the expression:

```
result = OPERATION val, parameter₁,...,parameterₙ
```

is expressed in this virtual machine as

```
result = val.OPERATION(parameter₁,...,parameterₙ).
```

However, there is an exception to this rule. The `OP` instruction provides support for binary arithmetic operations. It is provided even though it can be can rendered redundant by using other instructions. This is because its operation is quite complicated, and, if excluded, code bloat would result since a lengthy sequence of virtual machine instructions would be required to emulate its operation. Furthermore, this sequence has not been packaged as a function since it is frequently required and the function call overhead would be very costly.

We mentioned two criteria above by which Mamba moves functionality into objects, allowing us to reduce the number of virtual machine instructions. In detail, these are:

1. The common interface. Each object in this virtual machine is a namespace – a map from names to objects. In this machine, a name is any string of characters, called the *attribute name*. The value associated with an attribute name can be any Python object, called the *attribute value*. A pair consisting of an attribute name and its associated attribute value is called an *attribute*. In the above example, `__getitem__` is an attribute of string, tuple, list, and dictionary objects; `__getitem__` is the attribute name and the method object implementing the desired functionality is the attribute value.

2. A set of instructions to access the common interface. There are three main instructions to access and modify the namespace of an object:
    1. `GETATTR` – Given an object and an attribute name, it returns the associated attribute value.
    2. `SETATTR` – Given an object, an attribute name and a value, it changes the attribute value associated with the attribute name to the given value, if an attribute with the given attribute name already exists in the namespace of the object. Otherwise, it creates an attribute with the given name and value in the namespace of the object.
    3. `DELATTR` – Given an object, and an attribute name, it deletes the attribute name and its associated attribute value from the namespace of the object.

Execution Model

The execution model for this machine can be viewed from three levels: the thread level, the inter–procedural level and the intra–procedural level.

This virtual machine provides support for running only a single thread of Python code.

Interprocedural Control Flow in this virtual machine is provided by function calls and exception–handling. Function calls are supported by a call stack which stores the activation records of each function invocation. Each function invocation is given a new register set. When it makes a call to another function, the register set is saved and the called function is executed. The saved registers are restored when the called function returns. Nested functions are not currently supported.

Support for asynchronous interprocedural control flow is provided by exceptions. A terminating model of exception handling is used in which only one exception may be active at any given time. The point at which an exception is thrown and the point at which it is caught can span arbitrarily many function invocations.

Intraprocedural Control Flow is accomplished via jump instructions and exception handling.

Memory management is performed automatically without programmer intervention.

Instruction Set Overview

The instructions of the virtual machine fall into the following categories:

1. Namespace manipulation instructions:
    1. `GETATTR` See above
    2. `SETATTR` See above.
    3. `DELATTR`: See above.
    4. `GETATTRS`: A variant of `GETATTR` which works with multiple attribute names.
2. Control Flow instructions:
    1. Structured
        1. `CALL` Call a function
        2. `RETURN` Return from a function
        3. `RAISE` Raise an exception
        4. `HANDLE` Handle an exception
    2. Unstructured
        1. `JUMP` Unconditional jump
        2. `JUMPZ` Jump if == `0`
        3. `JUMPNZ` Jump if <> `0`
        4. `JUMPA` Jump if > `0`
        5. `JUMPNA` Jump if <= `0`
        6. `JUMPB` Jump if < `0`
        7. `JUMPNB` Jump if >= `0`
3. Miscellaneous
    1. `IMPORT` Import a module
    2. `NEW` Create a new object
    3. `OP` Perform a binary arithmetic operation

The Register Set

There are a maximum of 4096 registers in the virtual machine available at any time to a function invocation. The n^{th} register is designated R_n. All registers have the same functionality; they can be used with any instruction. The first 64 registers are global; their contents are preserved across function calls. The remaining 4032 registers are local; their values are saved before a function call is made, and restored upon return from that call. (It should be noted that only the registers used by the calling function are saved. Hence, the overhead of saving and restoring registers is restricted to set of registers used by the calling function). The global registers will be referred to as G_0, G_1, ..., and L_0, L_1, ... will denote the local registers.

The global registers G_{60} and G_{61} have a special purpose. G_{60} stores the primary exception datum and G_{61} stores the secondary exception datum. That is, G_{60} and G_{61} provide a place to store exceptions across function calls while the exception handling mechanism is trying to find a handler for them. The objects in these registers are matched against the objects given to the `HANDLE` instruction, in order to determine if a given exception handler should handle the exception the registers contain. G_{60} and G_{61} can also be used as parameters to the `RAISE` instruction, in order to re–raise the current exception.

Of the local registers, L_0 and L_1 have a special purpose. They are used to store the global and local namespaces of a function respectively. When a function is called, L_0 is set to the global namespace for the called function (even for function calls across modules), and L_1 is initialized to a new empty namespace. It should be noted that the local namespace is linked to the global namespace. Therefore, if a search of the local namespace fails, the search continues in the global namespace. For details, see documentation for the `GETATTR` instruction.

All instructions with the exception of `MOVE` have an encoding which uses 8 bits to address a register. As a result, these instructions can address only the first 256 registers. The remaining registers can be accessed only via the `MOVE` instruction, whose encoding uses 12 bits to address a register.

Objects and Attributes

The following is a list of object types supported by the virtual machine. With the exception of the `Namespace` object type, the following types are Python language types. A `Namespace` object is simply a collection of attributes.

1. `Integer`
2. `Long Integer`
3. `Float`
4. `Complex`
5. `String`
6. `Tuple`
7. `Code`
8. `List`
9. `Dictionary`
10. `Function`
11. `Class`
12. `Namespace`
13. `None`
14. `NotImplemented`
15. `Ellipsis`
16. `Type`
17. `Slice`
18. `Method`
19. `Instance`
20. `Module`
21. `Traceback`

The attributes that each of these objects have are specified in the *Python Language Specification, Section 3.3 Data Model – Special Method Names*. Certain special attributes and their uses are discussed later in the *Instruction Set Reference* section.

Code Objects

The fundamental object in the virtual machine is the code object. It should not be confused with a `Function` object; a `Function` object references a code object and provides the context in which to execute the instructions in the code object. A code object, on the other hand, contains the list of virtual machine instruction that constitute the body of a function. It also contains certain ancillary data needed by the instructions which it contains.

The first of these data is the *attribute name table*. Instructions such as `GETATTR`, `SETATTR`, and `DELATTR` require an attribute name parameter. The attribute name table in a code object contains the attribute names used by the instructions in that code object. Just before the virtual machine begins to execute the instructions in a given code object, it designates the attribute name table of that code object as the *current attribute name table* to which all references to attribute names made by `GETATTR`, `SETATTR`, `DELATTR`, etc., are made. When the virtual machine finishes executing code in that code object, the previous current attribute name table becomes the current attribute name table. There is always a current attribute name table which the `GETATTR`, `SETATTR`, `DELATTR`, etc., instructions implicitly reference.

The second important data item is the *pickle jar*. The pickle jar contains *pickles* – descriptions of data from which objects can be created. Atomic objects such as `Integers`, `Floats`, `Complexes`, `Longs`, and `Strings` are represented as pickles in the pickle jar, and then transformed into virtual machine objects on demand by the `NEW` instruction. A mechanism similar to the one used to manage attribute name tables is used for pickle jars. Just before the virtual machine begins to execute code in a given code object, it designates the pickle jar of that code object as the *current pickle jar.* When the virtual machine finishes executing code in that code object, the previous current pickle jar is made the current pickle jar. As such there is always a current pickle jar which the `NEW` instruction implicitly references. It should be noted that a code object's pickle jar can contain one or more pickled `Code` objects, thereby forming a tree of objects.

File Format

Each byte code file interpreted by this virtual machine corresponds to a Python module. Its filename must end with an extension of `.pyc`, because the `IMPORT` instruction appends this extension to a module name to derive the filename containing the module's code.

Each byte code file must contain a *single* pickled `Code` object. It is an error for any other type of object or for more than one object to be present. When a module is loaded via the `IMPORT` instruction, its `Code` object is unpickled, wrapped inside a `Function` object and then called with the `CALL` instruction. This `Code` object should contain a list of instructions which initialize the module by creating its global data and functions.

The format of a pickled `Integer` object is:

Field Name	Size (Bytes)	Description
TYPE	4	The value of this field must be 0.
VALUE	4	The value of the 32–bit integer. All data in Mamba's file format is assumed to use the byte ordering of the machine architecture that Mamba is running on.

The format of a pickled `Long` object is:

Field Name	Size	Description
TYPE	4	The value of this field must be 1.
VALUE		The value of this field must be an ASCII representation of the long integer, in base 10, terminated by a '\0'. If the length of the ASCII string (including the terminating '\0') is not a multiple of 4, pad bytes must be added to the end so that the start of the next object is aligned to a 32–bit address.

The format of a pickled `Float` object is:

Field Name	Size	Description
TYPE	4	The value of this field must be 2.
VALUE	8	This field must contain the value of the floating–point number as an IEEE 754 double–precision floating–point number.

The format of a pickled `Complex` object is:

Field Name	Size	Description
TYPE	4	The value of this field must be 3.
REAL	8	This field specifies the real component of the complex number. It must be an IEEE 754 double–precision floating–point number.
IMAG	8	This field specifies the imaginary component of the complex number. It must also be an IEEE 754 double–precision floating–point number.

The format of a pickled `String` object is as follows:

Field Name	Size	Description
TYPE	4	The value of this field must be 4.
LENGTH	4	The field specifies the length of the string.
STRING		This field contains an ASCII representation of the string. If the length of the string is not a multiple of 4, the end of the string must be padded so that the next pickle will be aligned to a 32−bit address.

The format of a pickled `Code` object is as follows:

Field Name	Size	Description
TYPE	4	The value in this field must be 6.
RC	2	The number of local registers used by the code in this code object.
IC	4	The number of instructions in this code object.
IL	4 * IC	The instructions of this code object. (Each instruction is 4 bytes long).
PI PI.S1 PI.DC PI.S2 PI.NDC	2	A specification of the formal parameters of the code object. The bitfield *abbbbbbb cddddddd* describes the layout of PI, where *a* = 1, if the code object has a * parameter, 0 otherwise. *bbbbbbb* = number of parameters that do not have default arguments. *c* = 1, if the code object has a ** parameter, 0 otherwise. *ddddddd* = number of parameters that have default arguments.
PNOT	(AI.DC + AI.NDC)*4	The table of parameter name offsets. The offsets are ordered so that they lexicographically order the arguments found in the following table.
PNT		The table of parameter names. Each formal parameter name is a string at the offset indicated in the table above, terminated by a null character, and followed by a 8−bit integer indicating its position in the list of formal parameters as stated in the function definition.
ETC	2	The number of entries in the exception table.
LTC	2	The number of entries in the line number table.
ET	ETC * 12	The Exception Table. Each entry in the exception mapping table consists of three 32−bit integers. The first two entries specify a range of instructions in IL, and the third specifies the instruction in IL that control should be transferred to if an exception occurs at an instruction within that range. The start and end addresses of ranges are inclusive. Ranges cannot overlap, and the exception table is sorted by the start−of−range in increasing order.
LT	LT*12	The Line Number Table. Each entry of the line number table consists of three 32−bit integers. The first two entries specify a range of instructions in IL, and the third specifies the line number in the source code file to which that byte code range corresponds. This information is used to generate informative tracebacks for the programmer during debugging. Like the exception table, byte code ranges may not overlap, and the line number table is sorted into increasing order.
ANTC	2	This field specifies the number of entries in the Attribute Name Table.
ANOT	ANTC*4	This specifies the offset of each attribute name in the Attribute Name Table.
ANT		The table contains a list of '\0'−terminated attribute names at the offset specified in the attribute name offset table.
PC	4	The number of pickled objects in the pickle jar of this code object.
PJ		The set of pickles for this code object. Each pickle must start at a 32−bit offset.

The format of an Indirection pickle is below. Indirection pickles are used when the range of normal pickle jar offsets would be insufficient.

Field Name	Size	Description
TYPE	4	The value in this field must be 22.
ADDRESS	4	The offset of the pickled object from the start of the pickle jar of this code object.

Translating Python to Mamba Assembler

The following translations are not intended to be exhaustive so much as illustrative. They cover the most commonly used constructs in a Python programmer's repertoire. Unimportant sections of code will be represented by an ellipsis in the following Python and assembler listings.

Translation of Python Expressions

Simple Literals

Literal constants such as `1`, `2.2`, `3L`, `4J` and `"Hello, World"` are represented as pickles inside the pickle jar of the code object that they appear in using the format given in the *File Format* section. They are unpickled into full–fledged Python objects by the `NEW` instruction whenever their values are required in a Python program. In order to alleviate the burden of dealing with numeric offsets into the pickle jar, the assembler provides symbolic addresses for pickles by means of the directive

pickle–name`:` *type value*

which instructs the assembler to insert a pickle in the required format into the pickle jar of any code object which references it with the pseudo–instruction

`UNPICKLE` *pickle–name*`,` $R_{\texttt{destination register}}$

`UNPICKLE` is a synonym for the `NEW` instruction when its task is to unpickle *pickle–name* and to place the resulting object into register $R_{\texttt{destination register}}$. We use the assembler directive

`ASSEMBLE` *ndpc dpc s1 s2*
`PARAMETERS` $name_1$, $name_2$, `...`, $name_n$
body of code object
`END`

to represent a code object, where *ndpc* and *dpc* are non–negative integers specifying the number of formal parameters which have default arguments and the number of formal parameters that do not have default arguments, respectively. *S1* and *S2* indicate whether the code object has `*` and `**` formal parameters, respectively. They can be either zero or one. The `PARAMETERS` specification contains an ordered list of formal parameter names. The list of pickles for a code object typically follows the `ASSEMBLE` ... `END` directive describing the code object.

Compound Literals

Compound literals such as `[1,2,3,4,5]` are not represented as pickled objects but must be built at run–time. The following code creates the compound literal `[1,2,3,4,5]`:

```
ASSEMBLE ...
      NEW         LIST, L2
      GETATTR     L2, "append", L3         # Get the append method

      UNPICKLE    T0, L5
      CALL        L3, 1, L4, 0             # Append a 1 at the end
      UNPICKLE    T1, L5
      CALL        L3, 1, L4, 0             # Append a 2 at the end
      ...
END
```

```
T0: INTEGER 1
T1: INTEGER 2
...
```

Dictionary constants can be created similarly. First, an empty dictionary is created. Then the `__setitem__` function is retrieved from it and is called with each key–value pair that should be in the dictionary. Tuples are created by first making a list with the desired tuple components, and then calling the list's `__tuplify__` function, which will transform the list object into a tuple.

Binary Arithmetic

Binary arithmetic operators are translated with the `OP` instruction. `OP` implements in–place operations as well as the regular kind. All aspects of Python arithmetic operations such as the coercion of operands and fall–backs to reverse operations in the absence of standard operations are implemented by this instruction. The expression `x + y` is translated as

$$\texttt{OP add, } R_x\texttt{, } R_y\texttt{, } R_r$$

Where R_x is the register containing `x`, R_y is the register containing `y`, and R_r is the register which should contain the result. They expression `x += y` is translated as

$$\texttt{OP iadd, } R_x\texttt{, } R_y\texttt{, } R_r$$

It is important to note that a result register is required even for the in–place operation because there is no guarantee that object `x` supports in–place addition. If it does not, the instruction would fall back to "normal" addition and possibly even to reverse addition, in the absence of support for "normal". The result would then be placed in R_r.

Unary Arithmetic

Unary arithmetic operations are translated by using function calls. For example the expression `-x` is translated as

```
GETATTR      Rx, "__neg__", Rr
CALL         Rr, 0, Rr, 0
```

Here the GETATTR instruction retrieves the negation function of R_x into R_r. A `CALL` is made to the negation function and the result is placed in R_r. R_r now contains the required value. It should be noted that the original value of the object in R_x has not been modified. The operations + and ~ are translated similarly using the attribute names `__pos__` and `__invert__` respectively. Please refer to *Section 3.3* of the *Python Language Specification* for details.

Comparisons

The `GETATTRS` instruction is used to translate Python comparison statements. For example, the expression `x < y` is translated as

```
GETATTRS     Rx, <, Rx
MOVE         Ry, Rx+1
CALL         Rx, 1, Rx, 0
```

The `GETATTRS` instruction returns a method which should be called with the second operand of the comparison. The other comparison operations are supported similarly. The `GETATTRS` instruction ensures that rich comparison methods such as `__lt__` are used if they are provided by the given object. Otherwise, the comparison method `__cmp__` is used. Each object in the virtual machine has an implicit `__cmp__` method which compares objects by address.

Identity

The Python language provides the `IS` operator to test whether two objects have the same identity. This operation is implemented by comparing the addresses of the two objects. The address of an object is obtained by calling its `__addr__` function, which returns an `Integer` containing the object's address. Once two `Integer` addresses have been obtained, they may be compared for equality using the `GETATTRS` instruction as shown above.

Containment

The Python language provides the `IN` operator to test whether one object is contained in another. The `__contains__` member function is used to implement this test. In order to check if object `x` contains object `y`, the `__contains__` function is extracted from object `x` and then called with object `y` as its argument. A non–zero value is returned if object `x` contains object `y`, zero otherwise. The `NOT IN` expression can be translated by inverting the return value.

Logical Operations

The truth value of an object can be determined by calling a function to determine its value; the correct function to call is found using the `GETATTRS` instruction. By using conditional jumps and the `OP` instruction with the `and` and `or` operations, short–circuiting logical operations can be performed.

Slicing

Slicing operations are implemented via an object's `__getslice__`, `__setslice__` and `__delslice__` methods. For example to compile the code `x[i:j] = k` the following translation is used.

```
GETATTR     Rx, "__setitem__", Rr
MOVE        Ri, Rr+1
MOVE        Rj, Rr+2
MOVE        Rk, Rr+3
CALL        Rr, 3, Rr, 0
```

Code to retrieve and delete slices can be obtained by using `__getslice__` and `__delslice__` respectively.

Indexing

Indexing is a special case of slicing which uses the functions `__getitem__`, `__setitem__`, and `__delitem__` instead of `__getslice__`, `__setslice__` and `__delslice__`.

Qualification

The qualification operation is built directly into the virtual machine in the form of the GETATTR and SETATTR instructions. When qualification occurs in a non–assignment context, the expression `x.y` is translated as `GETATTR` R_x, `"y"`, R_r. When used in the context of an assignment such as `x.y = z`, the SETATTR instruction is used: `SETATTR` R_x, `"y"`, R_z.

Function Calls

One step in the compilation of a function call that needs to be elucidated is the passing of keyword arguments. To do this, a `Namespace` is created containing a mapping of argument names to values. For example, to compile the function call `func(1,x=2,y=3)`, a namespace object is created and populated with the required values.

```
GETATTR     L0, "func", L2     # Retrieve "func" from globals
UNPICKLE    T1, L3             # Assume pickle T1: INTEGER 1
NEW         Namespace, L4
UNPICKLE    T2, L4             # Assume pickle T2: INTEGER 2
SETATTR     L2, "x", L4
UNPICKLE    T3, L4             # Assume pickle T3: INTEGER 3
SETATTR     L2, "y", L4
```

Now the first argument is in `L3` and the namespace containing the second and third arguments is in `L4`. We can now call the function with the instruction

```
CALL        L2, 1, L2, 1
```

This will call the function in `L2` with one positional argument and will put the result of the call in `L2`. The fourth operand to the `CALL` instruction indicates that a `Namespace` object containing the value of keyword arguments is provided in the register immediately following the registers containing the positional arguments (in this example, the keyword argument is in register `L3`).

Translation of Python Statements

Assignment

Let us assume that `x` and `y` are local variables and that we want to translate the expression code `x = y`. The following translation is used:

```
GETATTR     L1, "y", L2
SETATTR     L1, "x", L2
```

If `x` and `y` are global variables then `L0` can be used instead of `L1`. If `x` is global and `y` is local then `L0` can be used instead of `L1` in the `SETATTR` instruction. Otherwise, if `x` is local and `y` is global then `L0` can be used instead of `L1` in the `GETATTR` instruction. Compound assignments such as `x, (a,b) = y, (c,d)` must be decomposed into simpler assignments and then translated using the schema given above. More complicated expressions involving slicing and indexing can be translated using the `__setslice__` and `__setitem__` methods of the objects which are being assigned to.

Printing

To translate the statement `PRINT` *exp*, where *exp* is any Python expression, the expression *exp* should first be evaluated into a register. The `__repr__` method of the result should then be called. It will return the required output as a `String` object. The `print` method of the `String` object returned should then be called, which will print the string to standard output.

String Formatting

In Python, string formatting is performed with the `%` operator. This operation can be translated exactly as if it were a binary arithmetic operation involving the `%` operator by using the `OP mod` instruction.

String Representation

The "official" representation of an object can be obtained by calling its `__repr__` method. The "unofficial" representation can be obtained by calling the `__str__` method of the object.

If−Elif−Else and While

These translations require the use of the `__nonzero__` method to ascertain the truth value of an expression.

For−In

To translate this construct, the compiler should generate code to obtain the length of the container being iterated over by calling its `__len__` method. The `__getitem__` method of the container should then be called with each value from 0 up to the value returned by the call to `__len__` in order to obtain each item held in the container.

Try−Except

To compile a `try` statement, the starting and ending addresses of the virtual machine instructions covered by the translated `try` statement must be entered into the exception table, along with the address of the code to transfer control to in the event of an exception. The assembler alleviates the tedium of calculating these offsets by providing the `TRY` directive which does this. For example, the Python code

```
try:
      # Exception raising code
except ...:
      # Exception handling code sequence 1
except ...:
      # Exception handling code sequence 2
```

is translated

```
TRY EXCEPTION_HANDLER1
      translation of exception raising code

      # No exceptions were encountered. Jump over handlers.
      JUMP AFTER_EXCEPTION_HANDLERS
END

EXCEPTION_HANDLER_1:
      # Try to match the exception.
      # If match fails, try second handler
      HANDLE ..., EXCEPTION_HANDLER2

      # The match succeeded.
      translation of exception handling sequence 1
      JUMP AFTER_EXCEPTION_HANDLERS

EXCEPTION_HANDLER_2:
      # Try to match the exception.
      # If match fails, reraise the exception.
      HANDLE ..., RERAISE_EXCEPTION

      # The match succeeded.
      translation of exception handling sequence 2
      JUMP AFTER_EXCEPTION_HANDLERS
```

```
RERAISE_EXCEPTION
      # No handler matched. Reraise the current exception.
      RAISE G61, G62

AFTER_EXCEPTION_HANDLERS:
```

The `TRY` directive will insert the triple containing the address of the first instruction after the opening `TRY EXCEPTION_HANDLER`, the address of the last instruction before the enclosing `END`, and the address of the first instruction after the label `EXCEPTION_HANDLER` into the exception table for the containing code object. The `TRY` directive can be nested.

As illustrated above, the address that control is transferred to usually contains a `HANDLE` instruction which attempts to match a thrown exception to a particular `Class`, `Instance` or `String` object. If the exception matches, according to the *Python Language Specification*, execution continues at the next instruction, which contains the handler for the exception. Otherwise, the `HANDLE` instruction transfers control to another address which usually contains another `HANDLE` statement that attempts to match the exception. The last `HANDLE` instruction in such a chain usually transfers control to a `RAISE` instruction which uses registers G_{61} and G_{62} as arguments, thereby re–raising the current exception. See the *Instruction Set Reference* for a sample translation.

Function Definitions

Let us assume that the following function definition is found at the global scope.

```
def sum(a=1, b=1):
      return a+b
```

To compile a function definition, a pickled code object should first be created with the appropriate data as described in the section *Code Objects*. At run–time, the code object is unpickled and wrapped in a `Function` object. The default values for formal parameters and the global namespace for the function are then set up for the `Function` object. The following translation shows how the above Python code listing could be compiled.

```
ASSEMBLE ...
      NEW         Function,  L2          # Create a new function

      # Set up the __im_code__ attribute of the Function object
      UNPICKLE    C0, L3                 # Unpickle the code object
      SETATTR     L2, "__im_code__", L3  # Link to it

      # Set up the __globals__ attribute of the Function object
      SETATTR     L2, "__globals__", L0  # Use globals of this module

      # Set up the __defaults__ attribute
      NEW         List, L3               # Create the defaults list
      GETATTR     L3, "append", L4
      UNPICKLE    T1, L6                 # Assume pickle T1: INTEGER 1
      CALL        L4, 1, L5, 0           # Append 1
      UNPICKLE    T2, L6                 # Assume pickle T2: INTEGER 2
      CALL        L4, 1, L5, 0           # Append 2
      SETATTR     L2, "__defaults__", L3

      # Now link the function name to the function object
      SETATTR     L0, "sum", L2
END

C0: ASSEMBLE 0 2 0 0
      OP          add, L2, L3, L2
      RETURN      L2
END
```

Class Definitions

To compile a class definition, a `Class` object should first be created. The list of base classes should then be created and set as the value of the `__bases__` attribute. Finally, all objects declared with it must be compiled and made part of the class' namespace. In the following program we shall assume that `Base1` and `Base2` are global variables.

```
class Derived(Base1, Base2):
      ...
      data = 1
      ...
```

We may translate this class definition as

```
NEW         Class, L2               # Create a new Class
NEW         List, L3                # Create a new List
GETATTR     L3, "append", L4
GETATTR     L0, "Base1", L6         # Get Base1
CALL        L4, 1, L5, 0            # Append Base1
GETATTR     L0, "Base2", L6         # Get Base2
CALL        L4, 1, L5, 0            # Append Base2
SETATTR     L2, "__bases__", L3     # Set the __bases__ attribute
# Now compile the rest of the class
...
UNPICKLE    T0, L3                  # Assume T0: INTEGER 1
SETATTR     L2, "data", L3
...
```

Note that the last instruction sets `data` as an attribute of the object in `L2` – the class that was just created – not as an attribute of the global or local namespaces. It should also be noted that when functions declared within a class are compiled that their `__globals__` namespace is *not* the class in which they are compiled but the global namespace of the module.

Deletion

The `del` statement is supported directly by the virtual machine. Suppose that we want to delete the local variable `x` using the Python statement `del x`. The translation is

```
DELATTR L1, "x"
```

If `x` were a global variable the `DELATTR` statement would use `L0` instead of `L1`.

Example Translation: Modules

This example was chosen because it indicates how classes, class member data, function definitions with default values for parameters, exception handling, if–elif–else and while statements should be compiled.

```
class ValueNotFoundError:
      pass

class Tree:
      def __init__(self):
            self.root = None

      def value(self, key, exception=ValueNotFoundError):
            # Start at the root
            current = self.root

            # Proceed to search
            found = 0
            while 1:
                  if current == None:
                        break
                  elif current.key < key:
                        current = self.left
                  elif current.key > key:
                        current = self.right
                  else:
                        found = 1
                        break

            # Return value found or raise exception
            if found:
                  return current.value
            else:
                  raise exception

class AVLTree(Tree):
      def __init__(self):
            Tree.__init__(self)

      def insert(self, key, value):
            code to insert key, with balancing

# The main program
try:
      names = AVLTree()
      names.insert(1, "One")
      names.insert(2, "Two")
      name = names.value(3)

except ValueNotFoundError:
      print 'ValueNotFoundError'
      raise
```

In the following translation the `ENTRY` directive is introduced. This directive tells the assembler that the pickle name that follows it is the main pickle in the assembler file. When the byte code file produced by the assembler is imported, the code object designated by the `ENTRY` directive will be executed.

```
ENTRY MAIN_CODE_OBJECT
MAIN_CODE_OBJECT:
      ASSEMBLE          0 0 0 0
      PARAMETERS

# class ValueNotFoundError
      NEW               Class, L2
      SETATTR           L0, "ValueNotFoundError", L2

# class Tree:
      NEW               Class, L2
      SETATTR           L0, "Tree", L2

# def __init__(self):
      NEW               Function, L3
      UNPICKLE          C1, L4
      SETATTR           L3, "__im_code__",  L4
      SETATTR           L3, "__globals__", L0
      NEW               List, L4
      SETATTR           L3, "__defaults__", L4
      SETATTR           L2, "__init__", L3

# def value(self,key,exception=ValueNotFoundError):
      NEW               Function, L3
      UNPICKLE          C2, L4
      SETATTR           L3, "__im_code__",  L4
      SETATTR           L3, "__globals__", L0
      NEW               List, L4
      GETATTR           L4, "append", L5
      GETATTR           L0, "ValueNotFoundError", L6
      CALL              L5, 1, L5, 0
      SETATTR           L3, "__defaults__", L4
      SETATTR           L2, "value", L3

# class AVLTree(Tree):
      NEW               Class, L2
      NEW               List, L3
      GETATTR           L3, "append", L4
      GETATTR           L0, "Tree",   L5
      CALL              L4, 1, L4, 0
      SETATTR           L2, "__bases__", L3
      SETATTR           L0, "AVLTree", L2

# def __init__(self):
      NEW               Function, L3
      UNPICKLE          C3, L4
      SETATTR           L3, "__im_code__",  L4
      SETATTR           L3, "__globals__", L0
      NEW               List, L4
      SETATTR           L3, "__defaults__", L4
      SETATTR           L2, "__init__", L3

# def insert(self,key,value):
      NEW               Function, L3
      UNPICKLE          C4, L4
      SETATTR           L3, "__im_code__",  L4
      SETATTR           L3, "__globals__", L0
      NEW               List, L4
      SETATTR           L3, "__defaults__", L4
      SETATTR           L2, "insert", L3

# Assemble the try statement
TRY S0
      # names = AVLTree()
      GETATTR           L0, "AVLTree", L2
```

```
      CALL              L2, 0, L2, 0
      SETATTR           L0, "names", L2

      # names.insert(1, "One")
      GETATTR           L0, "names", L2
      GETATTR           L2, "insert", L2
      UNPICKLE          T1, L3
      UNPICKLE          T2, L4
      CALL              L2, 2, L2, 0

      # names.insert(2, "Two")
      GETATTR           L0, "names", L2
      GETATTR           L2, "insert", L2
      UNPICKLE          T3, L3
      UNPICKLE          T4, L4
      CALL              L2, 2, L2, 0

      # name = names.value(3)
      GETATTR           L0, "names", L2
      GETATTR           L2, "value", L2
      UNPICKLE          T5, L3
      CALL              L2, 1, L2, 0
      SETATTR           L0, "name", L2
      JUMP              AFTER_HANDLER
S0:
      # print `ValueNotFoundError`
      GETATTR           L0, "print", L2
      GETATTR           L0, "ValueNotFoundError", L3
      GETATTR           L3, "__repr__", L3
      CALL              L3, 0, L3, 0
      CALL              L2, 1, L2, 0

AFTER_HANDLER:
      # The global namespace for the module is returned
      RETURN            L0
END

T1: INTEGER 1
T2: STRING  "One"
T3: INTEGER 2
T4: STRING  "Two"
T5: INTEGER 3

# Assemble method Tree.__init__
C1:   ASSEMBLE          1 0 0 0
      PARAMETERS        self

      # Assemble self.root = None
      GETATTR           L0, "None", L3
      SETATTR           L2, "root", L3    # L2 contains parameter 1
      RETURN            L3
END

# Assemble method Tree.value
C2:   ASSEMBLE          2 1 0 0
      PARAMETERS        self key exception

      # current = self.root
      GETATTR           L2, "root", L5    # L2 contains parameter 1
      SETATTR           L1, "current", L2

      # found = 0
      UNPICKLE          C2_1, L5
      SETATTR           L1, "found", L5
```

```
      # while 1:
REP:
      # if current == None:
      GETATTR            L1, "current", L5
      GETATTRS           L5, ==, rL5
      GETATTR            L0, "None", L6
      CALL               L5, 1, L5, 0

      # break
      JNZ                L5, OUT

      # elif current.key < key
      GETATTR            L1, "current", L5
      GETATTR            L5, "key", L5
      GETATTRS           L5, <, L5
      MOVE               L3, "key", L6     # L3 contains parameter 2
      CALL               L5, 1, L5, 0

      # Jump to next test if test failed
      JZ                 L5, TEST2

      # current = self.left
      GETATTR            L2, "left", L5    # L2 contains parameter 1
      SETATTR            L1, "current", L5
      JUMP               BOTTOM

TEST2:
      # elif current.key > key
      GETATTR            L1, "current", L5
      GETATTR            L5, "key", L5
      GETATTRS           L5, >, L5
      MOVE               L3, "key", L6
      CALL               L5, 1, L5, 0
      JZ                 L5, TEST3

      # current = self.right
      GETATTR            L2, "right", L5
      SETATTR            L1, "current", L5
      JUMP               BOTTOM

TEST3:
      # else: found = 1
      UNPICKLE           C2_1, L5
      SETATTR            L1, "found", L5
      # break
      JUMP               OUT

      # Bottom of while loop
BOTTOM:
      JUMP               REP

OUT:  # End of while loop

      # if found:
      GETATTR            L1, "found", L5
      GETATTR            L5, "__nonzero__", L5
      CALL               L5, 0, L5, 0
      JNZ                L5, NOT_FOUND

      # return current.value
      GETATTR            L1, "current", L5
      GETATTR            L5, "value", L5
      RETURN             L5
```

```
NOT_FOUND:
      # else: raise exception.
      # Use the ".raise" function to derive the operands to RAISE
      # by setting up the global exception data registers G60 and G61.
      GETATTR            L0, ".raise", L5
      MOVE               L4, L6
      CALL               L5, 0, L5, 1
      RAISE              G60, G61
END


# Assemble method AVLTree.__init__
C3:   ASSEMBLE           1 0 0 0
      PARAMETERS         self

      # Assemble Tree.__init__(self)
      GETATTR            L0, "Tree", L3
      GETATTR            L3, "__init__", L3
      MOVE               L2, L4             # L2 contains parameter 1
      CALL               L3, 1, L3, 0

      GETATTR            L0, "None", L3
      RETURN             L3
END

C4:   ASSEMBLE           3 0 0 0
      PARAMETERS         self key value
      ...
END
```

Example Translation: The Function Call Mechanism

The following example illustrates the function call mechanism from the point of view of the caller and the callee. The function called will have positional parameters, default−valued parameters, a * parameter, a ** argument, and tuple parameters. The call to the function will pass positional parameters, keyword parameters, a * argument, and a ** parameter. The function, once called, will access keyword parameters, access the * and ** parameters, and unpack tuple parameters. Assume the following definition:

```
def function(a,b,c,(d,e,f),g,h=1,i=2,*j,**k):
        function body
```

Suppose that the function is called with the following code:

```
function (1, g=2, i=3, *(4,5,(6,7,8)), **{"h":9, "j":10})
```

The assembler code for function definition is:

```
FUNCTION:
      ASSEMBLE    5, 2, 1, 1
      # There are 5 positional parameters, 2 default-valued parameters,
      # a * parameter, and a ** parameter

      PARAMETERS        a b c d e f g h i j k
      # The parameter names

      # Parameters a, b, and c are in L2-L4 respectively.
      # Parameters g,h,i,j, and k are in L5-L9 respectively.

      # Unpack the tuple that is in L4 to obtain parameters d, e, and f.
      # We shall store parameters d, e, and f in L10-L12 respectively.

      GETATTR     L4, "__getitem__", L13

      # Get the first item
      UNPICKLE    T0, L15
      CALL        L13, 1, L14, 0
      MOVE        L14, L10

      # Get the second item
      UNPICKLE    T1, L15
      CALL        L13, 1, L14, 0
      MOVE        L14, L11

      # Get the third item
      UNPICKLE    T2, L15
      CALL        L13, 1, L14, 0
      MOVE        L14, L12

      code for function body
END
T0: INTEGER 1
T1: INTEGER 2
T2: INTEGER 3
```

The main issue demonstrated by the above example is tuple parameter unpacking. There are other ways to unpack tuple parameters. For example, parameters after the tuple parameter can be moved forward, and the unpacked value inserted in front of them so that the unpacked values maintain their order in the list of parameters. Another alternative is to create local variables to hold the values of unpacked tuple parameters.

We now show the assembler code for the function call:

```
GETATTR     L0, "function", L2

# Set up the sole positional argument in L3
UNPICKLE    T1, L3

# Set up the namespace containing arguments g and I in L4
NEW         Namespace, L4
UNPICKLE    T2, L5
SETATTR     L4, "g", L5
UNPICKLE    T3, L5
SETATTR     L4, "h", L5

# Create the tuple (4,5,(6,7,8)) in L5
NEW         List, L5
GETATTR     L5, "append", L6
UNPICKLE    T4, L8
CALL        L6, 1, L7, 0
UNPICKLE    T5, L8
CALL        L6, 1, L7, 0
# Create the tuple (6,7,8) in L8
NEW         List, L8
GETATTR     L8, "append", L9
UNPICKLE    T6, L11
CALL        L9, 1, L10, 0
UNPICKLE    T7, R11
CALL        L9, 1, L10, 0
UNPICKLE    T8, R11
CALL        L9, 1, L10, 0
GETATTR     R8, "__tuplify__", L9
CALL        L9, 0, L9, 0
# Join (6,7,8) and then create the tuple from the list
CALL        L6, 1, L7, 0
GETATTR     L6, "__tuplify__", L6
CALL        L6, 0, L6, 0

# Create the dictionary containing parameter h and j in L6
NEW         Dictionary, L6
GETATTR     L6, "__setitem__", L7
UNPICKLE    T11, L9
UNPICKLE    T9, L10
CALL        L7, 2, L8, 0      # Set up parameter h
UNPICKLE    T12, L9
UNPICKLE    T10, L10
CALL        L7, 2, L8, 0      # Set up parameter j

# Make the call
CALL        L2, 1, L2, 7

T1: INTEGER 1
T2: INTEGER 2
...
T10: INTEGER 10
T11: STRING "h"
T12: STRING "j"
```

The value 7 for the fourth parameter to the call requires some explanation: bit 0 is set to indicate the presence of a * parameter, bit 1 the presence of the ** parameter, and bit 2 the presence of a namespace of keyword arguments following the positional parameter.

Instruction Set Reference

1. GETATTR

Description

Given an object and an attribute name, retrieve the corresponding attribute value. If the given attribute name is not found, an `AttributeError` exception is thrown.

Form

GETATTR *object*, *attribute name*, *attribute value*

Parameters

1. *object* is an 8−bit unsigned integer indicating the register which references the object from which the attribute−value will be retrieved.
2. *attribute−name* is a 14−bit index into the current attribute name table specifying the string which contains the attribute name.
3. *attribute−value* is an 8−bit unsigned integer indicating the register into which a reference to the retrieved attribute value should be stored.

Semantics

The two main algorithms used by this instruction are:
1. the algorithm used to find the attribute value;
2. the algorithm used to bind the attribute value (if it was found and binding is required).

Algorithm 1: Finding the attribute value:
The attribute value lookup operation depends on the type of object in which the `GETATTR` instruction is searching:

1. Objects of type `None`, `NotImplemented`, `Ellipsis`, `Integer`, `Long`, `Float`, `Complex`, `String`, `Tuple`, `List`, `Dictionary`, `Slice`, `Function`, `Code`, `Method`, `Frame`, `Traceback`, and `Type` have a trivial attribute value lookup procedure − an <*attribute name*, *attribute value*> pair is either part of the object's namespace or it is not. If the attribute name is not found in the namespace then no other objects are searched, the search fails, and an `AttributeError` exception is thrown. Otherwise, if the attribute name is found in the namespace, the corresponding attribute value is returned.

2. Objects of type `Class`, `Instance`, `Module`, and `Namespace` have more elaborate lookup schemes:

    1. `Class`: If an attribute name is not found in a class object, its base class(es) must be searched. The list of base classes must be given in the `__bases__` attribute. If this attribute does not exist, or if its value is not a `Tuple` object where each component is a `Class` object, an `AttributeError` exception is thrown. Each class in the tuple of base classes is searched in order, with the same algorithm applied if the given attribute name is not found in a base class. Hence, a depth−first search of the class tree for a given

class is performed. The first attribute value corresponding to the given attribute name is retrieved. Two or more base classes may contain an attribute with the same name, but the attribute value corresponding to the attribute name in the *first* class searched will be returned. This means that the order in which base class are specified in the `__bases__` tuple matters.

2. `Instance`: If an attribute name is not found in an instance object of a class, its class is then searched. The class (which was called to create an instance) must be given by the instance's `__class__` attribute. This object must be of type `Class`. If the attribute is not found in the class specified by `__class__`, that class is searched using the depth−first algorithm given above.

3. `Module`: If an attribute name is not found in a module object, the set of built−in objects of the module is then searched. The object containing the module's built−in objects is given by the the module's `__builtins__` attribute. This object must be a `Namespace` object. It is searched with the algorithm given below.

4. `Namespace`: Since `Namespace` objects are used to implement the local namespace of a function, if an attribute name is not found in a namespace object, the object specified by its `__globals__` attribute is searched. The `__globals__` attribute must be an object of type `Module`, and is searched with the aforementioned algorithm. It is not compulsory for a namespace object to have a `__globals__` attribute.

Since the search algorithm for classes, instances, modules and namespace object, result in other objects being searched when the given attribute name is not found, it is useful to differentiate between the object in which the search started and the object in which the given attribute name was found (i.e., the object in which the search ended).

1. In the following discussion when we use the phrase "via an *object*", we mean that *object* is simply the object in which the search started, not necessarily the object in which the attribute name was found.

2. Furthermore, when we used the phase "inside an *object*", we mean that *object* is the object in which the attribute name search ended.

Algorithm 2: Binding the attribute value.
Assume that Algorithm 1 has executed and the attribute value has been located. We may now classify the type of the attribute value into two mutually exclusive categories: function objects and non−function objects.

1. Function objects are simply objects of the type `Function` (this category does not include objects that can behave like functions, such as instances of classes which have a `__call__` method defined):

    1. When a `Function` is retrieved via a class instance but found inside a base class of the instance, the function is wrapped inside a bound `Method` object which binds the first parameter of the function to the instance. The `Method` object is returned.

    2. When a `Function` is retrieved via a class, the function is wrapped inside an unbound `Method` object which binds the type of the first parameter to the class via which it was retrieved. This ensures that the unbound `Method`

object is called with an instance of that class as its first parameter. The Method object is returned.

3. When a `Function` is retrieved via an `Instance`, but also found inside that `Instance`, no bound `Method` is created. The function is returned, unadorned.

4. When a `Function` is retrieved from a `Module` or `Namespace`, no bound `Method` is created. The function is returned unadorned.

5. When a `Function` is retrieved from any other type of object, the `Function` is wrapped inside a bound `Method` object which binds the first parameter to that object. The `Method` object is returned.

2. Non– function objects are not bound. Their values are simply returned as–is.

For more details, see the *Python Language Specification*.

Encoding

`00xxxxxx xxxxxxxx` *<object> <attribute value>*

`xxxxxx xxxxxxxx` = 14–bit attribute–name.

Total instruction length: 4 bytes.

Example

The `GETATTR` instruction provides functionality for the Python . (dot) operator when it occurs in a non–assignment context. Consider the following code:

```
currentTemperature = temperature.centigrade()
```

Let `currentTemperature` be a global variable and `temperature` be a local variable which is an instance of class `Temperature`, containing a function `centigrade(self)`. When compiled, this would be:

```
GETATTR     L1, "temperature", L2   # L1 contains local namespace
GETATTR     L2, "centigrade", L2    # create a bound method object
CALL        L2, 0, L2, 0            # Call bound method object;
                                    # return value in L2
SETATTR     L0, "currentTemperature", L2
                                    # L0 contains global namespace
```

2. GETATTRS

Description

This instruction will locate the first of a sequence of attributes that a specified object possesses.

Form

`GETATTRS` *object, attribute–sequence, attribute value*

Parameters

1. *object* is an 8–bit integer indicating the register which contains an object in which the attribute–name search should start.
2. *attribute–sequence* is an 8–bit unsigned integer indicating an attribute name sequence which is built into the virtual machine.
3. *attribute value* is an 8–bit integer indicating a register in which to store the attribute value that is found.

Semantics

Given an object and a sequence of attribute names, this instruction will perform the operation of the `GETATTR` instruction for the given object and each attribute name in the sequence, in the order it appears in the sequence. If no attribute–name in the sequence is found, an `AttributeError` exception is thrown. Otherwise, a reference to the attribute value corresponding to the first attribute name found is returned.

The attribute sequence table is as follows:

Seq No.	Name 1	Name 2	Name 3
0	`__lt__`	`__cmp__ *`	`__.d__lt__`
1	`__gt__`	`__cmp__ *`	`__.d__gt__`
2	`__eq__`	`__cmp__ *`	`__.d__eq__`
3	`__ne__`	`__cmp__`	`__.d__ne__`
4	`__ge__`	`__cmp__ *`	`__.d__ge__`
5	`__le__`	`__cmp__ *`	`__.d__le__`
6	`__nonzero__`	`__len__`	`__.d__len__`
7	`__str__`	`__repr__`	`__.d__repr__`

If an attribute name marked with an asterisk (*) is found, its attribute value will not be returned directly but will be wrapped inside a Function object, which calls `__cmp__` and translates its return values appropriately. This is essential in all cases except Sequence 3, in which `__cmp__` returns 0 if the object compare equal, and non–zero otherwise. These values are in agreement with the values which are returned by the rich comparison method `__ne__`. Hence, no translation is required for this case.

The `__.d__` attributes refer to attributes that are present in all objects by default, which compare objects by their identity. This is done to be compliant with Python semantics.

Encoding

`10110100` *< object> < attribute name sequence> <attribute value >*

Total instruction length: 4 bytes.

Example

The `GETATTRS` instruction is used to provide support for Python operations which allow fall–back methods to be used, if present. Suppose that we want to translate the expression `x < y` where `x` and `y` are local variables.

We may translate this code as:

```
GETATTR     L1, "x", L2
GETATTRS    L2, <, L2
GETATTR     L1, "y", L3
CALL        L2, 1, L2, 0
```

3. SETATTR

Description

Given an object, an attribute name, and a value, this instruction will change the attribute value for the given attribute name to the given value. This operation is called rebinding. If the given attribute name is not found, it will create an *<attribute name, attribute value>* pair with the given attribute name and attribute value in the object. This operation is called binding.

Form

SETATTR *object*, *attribute name*, *attribute value*

Parameters

1. *object* is an 8−bit value indicating the register which references the object in whose namespace the binding/rebinding will occur.
2. *attribute name* is a 14−bit index into the current attribute name table specifying the string which contains the attribute name.
3. *attribute value* is an 8−bit value indicating the register which references the value to be bound/rebound to the *attribute name*.

Semantics

Unlike the `GETATTR` instruction which may search other objects for the attribute name, no such search is employed by this instruction. The instruction modifies the namespace of the object given by its first parameter.

To understand the conditions under which exceptions are thrown by this instruction, it is important to know that objects of the virtual machine can be categorized in terms of the modifiability of their namespaces. These categories are parameterized according to two criteria: the mutability of the attribute names, and the mutability of attribute values:

Attribute−name−immutable objects have a fixed set of attributes. It is illegal to bind values to attribute names in objects of this type, if bindings for those attribute names do not already exist.

Attribute−value−immutable objects do not allow the attribute values for objects to be changed. (The attribute values are usually set up by the virtual machine to refer to built−in functions.)

Thus in terms of namespace modifiability, objects in the virtual machine can be partitioned into three categories:

1. Attribute−name−immutable, attribute−value−immutable:
   `Integer`, `Long`, `Float`, `Complex`, `String`, `Tuple`, `Code`, `List`, `Dictionary`, `Ellipsis`, `Type`, `Slice`, `None`, and `NotImplemented`.
2. Attribute−name−immutable, attribute−value−mutable:
   `Function` and `Method`.

3. Attribute–name–mutable, attribute–value–mutable:
   `Class`, `Instance`, `Namespace`, and `Module`.

If the namespace of an object is attribute–name–immutable and the instruction tries to bind a new attribute in it, or if the namespace is attribute–value–immutable and the instruction tries to rebind an already bound attribute name to a new value, an `AttributeError` exception will be thrown.

Encoding

`01xxxxxx xxxxxxxx` *<object> <attribute value>*

`xxxxxxxxxxxxxx` = 14 bit attribute–name.

Total instruction length: 4 bytes.

Example

Assume that the following Python code is inside a function:

```
global y
y = 1
x = 2
```

We may translate the above code as follows:

code to create a reference to the integer 1 in L2
```
SETATTR      L0, "y", L2
```
code to create a reference to the integer 2 in L3
```
SETATTR      L1, "x", L3
```

Recall that `L0` and `L1` point to the global and local namespace, respectively.

4. DELATTR

Description

Given an object and an attribute name, remove the <*attribute name*, *attribute value*> pair associated with the given attribute name if such a pair exists, or raise an `AttributeError` exception otherwise. The process of removing an <*attribute name*, *attribute value*> pair is called unbinding.

Form

`DELATTR` *object*, *attribute name*

Parameters

1. *object* is an 8–bit value indicating the register which references the object from whose namespace the attribute should be unbound.
2. *attribute name* is a 14–bit index into the current attribute name table specifying the string containing the attribute name.

Semantics

The operation of this instruction depends on the modifiability of *object*'s namespace. (Modifiability categories are discussed in the `SETATTR` instruction.) If the namespace is attribute–name–immutable, an `AttributeError` exception will be raised.

Otherwise, the instruction will search for a <*attribute name*, *attribute value*> pair with the given attribute name in the namespace of the *object.* If no such pair is found, an `AttributeError` exception is raised. Otherwise, the <*attribute name*, *attribute value*> pair found will be deleted.

The search algorithm for the <*attribute name*, *attribute value*> pair does not proceed into other objects, if the search fails for the given object. The `DELATTR` instruction is only concerned with *object*.

Encoding

`10001000` <*object*> `00xxxxxx xxxxxxxx`

`xxxxxx xxxxxxxxx` = 14–bit *attribute name*.

Total instruction length: 4 bytes.

Example

The purpose of this instruction is to provide functionality for Python's `del` statement.

Example 1: Suppose that the following code occurs inside a function and that `temp` is a local variable. We may translate the statement

```
del temp
```

as

```
DELATTR L1, "temp".
```

Recall that `L1` contains the local namespace of a function.

Example 2: Suppose in the following code that `Temperature` is a class object and that `toCentigrade(self)` is one of its methods. We may delete the method with the following Python code:

```
del Temperature.toCentigrade
```

The code may be translated as:

```
# Code to move Temperature into register n
DELATTR Rn, "toCentigrade"
```

5. CALL

Form

CALL *callable object,*
number of positional arguments,
flag indicating presence of keyword arguments,
*flag indicating presence of * argument,*
*flag indicating presence of ** argument,*
return value

Parameters

1. *callable object* is an 8–bit unsigned integer indicating the register which contains a reference to the object to be called. This object may be a function, a bound method, an unbound method, a class, or a class instance.
2. *number of positional arguments* is an 8–bit value containing the number of positional arguments that are to be passed to the function designated by *callable object*.
3. *return value* is an 8–bit unsigned integer indicating the register which will hold the value returned by the function.
4. *keyword arguments flag* indicates whether a `Namespace` object argument containing a mapping from argument names to argument values is present. This argument is used to represent arguments that are passed by using the syntax:
   ```
   function(..., argname=argvalue, ...)
   ```
5. The * *argument flag* indicates whether an object representing a set of positional arguments will be passed to the function. A tuple is usually used here. More generally, any object which defines the `__len__` and `__getitem__` functions appropriately can be used here. Hence, list, strings, dictionaries and classes instances which have the required function defined are acceptable as arguments.
6. The ** *argument flag* indicates whether a dictionary containing a mapping from arguments names to argument values will be passed to the function.

Order of Arguments

The positional arguments must be placed in the first register after the return value register. They must be placed in a set of consecutive registers, followed (optionally) by the register containing the namespace object (containing the call's keyword arguments), followed (optionally) by the register containing the * argument, followed (optionally) by the register containing the ** argument.

Any register can be used to hold the reference to the callable object.

Semantics

The first stage of the operation of this instruction is function extraction. If the callable object is a an `Instance`, the `__call__` method is retrieved from it. If no such method is present, an `AttributeError` exception is raised. If the callable object is a `Class`, the `__init__` method is retrieved from it. As with `Instances`, if this operation fails, an `AttributeError` is thrown. If the callable object is a bound method, the function associated with it is retrieved from the `__im_func__` attribute and the first parameter is retrieved from the `__im_self__` attribute. However, if the callable object is an unbound method, the first parameter is not retrieved from `__im_self__` but a check is performed to ensure that the first parameter passed (explicitly via the `CALL` instruction) is a sub–class of the class referenced by the `__im_class__` parameter.

Parameter setup is done in accordance with the *Python Language Specification.* In the callee, incoming parameters are placed in a contiguous set of registers beginning at register `L2`. All positional parameters are placed first, followed (optionally) by the * parameter if the function accepts one, followed (optionally) by the ** parameter if the function accepts one. The * parameter will be a `Tuple`, and the ** parameter will be a `Dictionary`.

When a function is called, a new local register set will be created for it. The registers will form a consecutive range starting at `L0` (register 64). Registers `L0` and `L1` are reserved for the global and local namespaces, respectively. As soon as a function call begins, `L0` is set to point to the object referenced by its `__globals__` attribute. If the `__globals__` attribute is not present, it is assumed that the function has no global variables, and `L0` is set to `None`. A local namespace is then set up for the function in `L1`. The local namespace is also modified to contain an attribute `__globals__` which references the object referenced by `L0`. Hence, if an attribute name is not found during a search of the local namespace, the search will proceed to the globals, in accordance with the scoping rules of Python.

Finally, the code object containing the byte-code body of the function is retrieved via the `__im_code__` attribute. The instructions are then interpreted.

Encoding

$10000f_1f_2f_3$ *<callable objects> <positional argument count> <return value>*

f_1 = 1, if keyword arguments are present, 0 otherwise.
f_2 = 1, if a * argument is present, 0 otherwise.
f_3 = 1, if a ** argument is present, 0 otherwise.

Total instruction length: 4 bytes

Example

Example 1: (From the perspective of the caller). Suppose a reference to a function declared as `def function(a,b,c,*e,*f)` is contained in `L6`, and we want to want to compile the function call expression `function(1, 2, c=3, d=4, *x, **y)`. Let us further assume that `x` and `y` are local variables and that we want the result in `L7`. We may translate the code as follows:

```
# First set up the parameters starting in register 72
UNPICKLE    T0, L8          # assume pickle T0: INTEGER 1
UNPICKLE    T1, L9          # assume pickle T1: INTEGER 2

# Now set up the namespace
NEW         Namespace, L10

# Put c = 3 in the namespace
UNPICKLE    T2, L11         # assume pickle T2: INTEGER 3
SETATTR     r74, "c", L11

# Put d = 4 in the namespace
UNPICKLE    T3, L11         # assume pickle T3: INTEGER 4
SETATTR     L10, "c", L11

# Place *x and *y
GETATTR     L1, "x", L11
GETATTR     L1, "y", L12

# Make the call
CALL        L6, 2, L7, 7
```

Example 2: (From the perspective of the callee). Suppose that we are compiling a function whose signature is `def function(a,b,c,d,*e,*f)`. Then, when this function is called, parameters `a` thru `f` can be accessed in registers `L2` through `L8` respectively.

6. RETURN

Description

Return a value from a function.

Form

RETURN *<return value>*

Parameters

Return value is an 8–bit unsigned integer indicating the register which a reference to the object which should be returned as the return value of the function.

Semantics

This instruction will place the specified object from the callee and return it in the caller's register designated to hold the return value by the CALL instruction which invoked the function.

It will also restore, upon return, the register set of the calling function.

Encoding

`10001100 00000000 00000000` *<return value>*

Total instruction length: 4 bytes

7. RAISE

Description

Raise an exception.

Form

RAISE *first exception object*, *second exception object*

Parameters

Both parameters are 8–bit unsigned integer values which indicate registers.

Semantics

The RAISE instruction raises an exception with the values referenced by *first exception object* and *second exception object* as the primary and secondary values of the exception.

When an exception is raised, the virtual machine tries to locate code that will handle the exception, using the algorithm below.

Let IP be the address of the instruction raising the exception
Loop
    If there is a range in the current exception table containing IP
        Jump to the instruction associated with the range
    Otherwise
        Destroy the current function invocation; the new current exception table is now the caller's

Encoding

`10010000 00000000` <*first exception object* > <*second exception object* >

Total instruction length: 4 bytes.

8. HANDLE

Description

This instruction attempts to match an exception which has been thrown by the `RAISE` instruction or by the virtual machine. The matching algorithm is given below. If the matching fails the instruction pointer will be set to the location specified by *label*. The purpose of the label is to jump to another `HANDLE` statement (the usual case), or to jump to a `RAISE` statement which will repropagate the exception after all attempts to match it have been exhausted.

Form

HANDLE *object, label*

Parameters

1. *object* is an 8–bit unsigned integer which indicates the register containing the object which will be matched against the current exception.
2. *label* is a 16–bit signed relative offset which indicates the location that the code should branch to, if the match fails.

Semantics

The `HANDLE` instruction uses the following algorithm to match an exception:

If *object* is a string
    If the primary exception datum (the object in G_{60}) is a string
        Compare the address of the two objects for equality
        If they do not compare
            Control is transferred to *label*
    Otherwise
        Control is transferred to *label*
Otherwise, if *object* is a class object
    If the secondary exception datum (the object in G_{61}) is a class object
        If *object* is not a superclass of the secondary exception datum
            Control is transferred to *label*
    Otherwise
        Control is transferred to *label*
Otherwise
    Control is transferred to *label*

Encoding

`10010100` *<object>* *<16−bit signed offset>*

Total instruction length: 4 bytes

Example

Suppose we want to translate the following Python code:

```
try:
      # code which might raise an exception
except TypeError:
      # code which handles a TypeError
except AttributeError:
      # code which handles an AttributeError
```

The code is translated:

```
TRY H1
      translation of code which might raise an exception
END
JUMP  OVER
H1:
      GETATTR     L0, "TypeError", L2
      HANDLE      L2, H2
      translation of code which handles a TypeError
      JUMP        OVER
H2:
      GETATTR     L0, "AttributeError", L2
      HANDLE      L2, REPROPAGATE
      translation of code which handles an AttributeError
      JUMP        OVER
REPROPAGATE:
      RAISE       G60, G61
OVER:
```

9. MOVE

Form

`MOVE` *source register, destination register*

Parameters

Source register and *destination register* are two 12−bit unsigned integers which designate registers.

As a result of the operands being 12 bits each, each operand of this instruction can access a total of 4096 registers, in contrast to the 256 registers that other instructions can access.

Semantics

This instruction copies the reference contained in *source register* into the *destination register*; the contents of the destination register are the same as that of the source register following the execution of this instruction. Hence, this is a non−destructive operation.

Encoding

`10011000` *<source register*$_{12}$*>* *<destination register*$_{12}$*>*

Total instruction length: 4 bytes.

10. JUMP

Form

`JUMP` *offset*

Parameters

offset, a 24–bit signed relative offset indicating an instruction to which control should be unconditionally transferred.

Semantics

This instruction transfers control to the the given instruction by adding the given offset, less one, to the current instruction pointer.

For example, a jump instruction with an offset of zero is a no–op, and a jump instruction with an offset of −1 is an infinite loop, since it jumps back to the current instruction.

Encoding

`10011100` *<24–bit signed relative address>*

Total instruction length: 4 bytes.

11. Jcond

Description

Each of these instructions will jump to the given address if the object referenced by the given register satisfies a specified condition. The conditions are Z, NZ, A, NA, B, NB.

Form

Jcond *<test value >* *<address>*

Parameters

1. *test value*, an 8–bit unsigned integer indicating a register which contains a reference to an object integer object.
2. *address*, a 16–bit relative offset, to which control will be transferred if the referenced object satisfies the jump condition.

Semantics

Code	Meaning	Condition
Z	Jump if zero	Value == 0
NZ	Jump if nonzero	Value != 0
A	Jump if above	Value > 0
NA	Jump if not above	Value <= 0
B	Jump if below	Value < 0
NB	Jump if not below	Value >= 0

The referenced object must be of type `Integer`. If the integer value contained in the object satisfies the given condition, control is transferred to the given address. If the condition is not met, execution resumes at the instruction immediately following the J*cond*. If the given object is not an integer, a `TypeError` exception is raised.

Encoding

JZ	`10100000`	<*test value*> <*address*>
JNZ	`10100100`	<*test value*> <*address*>
JA	`10100001`	<*test value*> <*address*>
JNA	`10100010`	<*test value*> <*address*>
JB	`10100011`	<*test value*> <*address*>
JNB	`10100100`	<*test value*> <*address*>

Total instruction length: 4 bytes.

13. NEW

Description

Create a new object, or unpickle a pickled object into a full−fledged virtual machine object.

Form

`NEW` <*offset*>, <*register*>

Parameters

1. *offset* is a 19−bit offset into the current pickle jar. Values of *offset* greater than $2^{19}-64-1$ are reserved, and denote new objects whose contents are not pickled. The values and their corresponding object types are given below.
2. *register* is an 8−bit integer indicating the register which will hold the newly created object.

Semantics

Objects in the virtual machine may be classified into two categories based on their method of creation − pickled and unpickled objects. Pickled objects are simple, atomic objects such as `Integers`, `Longs`, `Floats`, `Complexes` and `Strings` whose representations are stored at compile−time in an area of each function's code object called the pickle jar. When one of these objects is required to be incarnated at run−time, its offset in the pickle jar is passed to the `NEW` instruction and the object is incarnated from its pickled representation. Non−pickled objects (whose types are shown in the table below) are not stored in the pickle jar. They are created at run−time from scratch. In order to create them, an offset between $2^{19}-64$ and $2^{19}-64+21$ is passed to the NEW instruction. In both cases, pickled and unpickled objects are returned in the desired register.

If an object greater than or equal to $2^{19}-64-1$ is needed, an "indirection" pickled object can be used.

Encoding

`10101xxx xxxxxxxx xxxxxxxx` *register*

`xxx xxxxxxxx xxxxxxxx` = 19−bit offset

Total instruction length: 4 bytes.

Offsets for non−pickled object types are given by the following table:

Type	Offset
List	$2^{19}-64-1+9$
Dictionary	$2^{19}-64-1+10$
Function	$2^{19}-64-1+12$
Class	$2^{19}-64-1+15$
Namespace	$2^{19}-64-1+21$

14. IMPORT

Description

Import a module from a `.pyc` file.

Form

IMPORT *module name, register*

Parameters

1. *module name* is a 14−bit offset (for consistency with other instructions) into the current attribute name table which specifies a string containing the name of the module to be imported. This instruction (ab)uses the attribute name table as a repository for string operands to its *module name* parameter.
2. *register* is an 8−bit integer indicating the register which should hold the module object, after it has been loaded.

Semantics

The IMPORT instruction will:

1. Open the named file.
2. Unpickle the (sole) code object inside it.
3. Wrap the code object inside a function object.
4. Call the function it has just created.

5. Put the object returned by the function into the specified register. It is the obligation of the code inside the code object to return the global namespace of the module, if the object put into *register* is to make any sense.

Encoding

`10110000` *<register>* `00xxxxxx xxxxxxxx`

`xxxxxx xxxxxxxx` = 14–bit offset

Total instruction length: 4 bytes.

Example

For example, suppose that we want to load the `os` module, so that we can write code to remove the file `.cshrc`. We can do the following:

```
IMPORT       "os", L2
GETATTR      L2, "remove", L3

UNPICKLE     FILE_NAME, L4
CALL         L3, 1, L3, 0

FILE_NAME: STRING ".cshrc"
```

15. OP

Description

This instruction will perform a binary or in–place arithmetic operation. If necessary, operand coercion and fall–back to reversed operation is performed.

Form

`OP`*xxxxxx object1, object2, result*

Parameters

1. *object1* and *object2* are 8–bit integers indicating registers which contain references to the operands to be operated upon.
2. *xxxxxx* is a 6–bit integer which refers to the operation to be performed (see table below).
3. *result* is an 8–bit integer indicating a register in which the result is to be stored.

Semantics

When a binary operator is applied to its operands in Python, coercion of the operands may be necessary before the required operation can be performed. The virtual machine performs operand coercion and the required operation in one step via the `OP` instruction.

This instruction takes two operands pointed at by *object1* and *object2* and coerces them into two objects of a "common" type (if necessary) using the coercion procedure specified in the Python language reference. The result of the operation is returned in the result register.

This instruction is used to compile both ordinary operations and in–place operations.

Encoding

`11xxxxxx` *<object1>* *<object2>* *<result>*

`xxxxxxx` = a 6–bit integer indicating which operation should be performed.

Total instruction length: 4 bytes.

The operation field is encoded as follows:

afffff: where *a* = 0: if the operation is simple, e.g. `__add__`
a = 1: if the operation is in–place, e.g. `__iadd__`
and *fffff* indicates the basic operation given in the table below.

Operation	Value
`__add__`	0001
`__sub__`	0010
`__mul__`	0011
`__div__`	0100
`__mod__`	0101
`__pow__`	0110
`__lshift__`	0111
`__rshift__`	1000
`__and__`	1001
`__xor__`	1010
`__or__`	1011

Reference

van Rossum, G., and Drake, Jr., F. L. *Python Reference Manual, Release 2.2 (December, 2001).* Available from `http://www.python.org`.