VI

Object-Oriented Computer Architecture:

- Concepts and Issues
- The REKURSIV Object-Oriented Computer Architecture

D. M. Harland

Rapporteur: G. Dixon

Object-Oriented Computer Architecture

Two lectures for the International Seminar on Object-Oriented Computing Systems
Newcastle, September 6-9, 1988.

To be presented by :

David M Harland
Technical Director, Linn Smart Computing Ltd, Glasgow G45
and Professor of Computer Architecture, Strathclyde University.

(1). Object-Oriented Computer Architecture: Concepts and Issues

This lecture will introduce the relevant concepts and set out the architectural issues which arose during the design of the hardware developed by Linn for the object-oriented programming paradigm.

Issues covered will include:

Stores :

Object persistence
Object swapping
Object security (unique identifiers, range checking, symbolic activation)

Processors :

High level instructions (no semantic gap, even higher ordered and recursive)
The MIPS rate (clock rates, caches, pipelines and prefetch units)

Technology :

Software?
Off the shelf?
Semi-custom?
Full-custom?

(2) The REKURSIV Object-Oriented Computer Architecture

This lecture will concentrate on the practicalities of configuring the REKURSIV to a variety of different application domains and will discuss topics such as

Microcoding an object-based instruction set,
Language integration,
Process communication,
Garbage collection,
The future (e.g. distributed object stores).

Various examples will be given.

Demonstrations :

A simulator for the microcoded architecture to run on a Sun using X-windows will be available, as will a REKURSIV accelerator board for a Sun.

Reference :

REKURSIV Object-Oriented Computer Architecture, by D M Harland, published by Ellis Horwood Ltd, August 1988.

OOPS Issues

- complexity
- multiple paradigms
- abstraction mechanisms
- types
- polymorphism
- expressivity
- parallelism
- verifiability
- technology

VI.1

Storage Issues

- von Neumann storage
- semi inferent retention
- object representation
- persistence
- object swapping
- high level instructions
- security
- type protection
- range protection

REKURSIV Technology

- 40 bit tagged architecture
- totally object oriented
- 32 bit word
- compact representations for 32 bit objects
- three microcodable ASICs
  - 1.5 micron CMOS
  - sea of gates
  - 10MHz
  - Logik
    - sequencer for high level instruction sets
    - 299 pins
    - 60 bit control word
    - 18 fields
  - Objekt
    - object oriented memory management
    - 299 pins
    - 32 bit control word
    - 11 fields
  - Numerik
    - 32 bit alu, barrel shifter, multiplier and register file
    - 223 pins
    - 70 bit control word
    - 17 fields

EXAMPLE - CONSing

Stack has CAR above CDR; replaced with CONS.

```
CONS:   page_allocator NOFETCH ldidx d=brch (2) ldidx
        d=idx ldsb grabspace&setaddr&ldab&initflags&ldnb NOFETCH \
              cjbr @GC MEMOFLO
        d=bron (CONS_TYPE) ldtb decsp newesp idx2 loadaddr
        d=estk idxput idx1 loadaddr
        d=estk idxput ldrb
        d=ref ldestk
```


REKURSIV Slides

EXAMPLE - CDRing

Stack has a CONS, replace with head of third link.

(DEF CADDDR (x) (CAR (CDR (CDR (CDR x)))))

```
CADDDR:d=estk pagebus
        d=vrr pagebus
        d=vrr pagebus
        d=vrr pagebus
        idx2 loadaddr
        idxget
        d=memout ldestk
```


Linn Smart Computing: Tel (44) 41 631 1483 September '88

EXAMPLE - TREECOPYing

Stack has a tree, replaced with copied tree.

```
(DEF COPYTREE (x)
        (COND ((ATOM x) x)
              (T (CONS (TREECOPY (CDR x))
                       (TREECOPY (CAR x))
                 )
              )
)
```

```
TREECOPY: d=estk pagebus
        d=bron (CONS_TYPE) ldsym
        cjupcor !IS_SYM d=vtr
        incsp newesp inccsp
        jbr @TREECOPY ldestk d=vrr pushupcor
        u=sp (1) usubbrch newesp
        readestk
        d=estk pagebus
        idx2 loadaddr
        idxget incsp newesp
        jbr @TREECOPY d=memout ldestk
        jbr @CONS
```


Copytree Benchmark

```
(DEF COPYTREE (ITEM)
  (COND ((ATOM ITEM) ITEM)
    (T (CONS (COPYTREE (CDR ITEM)) (COPYTREE (CAR ITEM))))))
```

	RATIO
Xerox 1186	171
TI Explorer	57
Tektronix 4406	28
Symbolics 3675	18
MIPS R/2000 (RISC)	1.2
REKURSIV 10MHz	1.0

EXAMPLE - Creating An Object

Stack has size, above type, above initial values for components; replace with object.

```
CREATE: page_allocator decsp newesp NOFETCH
        d=estk ldsb grabspace&setaddr&ldab&initflags&ldnb ldidx decsp newesp \
                cjbr @GC MEMOFLO
        d=estk ldtb cjbr @empty IDX=0 loadaddr readestk

        newmark d=estk decsp newesp idxput cldrb
        cjm IDXOK decidx loadaddr readestk

        incsp newesp
        d=ref ldestk
        ...

empty:  incsp newesp d=brch (NIL) ldrb
        d=ref ldestk
        ...
```


Creating An Object

Assuming that the stack has been loaded with the size, type and initialising values for each component:

- allocate new object identifier and check for exhaustion
- page the new identifier to align pager tables, test for a clash at that slot and if necessary squeeze that object out to backing store
- at that set of table slots, and their output registers,
  - put the identifier into the object number table
  - put the object's size into the size table
  - put the object pointer into the address table
  - put the address of the first component into the memory address register
  - set the NEW flag
  - clear the MOD flag and tag flag
  - put the type into the type table
  - step up the object pointer to the end of the object and check that the object pointer has not advanced off the end of memory; call the garbage collector if necessary
- initialise each component from the stack
- replace the size, type and initialising values on the stack by the new object identifier

On the REKURSIV this sequence takes 5+2*N cycles, for N components, and it runs at 10MHz (later 16MHz).

EXAMPLE - Linear Scanning

Stack has object above symbol, replace with value or nil.

```
LSCAN:  decsp newesp
        d=estk pagebus readestk
        d=estk ldsym idx1 loadaddr

*1:     cjbr @*2 !IDXOK idxget
        cjbr @*1 !NILSYM d=memout idxup2 loadaddr

        cjbr @*2 IS_NIL decidx loadaddr d=memout
        idxget
        d=memout ldestk
        ...

*2:     d=brch (NIL) ldestk  /* not found */
```


EXAMPLE - Hashed Scanning

Stack has object, above symbol, above initial hashed index; replace with value or nil.

```
HSCAN:  decsp newesp
        d=estk pagebus readestk decsp newesp
        d=estk ldsym readestk
        d=estk ldidx loadaddr ldreg

        newmark idxget
        cjbr @*1 NILSYM d=memout incidx loadaddr
        jrp @*2 IDXDONE idxnext loadaddr

*1:     cjbr @*2 IS_NIL d=memout idxget
        d=memout ldestk
        ...

*2:     d=brch (NIL) ldestk  /* not found */
```

Sending a Message

- Stack has message, above arguments, above receiver
- get receiver's class
- if selector/class pair not cached, search for message following superchain and cache resulting selector/class/method relationship.

... so have method in object store

- if primitive, execute microcode body and continue with next instruction
- if not primitive or primitive fails :
- if method not in instruction cache, cache it
- establish new context and invoke its first instruction.

... so have new instruction in execution.

REKURSIV SMALLTALK INSTRUCTIONS

Linn Smart Computing Ltd
257 Drakemire Drive
Glasgow G45 9SN

Tel : (44) 41 631 1483
Tlx : 77301 LINN G
Fax : (44) 41 631 1486

Contents

The REKURSIV Smalltalk Instructions fall into several categories, mainly control flow and variable manipulation, and these will be generated by the compiler during interactive sessions. In addition, there are low level instructions for manipulating the stack and interfacing to the object store, but these and various high-level instructions for building the class structures, method and dictionaries of the kernel classes, are used during system startup (before messages can be sent).

Control Flow

Send	Send a message
Exit	Exit a method, return to sender's context
BlkExit	Exit a block, return to home context

Variables

Liv	Load instance/ class variable
Siv	Store instance/ class variable
Lv	Load method/ block variable
Sv	Store method/ block variable

Basic Object Creation

Integer	Load an integer
Character	Load a character
Pseudo	Load a pseudo variable
String	Load a string

Low Level Object Creation

Alloc	Allocate an empty object of a given type and size

Low Level Stack Manipulation

Push	Load binary quantity, untyped

Low Level Object Manipulation

Get	Load component of an object
Put	Store component of an object

Building Class Structures

Object
Class
Method
Dictionary

SEND <nargs> <selector>

SEND		nargs	selector

This instruction takes arguments on the stack, and looks up the method with the selector specified by the operand on the class of the receiver. The receiver is the first argument, so SEND always has at least one argument. If necessary, Send will follow the single-inheritance superchain off the receiver's class in order to find the method. If no method is found nil is returned.

The method will be either primitive, or abstract. In the case of a primitive, the appropriate microcode will be executed immediately, and the number of arguments and their types checked as appropriate to each operation. For an abstract method, it will be verified that there is the correct number of arguments, and that method will then be invoked.

If a primitive method fails, the stack will be reset, with the method for that selector on the stack above the arguments, and then the corresponding abstract method will be invoked. Usually such a 'failure' will merely report an error, but this mechanism can also be employed to cater for unusual cases for which the microcoded primitive form has no option.

When an abstract method is invoked all temporary variables are initialised to nil.

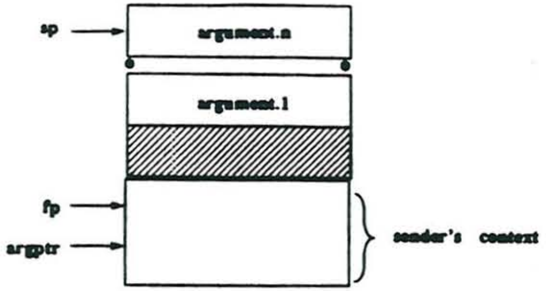
Notes :

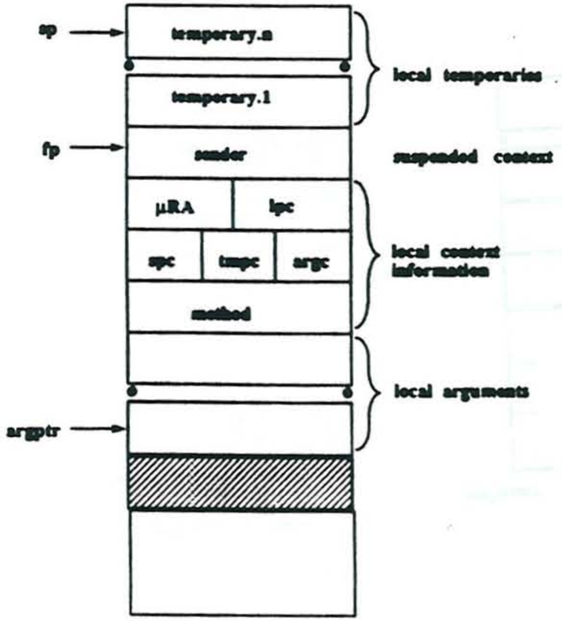
(1). Argument one is the receiver

(2). There are less than sixty four arguments allowed

(3). Selectors are sixteen-bit binary codes (these may well be converted into compact objects of type Selector within the machine).

SEND





ALLOC

ALLOC	

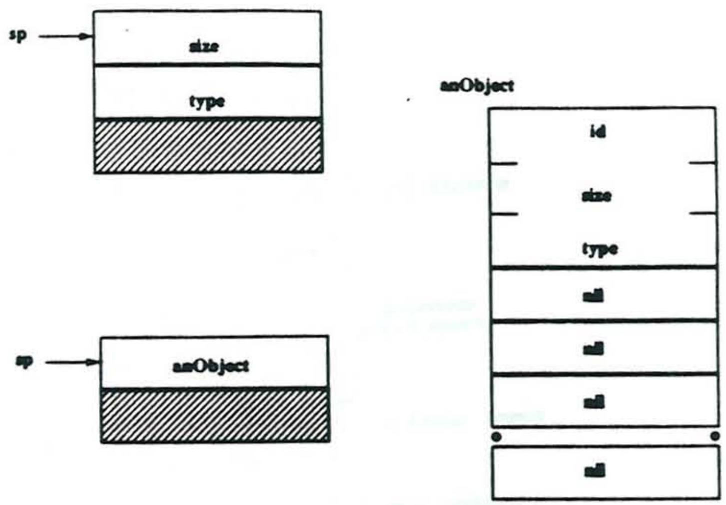
This instruction takes <size> above <type> on the stack and replaces them with an object of that type and size whose components are initialised to nil.

If the size is zero, a valid empty object is created. No provision is made to ensure that all empty objects of a given type are, in fact, unique instances of that type of object.

Notes :

(1). This instruction is used during the bootstrap, for example to create the Arrays for class variables of metaclasses.

ALLOC



PUSH <binary value>

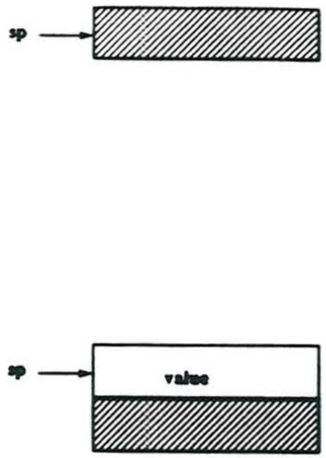
PUSH		binary value

This instruction simply pushes its 24 bit binary operand onto the stack.

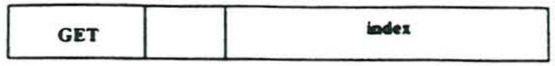
Notes :

(1). No type is assumed, just a 24 bit binary quantity.

PUSH



GET <index>



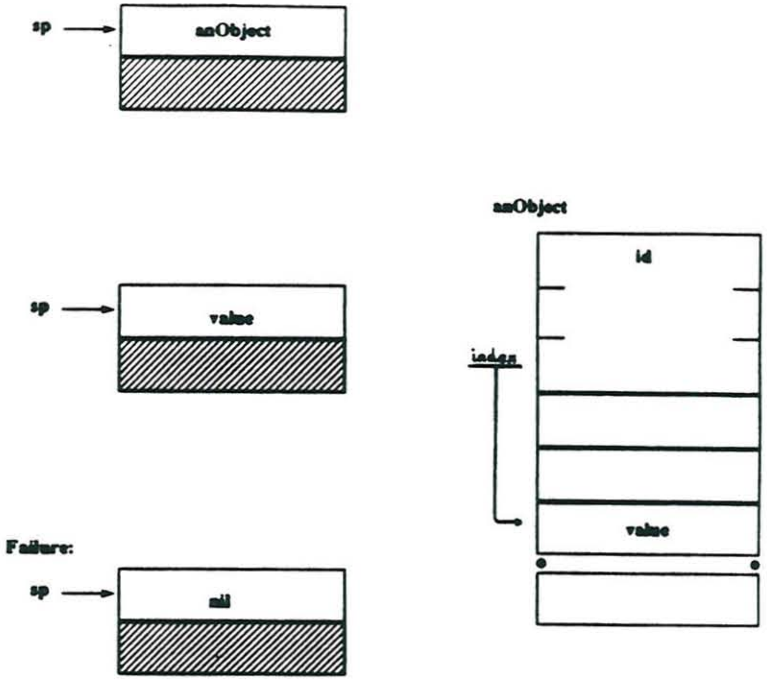
The object on top of the stack is replaced with the component whose offset is specified by the operand. If the index is invalid, nil is pushed.

Notes :

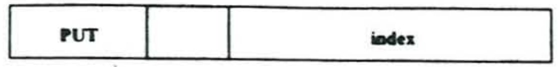
(1). If the index is zero, the type of the object will be pushed onto the stack.

(2). This instruction provides a low level interface to the object store and is used during the bootstrap.

GET



PUT <index>



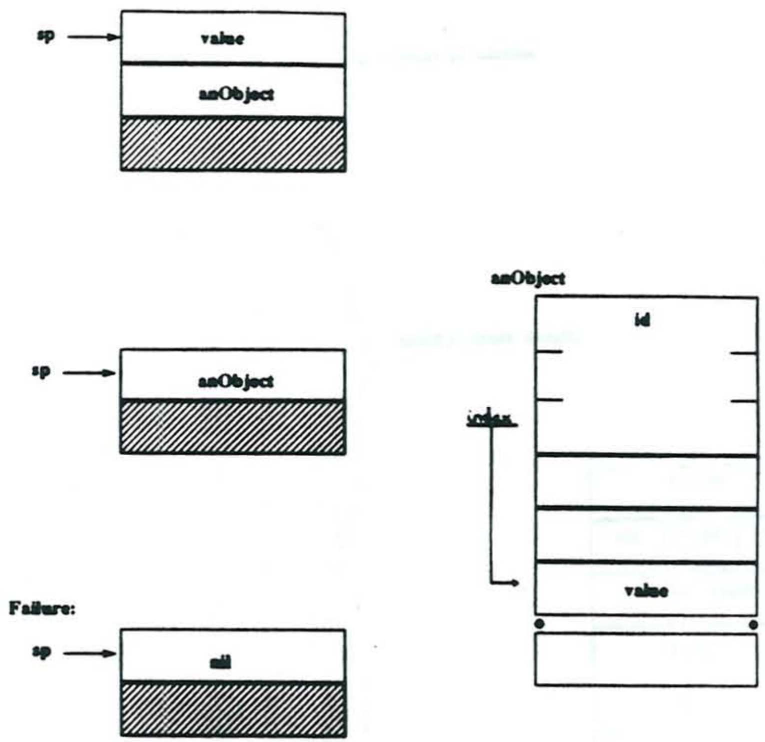
The stack holds the replacement value above the container object, and this value is written into the container object at the index specified by the operand. The result is the object, or is nil if the index is invalid.

Notes :

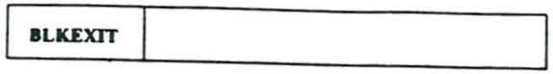
(1). It is not possible to write to the component at index zero (the type of an object).

(2). This instruction provides a low level interface to the object store.

PUT



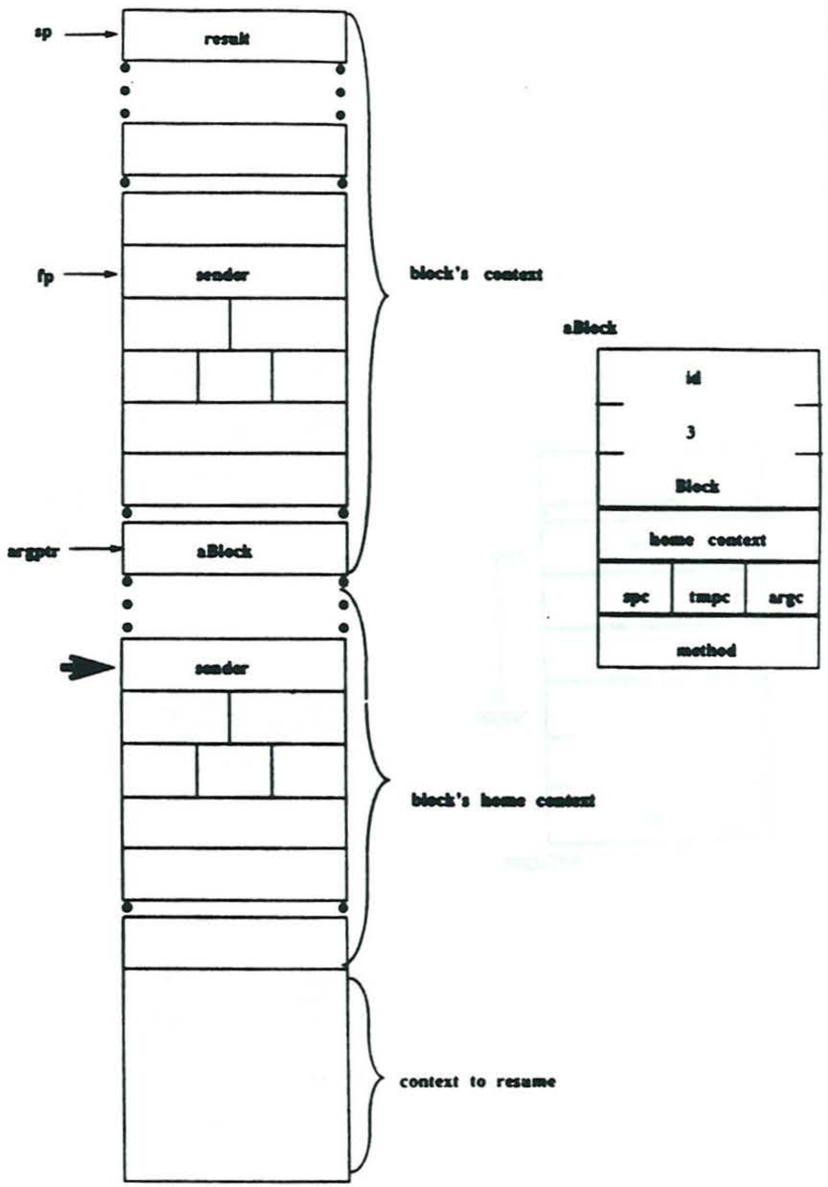
BLKEXIT



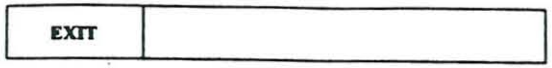
This instruction causes the current context, which is a Block's context, to cause its 'home' context to EXIT.

This can work only if the home context has not already EXITed.

BLKEXIT

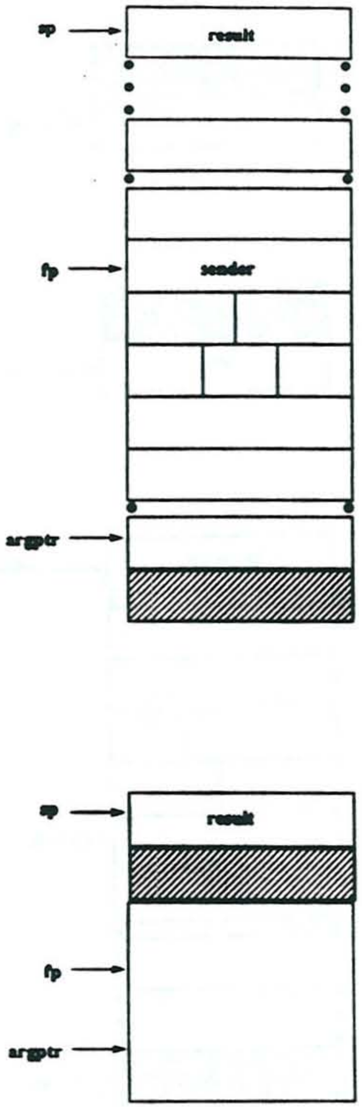


EXIT



This exits from a block or a method, returning to the suspended 'sender' context. If necessary the resumed context's method code will be re-cached.

EXIT



LIV <mode><offset>

LIV		mode	offset

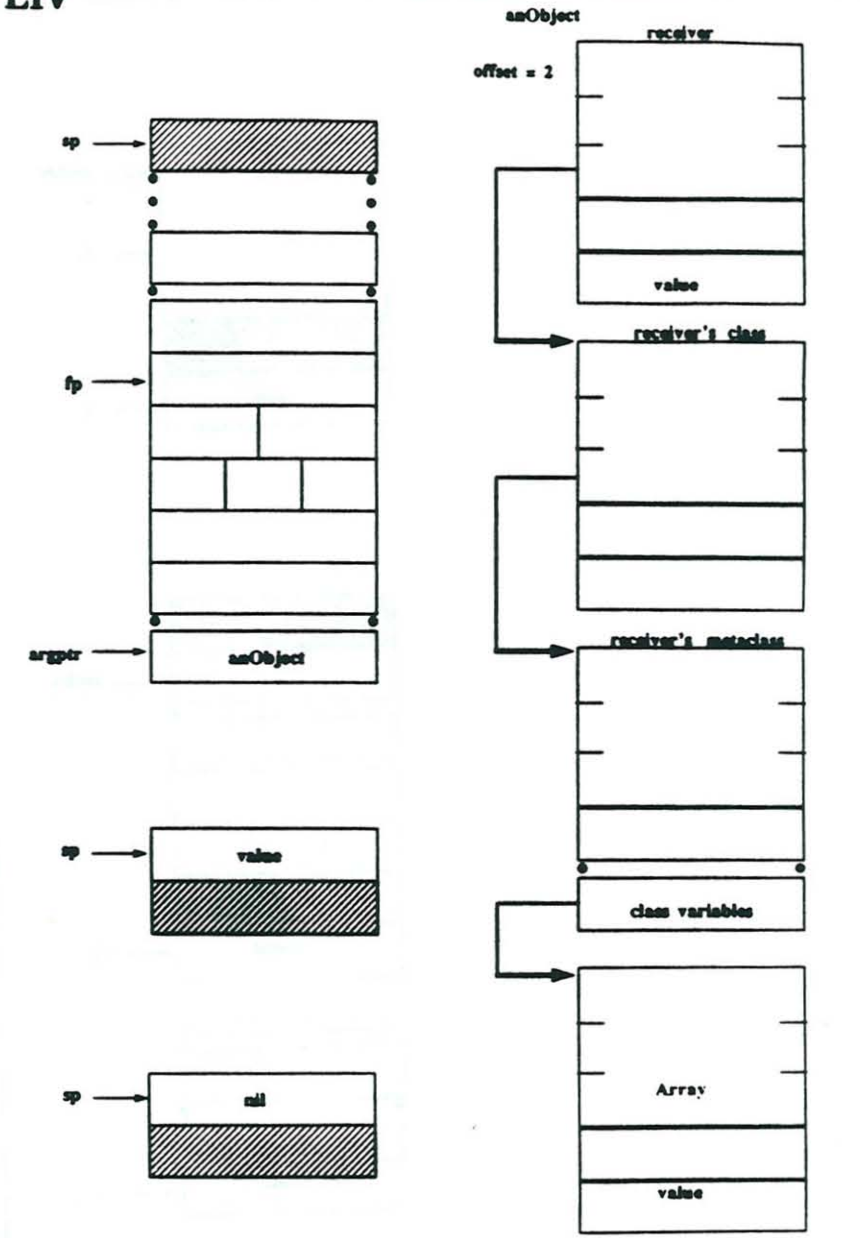
If the mode is set, lowly classed variable from the metaclass of the receiver, otherwise load onto the stack the instance variable at the offset specified by the operand into the current receiver. If the offset is invalid, nil is returned. Components of the receiver start at offset one.

Notes:

(1). Although the machine allows it, it has been deemed poor practice for methods of one class to directly access instance variables of classes in the superclass chain, so the compiler ensures that the instance variables of a class can be accessed directly only by methods of that class.

(2). Class variables are held in an Array as part of the metaclass of the receiver.

LIV



SIV <mode> <offset>

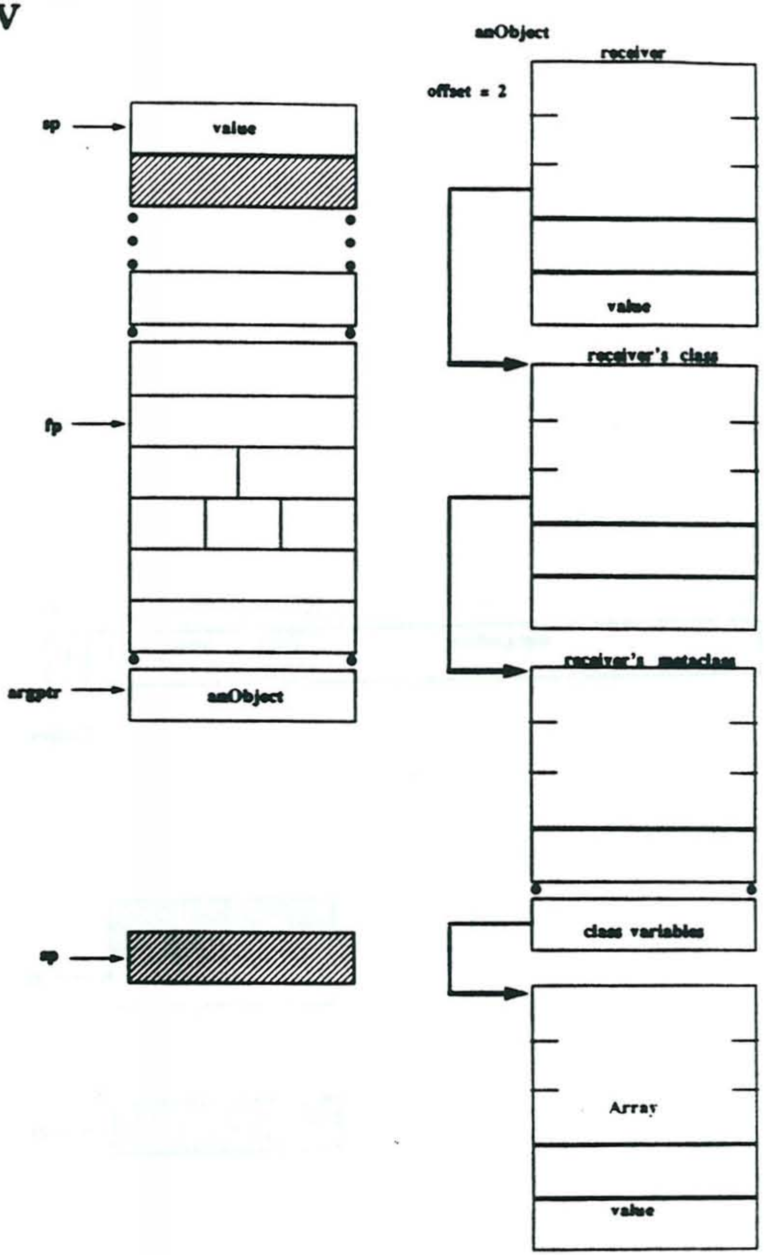
SIV		mode	offset

If the mode is set, store (and pop) into a class variable otherwise store (and pop) the stack top into the instance variable at the offset specified by the operand into the current receiver. If the offset is invalid the update operation is abandoned. Components of the receiver start at offset one.

Notes :

(1). Class variables are held in an Array as part of the metaclass of the receiver.

SIV



INTEGER <binary integer>

INTEGER		binary integer

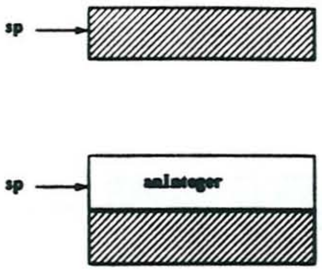
Load onto the stack an Integer object whose value is specified by the 24-bit operand.

Notes :

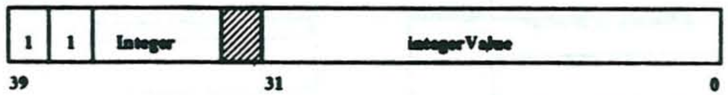
(1). Integers are represented as compact objects, with 32 bit data tagged to 40 bits.

(2). There will probably need to be a way of creating 'big' integers, with 32 bits.

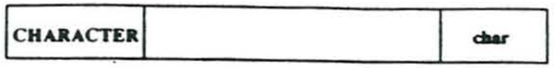
INTEGER



anInteger



CHARACTER <character>



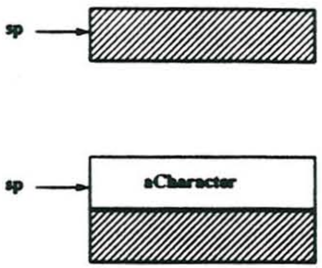
Load onto the stack a Character object whose value is specified by the 8-bit operand.

Notes :

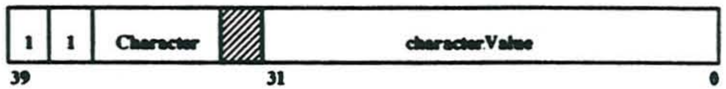
(1). Characters are represented as compact objects, with 8-bit data tagged to 40 bits.

(2). If necessary, font and scale information can be incorporated into the upper bytes of the character's representation.

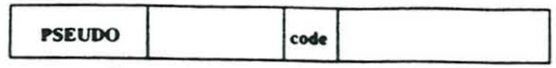
CHARACTER



aCharacter



PSEUDO <code>



This instruction loads one of a variety of special values onto the stack, as specified by the operand.

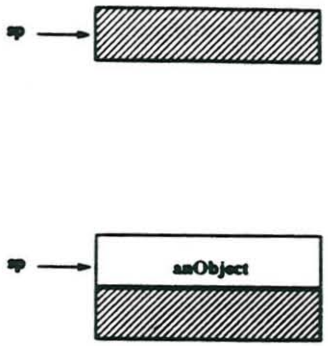
case <code> of

1	true
2	false
3	nil
4	receiver
5	context
6	method
7	objects
8	super

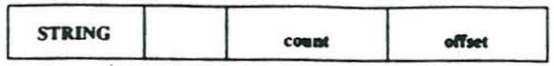
Notes:

(1). All of these quantities are represented as objects, not binary quantities, so when loaded onto the stack they are tagged appropriately.

PSEUDO



STRING <count> <offset>



This instruction will cause a String object to be loaded onto the stack. The string is derived by indexing into the current method, at an offset specified by the operand, to extract the string's object identifier. If this is nil, the string must first be built up from the literal in the method. In this case, the number of characters packed into the words that follow is given by <count>. These start at the word after that given by the <offset>.

A string object is built using this literal, and the result written both on to the stack and into the method at the specified offset. Thus, only string literals which are actually used get created as objects, and once created for a given method are thereby cached by it.

Notes:

(1). The characters are packed running down the word, first at the highest byte.

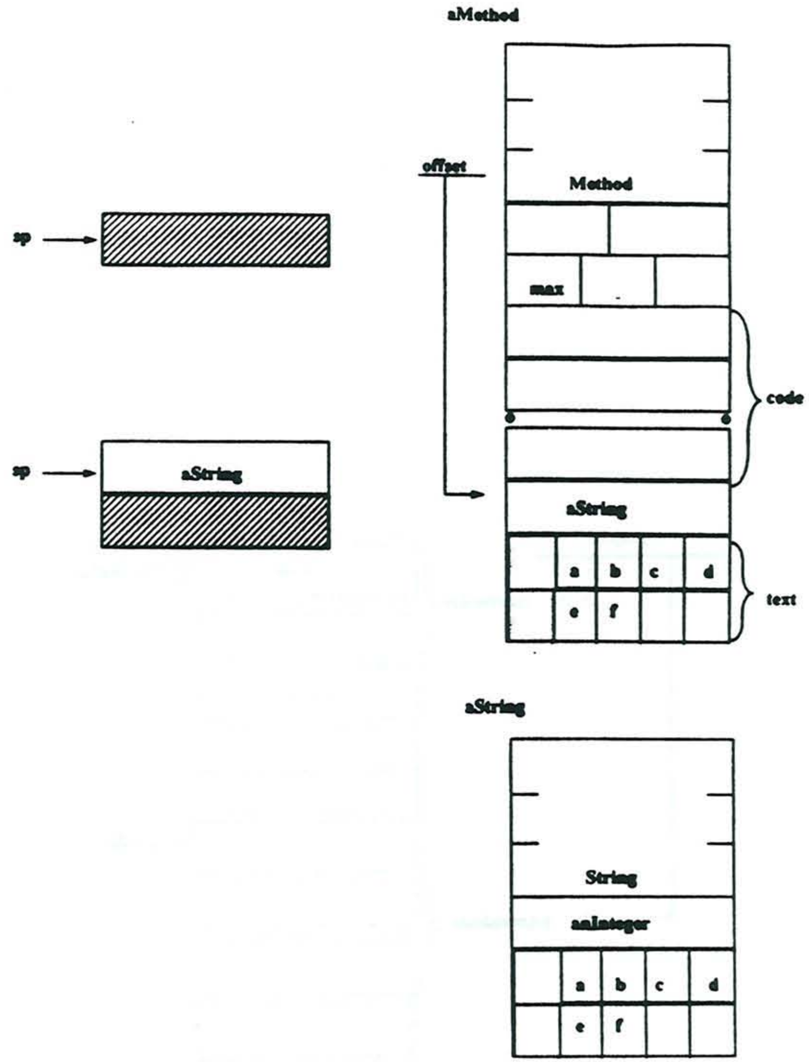
(2). Trailing bytes after the last character in the final word are null padded, to nil.

(3). Characters in a String object start at index one.

(4). The longest String literal has 4K characters.

(5). The character count of a String is an Integer, so dynamically generated strings may grow very large!

STRING



LV <level> <offset>

LV		level	offset

The value of the block or method variable (argument or temporary) at the offset specified by the operand, within the context at a level down the static chain specified by the operand, is loaded onto the stack.

Notes:

(1). The current lexical level is zero.

(2). The first variable is at offset zero.

(3). Blocks can access method variables by reaching down the static chain one level beyond their nesting depth in that method.

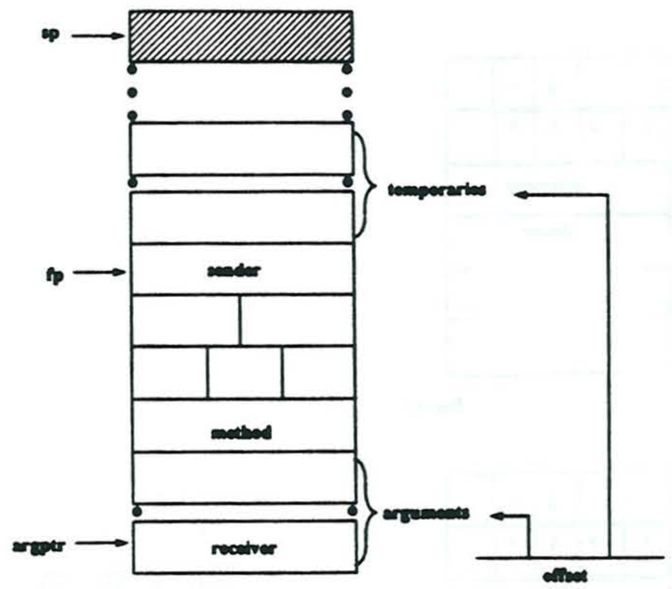
(4). Since this instruction is generated by the compiler, it is assumed that the depth level down the static chain, and the offset into the resulting context, are valid. No checks are made to ensure that this is so.

(5). Arguments reside at the base of each context, tempories are accessed as if they were arguments, at offsets above the actual argument (allowing a space for the context's method and linkage information).

(6). There is a maximum of fifteen lexicographical nesting levels.

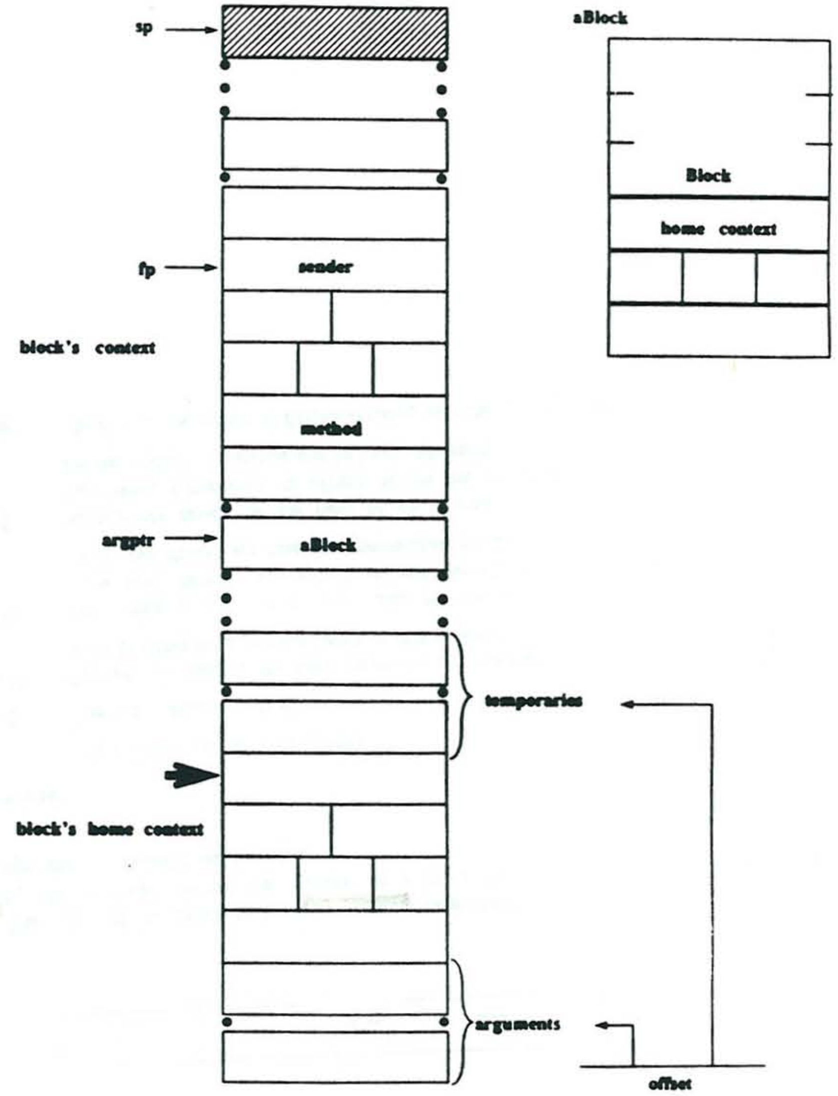
LV

Local variables (level=0)



LV

Block variables one level back from current block's context (level =1)



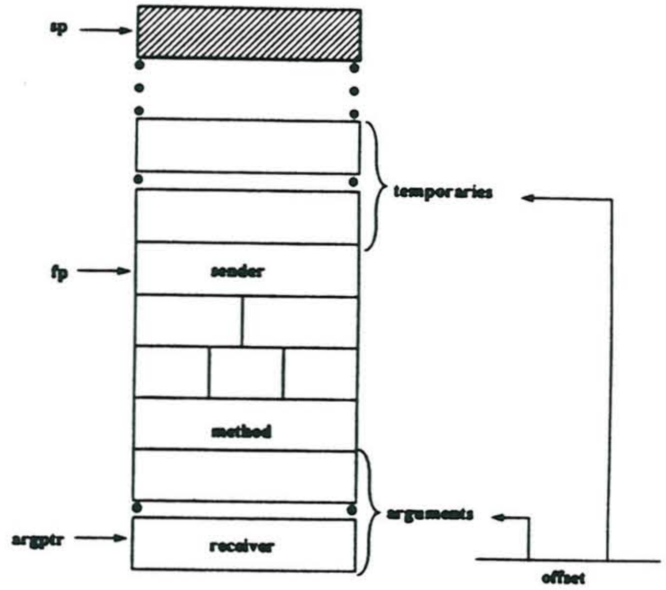
SV <level> <offset>

SV		level	offset

The top of the stack is stored into the variable (argument or temporary) at the offset specified by the operand, within the context at a level down the static chain specified by the operand. The stack is cleared.

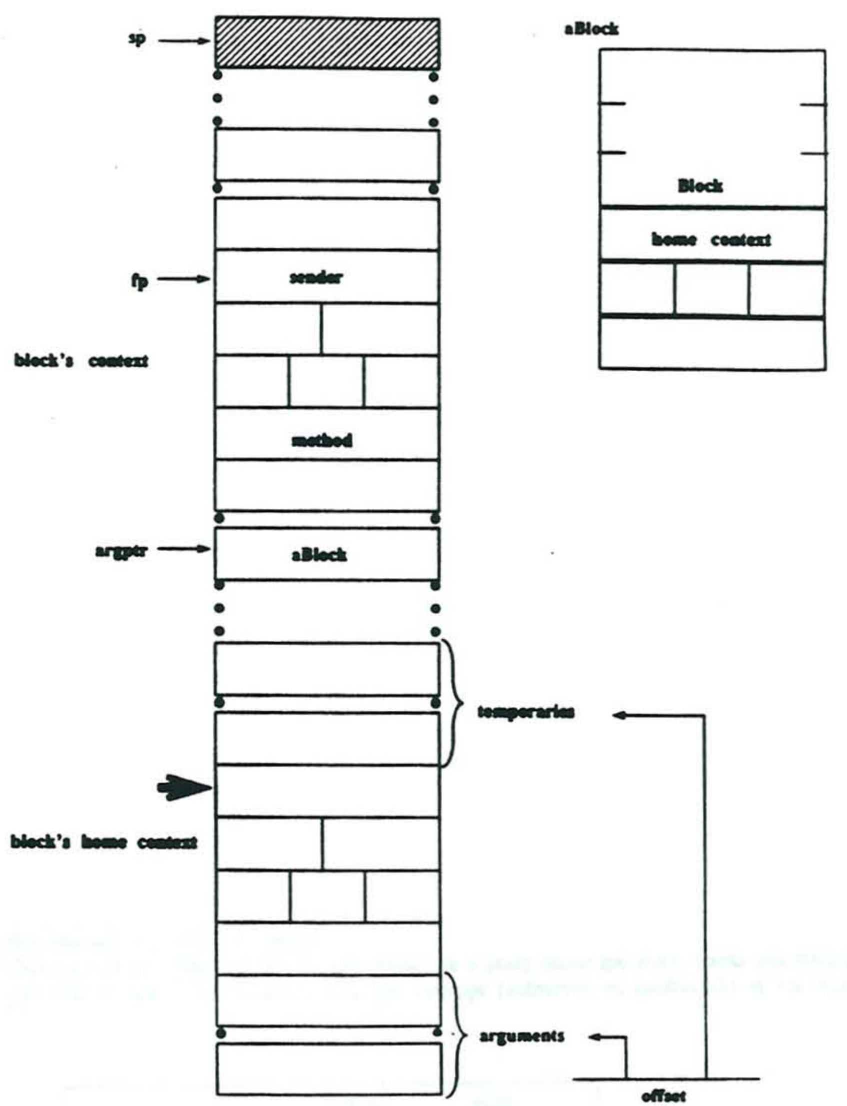
SV

Local variables (level=0)



SV

Block variables one level back from current block's context (level =1)



DICTIONARY <count>

DICTIONARY		count

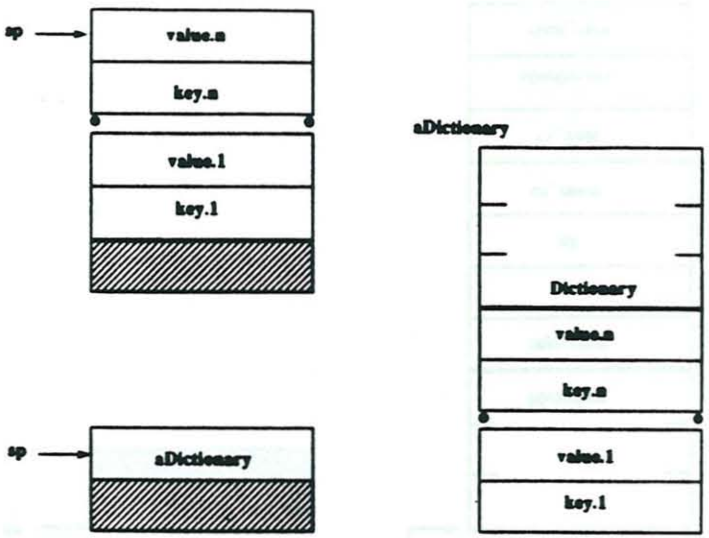
The stack contains <count> pairs; replace them with a Dictionary initialised with these key/value bindings. The size of the dictionary will be twice the count.

Note:

(1). A dictionary is always even-sized, it holds pairs, key bound to value. The key and value may be anything.

(2). This instruction is used during the bootstrap.

DICTIONARY



CLASS <cvarc> <objid>

CLASS		cvarc	objid

The stack has a dictionary above the instance variable base offset, the instance variable count, the class name, a nil, the superclass and the class for the desired new class. If there are any class variables to be made, it must be a metaclass, so make an Array <cvarc> big and push this onto the stack. Make a Class of the appropriate size, replacing its components on the stack with the class's object identifier. The new Class object will have either the next available object number or the identifier specified by <objid>, to create a kernel class.

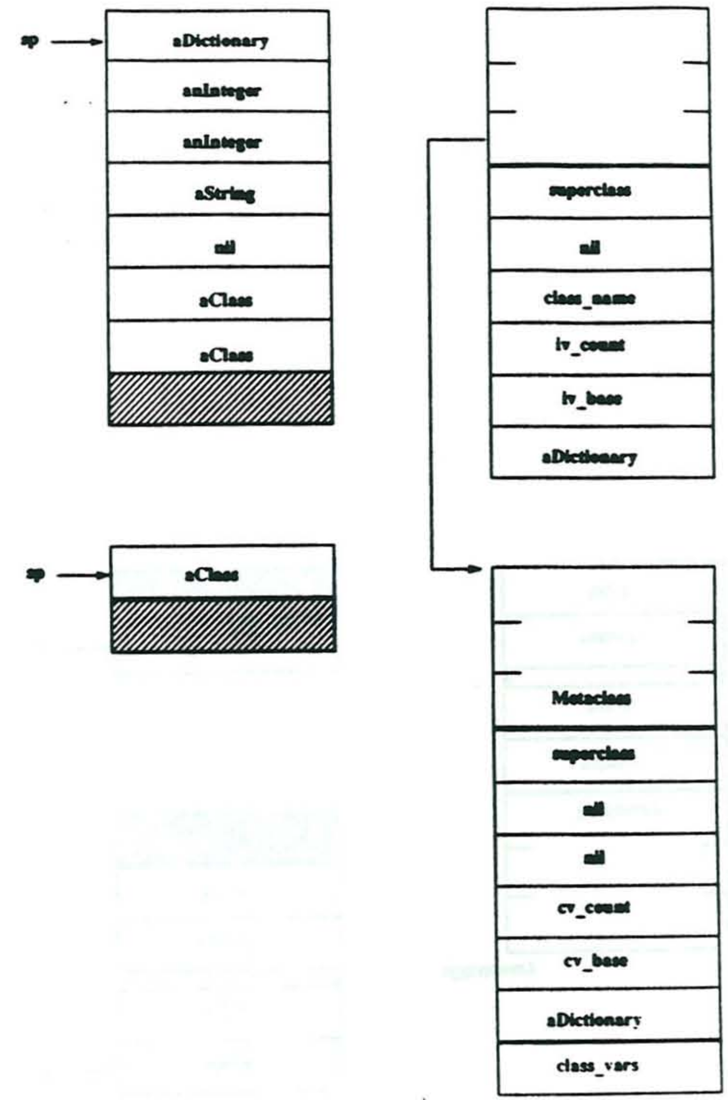
Note:

(1). Only metaclasses have the "classvars" component, which is an Array holding the class variables for the class of which this is the metaclass.

(2). This instruction is used during the bootstrap.

(3). There is a limit of 15 class variables per metaclass.

CLASS



METHOD <count>

METHOD		count

This sixteen-bit operand specifies the number of codewords following the instruction. A Method object of size <count> plus two is created. This is then initialised from the codestream and the instruction pointer advanced to the next instruction.

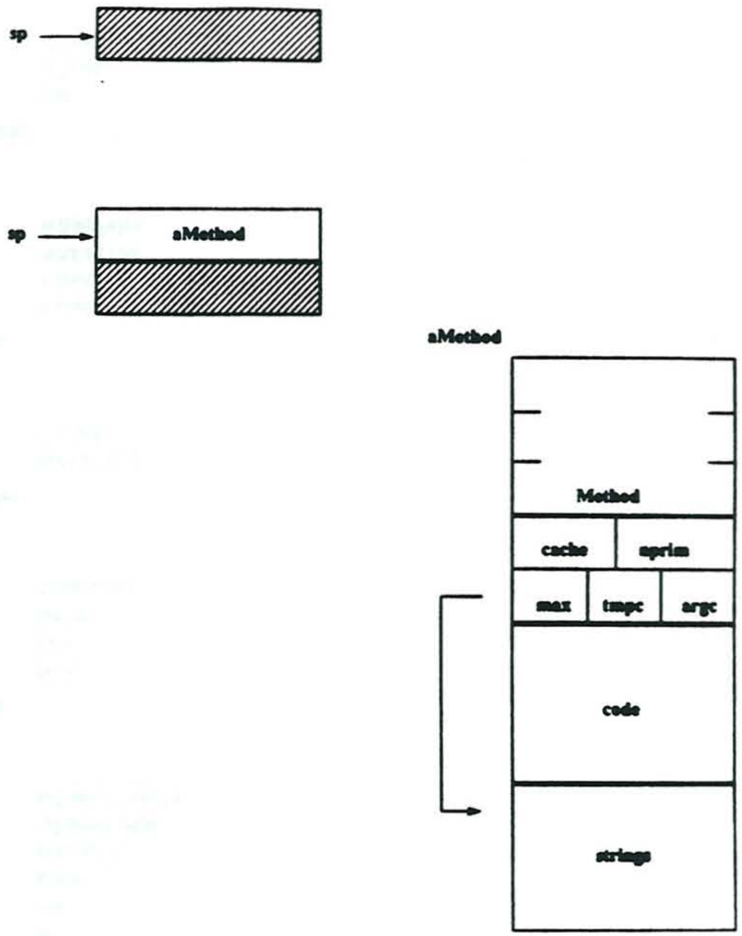
The resulting method is pushed on the stack.

Notes :

(1). This instruction is used during the bootstrap.

(2). All methods are compiled into the REKURSIV Smalltalk Instructions and stored in the object store as objects of type Method. Their identifiers are bound to their selectors in the message dictionaries of their parent class. When a method is invoked it is cached into the instruction cache within the processor's pipeline. Methods are cached as they are required. The cache is flushed when it overflows, and dynamic re-caching is recommenced. REKURSIV Smalltalk instructions are designed to be relocateable by virute of being position-independent. There are no jump instructions, there are no addresses, so caching is efficient.

METHOD



PRIMITIVES

The list of microcoded Smalltalk primitives includes:

String

- at:
- at:put:
- size

List

- car
- cdr

KeyedCollection

- find
- bind

Integer

- <
- <=
- >
- >=
- timesRepeat:
- validIndexOf:
- +
- -
- *
- /
- %

Array

- length

SequenceableCollection

- at:
- at:put:

Object

- classOf
- size
- isNil
- notNil
- hash
- =
- ~=
- cons:
- isKindOf:
- isMemberOf:
- isSameClassAs:

Class

- new
- new:
- name
- superclass

Context

- blockCopy
- sender

Block

- value
- value:
- whileTrue:
- whileFalse:

Boolean

- not
- ifTrue:
- ifTrue:ifFalse:

REKURSIV Smalltalk Instruction Set

SequenceableCollection

- at:
- at:put:

Object

- classOf
- size
- isNil
- notNil
- hash
- =
- ~=
- cons:
- isKindOf:
- isMemberOf:
- isSameClassAs:

Class

- new
- new:
- name
- superclass

Context

- blockCopy
- sender

Block

- value
- value:
- whileTrue:
- whileFalse:

Boolean

- not
- ifTrue:
- ifTrue:ifFalse:

Linn Smart Computing: Tel (44) 41 631 1483 *August 1988*

STATISTICS

The instructions are used in different ways depending upon algorithm, however because there are so few instructions available there is relatively little scope for variation - all that a string processing application is likely to do to distort the figures is change the String and Character instruction counts, for example. The only major distortion of instruction counts is likely to be caused during system bootstrap, when the class building instructions are used, before there are sufficient classes to support message passing operations.

Static Statistics

The static instruction counts for the example used in the following test are :

```
        23          pseudo
        16          send
        12          integer
50%--------
        10          exit
60%--------
        8           lv
70%--------
        6           push
        6           method
80%--------
```

Dynamic Statistics

Taking the bootstrap alone we get dynamic frequencies as follows :

```
        21.8        pseudo
        17.4        push
        17.3        method
60%--------
        14.7        integer
70%--------
        11.0        object
80%--------
        7.2         class
90%--------
```

If we now include a small amount of "interactive use" we get

```
        19.8        push
        17.1        pseudo
        12.3        integer
```



```
50%--------
        10.5        send
60%--------
        9.5         lv
70%--------
        9.2         method
80%--------
        5.0         object
        4.6         exit
90%--------
```

so clearly half the instructions executed are various kinds of stack-push of trivial quantities. Message sending and variable access are only beginning to become significant. The bootstrap still dominates, however.

Running a longer test eliminates the effect of the bootstrap, viz

```
        21.1        push
        19.1        send
        17.7        lv
60%--------
        12.2        pseudo
70%--------
        10.1        integer
80%--------
        8.5         exit
90%--------
        2.6         sv
        1.3         siv
```

which doubtless begins to show a realistic working mix.

By optimising Send to carry its selector as an operand in the instruction, rather than having it first placed on the stack by Push, the following counts obtain

```
        23.6        send
        21.8        lv
50%--------
        15.1        pseudo
60%--------
        12.4        integer
70%--------
        10.5        exit
80%--------
        3.2         sv
        2.7         object
        1.6         siv
90%--------
```

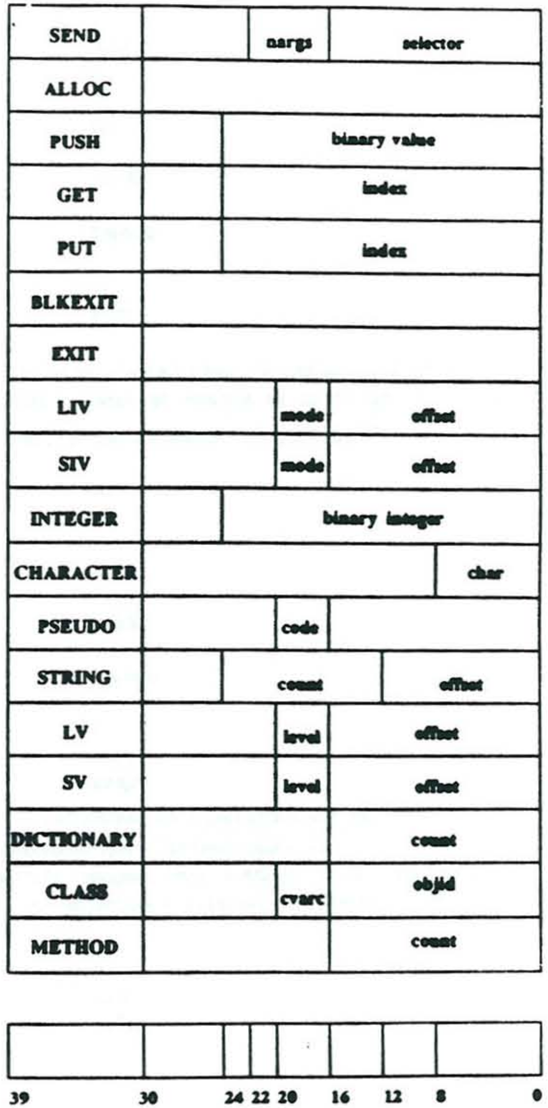
The Push instruction is still used, during method construction, but has been relegated to that set of instructions which get called upon only during system creation.

The execution times for each instruction, accumulated during the test, excluding the bootstrap, were as follows :

	mS	%time	%opcodes
send	13.94	61	23.6
exit	3.99	17	10.5
lv	3.24	14	21.8
sv	0.64	2	3.2
pseudo	0.27	~1	15.1
integer	0.22	~1	12.4

From these figures it is clear that six instructions consume 99% of the time, and two of these nearly 80%, these being Send and Exit. These same six instructions comprise some 90% of the dynamic opcode counts. It is clear, therefore, that message passing in this Smalltalk instruction set is the main activity. It should be noted, however, that in the time allotted to message sending, the Send instruction includes the time spent within a primitive if that message was implemented directly in microcode. Since 80% of messages sent in the example were primitive, it is not surprising that Send is the dominant instruction.

Instruction Formats



General Notes

(1). The stack is pre-incremented, so always points at the topmost element. The registers are :

fp
sp
argptr

and these are stack addresses, not objects.

However, there are also registers in object format :

current context
current receiver
current method

for fast access.

(2). Garbage collection is automatic.

(3). Execution times?

(4). There are no addresses, no jumps, position-independent code, methods are cached in.

(5). Instructions needed only during bootstrapping can thereafter be deleted from the control store map and thence rendered unavailable leaving only the message-based instructions.

THE MESSAGE AND METHOD CACHES

Message lookup, method invocation and method execution are all assisted by the existence of caches.

That is, the instruction cache is loaded only with those methods which get invoked. These are loaded dynamically, when needed. This cache is within the processor architecture, close to the sequencer, so that opcode decoding and operand stripping can be pipelined.

Each method so cached is tagged with the start address of its codestream in the instruction cache. This gets set when the method's codestream is first loaded into the cache. A simple examination of a method reveals both whether or not it has been cached, and identifies its start point in the cache in the case that is has.

The instruction cache is filled on a first-come basis, so when it overflows all methods which point into it are reset to indicate that they are no longer cached, and the cache pointer reset to the base of the cache. To facilitate this unlinking, an array of method identifiers is kept, and this is automatically scanned when the cache is cleared and each method identified by it has its cache address set to zero, to indicate not-cached.

Thus, given a method, it can quickly be established that it has been cached and, because of being 'locked into' the processor's sequencer, it can executed very efficiently. Clearly, therefore, because it is truly a method cache, there will be no page faults from the codestream during execution of a method, and so no need for disk access. The instruction cache is quite large, up to 128K instructions, and because methods tend to be fairly short, a few dozen instructions, there can be many methods in the cache at any given time. To remove the possibility of the cache management table causing a premature flush, the cached-method table grows automatically when necessary.

It is in the nature of message-based systems that much time can be spent merely searching the message dictionaries associated with any given item of data, trying to find the meaning of a particular message. This could involve searching the entire class hierarchy, to find the proper method. An optimisation over repeatedly searching the class hierarchy is to maintain a cache that records which method was found each time a particular pair of selector and receiver class are looked up. If that message has been sent to that class of data before, the cache identifies the appropriate method, so a full search is not necessary.

The message cache is organised as a triple. It records a binding between a selector, a class and a method. Given the selector and the class, it provides the proper method. A straightforward hashing algorithm is employed, using the low order bits of the selector and the class to provide a cache index. If the selector and class match those of the cache, the method is extracted. If they do not match, the method is found by searching the class hierarchy of the receiver for that message and the message, class and method are then written to the cache for future reference. This cache can grow if necessary

The access time for the message cache is half a dozen cycles, to establish that there is no match, with a further three cycles to extract the method should a match occur. This is likely to be far faster than searching even the first-level class's dictionary.

Once the desired method has been identified, it then takes only half a dozen more cycles to establish whether its codestream has been loaded into the instruction cache, and to start setting this up in the pipeline.

Performance

The message cache performed, in small scale tests using different hashing algorithms, as follows

hash	%slots	%hits	cycles	µS
1	13	81	154299	22603
2	12	81	155172	22814
3	11	78	162370	23744
0	0	0	161207	24198
4	5	47	169733	25170
5	4	50	171657	254[illegible]

(The cache had 255 slots, and 1903 messages sent during the test).

for hashing algorithms

0	none
1	(class XOR message) $_{0..8}$
2	(class ADD message) $_{0..8}$
3	(class $_{0..3}$ << 4) | message $_{0..3}$
4	((NOT class) AND message) $_{0..8}$
5	(class AND message) $_{0..8}$

where the result is always ORed with one to guarantee a valid (non-zero) index.

The favourite algorithm is therefore the XOR of the message and receiver class. Those algorithms which performed poorly were actually disadvantageous, presenting an overhead rather than an optimisation. It should be noted, however, that these tests were carried out on a very small execution profile, some 25 milliseconds during which less than 2000 messages were sent; longer tests on a much larger system will be needed to properly evaluate the benefits of the message cache.

DISCUSSION

The discussion began by Dr. Kay asking Professor Harland how small he hoped to make the Recusiv board. Professor Harland replied that their aim was to reduce the size to that of a normal VME card, with double eurocard connections. Smaller that this was unlikley, and the target was to achieve this reduction by next year.

Professor Randell asked whether the essence of the Rekursiv architecture could be summarised as being the ability to execute tests and conditional branches in parallel. Professor Harland replied that this capability was only one of the features of the architecture.

Professor Morrison asked Professor Harland how confident he thought his type system was for representing all types, as concern was expressed about the size of the available tags. Professor Harland replied that the tags could also be a word (in addition to the 5 bits used by the compact types), and he thought that would be sufficient. Professor Atkinson followed up on the previous question, being worried about the type system not supporting persistence data fully. Professor Harland replied that Rekursiv does not support a type system, but such problems could be solved by building suitable tools such as a browser to browse the objects in the object store.

A member of the audience pointed out that the buyers of computer hardware are willing to pay extra for faster processing, but questioned whether they are also willing to pay extra for the security of the type system supported by the Rekursiv architecture. Professor Harland disagreed, pointing out that the type security supported by the architecture was becoming a requirement, in particular for military use, and that to date 17 machines had been sold. In reply to a question asking whether these machines had been bought by military users, Professor Harland replied that none had been bought by the miltary. The problem with buyers such as the military being that they take such a long time to think about buying a product that once decided, then the product is already obselete.

Professor van der Poel asked where the names Rekursiv, Objekt etc came from. Professor Harland replied that the company (Linn Products) make hi-fi, and have a habit of misspelling names, so that when the marketing people were consultated as to what to call the architecture, the fact that it suppoprts recusive computations in the micro-code suggested that it be called Rekursiv, with the name clearly being a misspelling of recursive. This theme continued with the other components of the architecture.

Professor Randell asked whether a number of the machines could be used together. Professor Harland replied that this was one of the aims of the group, and that his main interest was now in constructing a distributed object store that could be shared by a number of machines. One possibility being to use 6 bits of the object id to name a particular machine. Professor Harland though that this was an interesting problem, with lots of tricky problems. Professor Atkinson stated that a similar approach had been adopted by the IBM model 38 architecture, but there larger word sizes had been used. Professor Harland replied that he throught his approach would be sufficient to construct a distributed object store.