Appendix A Object Formats

This appendix details the representation of objects used in the SELF memory system. Section 6.1 presents an overview of the SELF memory system.

A.1 Tag Formats

A SELF memory space is organized as a linear array of aligned 32-bit words. Each word contains a low-order 2-bit tag field used to interpret the remaining 30 bits of information. A reference to an integer or floating point number encodes the number directly in the reference itself. References to other SELF objects and references to map objects embed the address of the object in the reference (there is no object table). The remaining tag format is used to mark the first header word of each object, as required by the scanning scheme discussed in section 6.1.2. Pointers to virtual machine functions and other objects not in the SELF heap are represented using raw machine addresses; since their addresses are at least 16-bit half-word aligned the scavenger will interpret them as immediates (either integers or floats) and so will not try to relocate them.

31	2	0
30-bit signed integer		`00`

integer immediate (or virtual machine address)

31	2	0
top 30 bits of word-aligned address		`01`

reference to SELF heap object

31	2	0
30 bits of IEEE floating point number		`10`

floating point immediate (or virtual machine address)

31	2	0
scavenging fields and hash field		`11`

mark header word (begins SELF heap object)

These tag formats were chosen to speed common operations on references. Tagged integers may be added, subtracted, or compared directly, as if the integers weren't tagged at all (since the tag field is zero for integers). Overflow checking is also free of additional overhead since the tag bits are low-order and overflows on the tagged format are detected by the hardware just as would be overflows on normal untagged integers. Integer multiplies and divides require an extra shift instruction prior to invoking the corresponding untagged operation. Other conversions between tagged and untagged integers require only an arithmetic shift instruction. Another nice benefit of this integer tag format is that object array accesses may use the tagged index directly to access the elements of the array; no extra multiplies or shifts are required to convert the index into an offset as would be required for untagged integers in a traditional language.

The tag format for references to heap objects is also relatively free of overhead. Since SELF objects always begin on word boundaries, on byte-addressed architectures the 2-bit low-order tag format does not reduce the available address space (the **`01`** tag is simply replaced with a **`00`** tag to turn a tagged reference into a raw word-aligned address). Additionally, on machines with a register-offset addressing mode, the tag can be stripped off automatically when accessing a field within the referenced object at a constant offset by folding the decrement into the offset in the instruction. For example, to access the n^{th} word in an object (origin 0), where n is a compile-time constant, the compiler can simply generate (in SPARC assembly syntax)

```
ld [%object + (n*bytes_per_word - 01)], %t
```

where the offset in the load instruction is a compile-time constant.

Testing whether a reference is an integer immediate requires a simple

```
andcc %object, #3, %g0
bz _integer
```

sequence to check the two low-order bits for a **`00`** tag. Testing for a heap object reference also only requires a

```
andcc %object, #1, %g0
bnz _heap
```

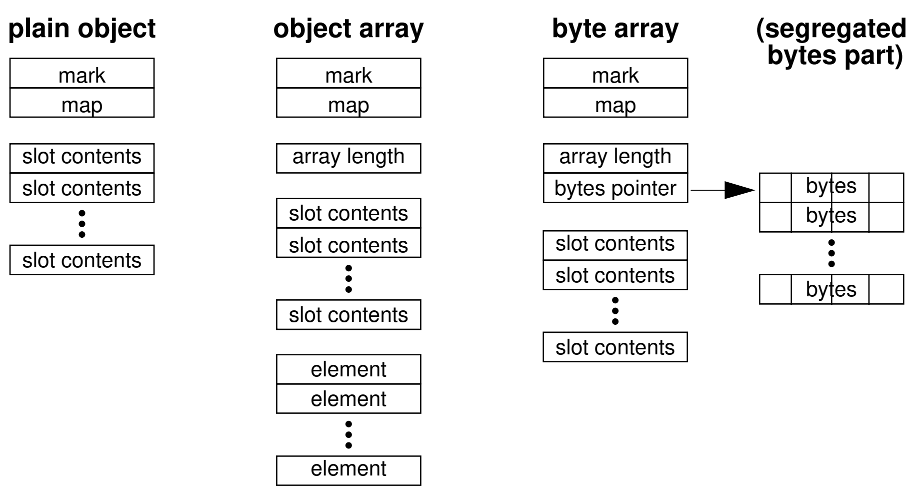
sequence to test whether the low-order bit is non-zero; this sequence assumes that the reference cannot be a mark word, which is the case for all object references manipulated by user programs. Testing for a floating point immediate similarly only requires a

```
andcc %object, #2, %g0
bnz _float
```

sequence to test the second-to-lowest-order bit (again, assuming the reference cannot be a mark word).

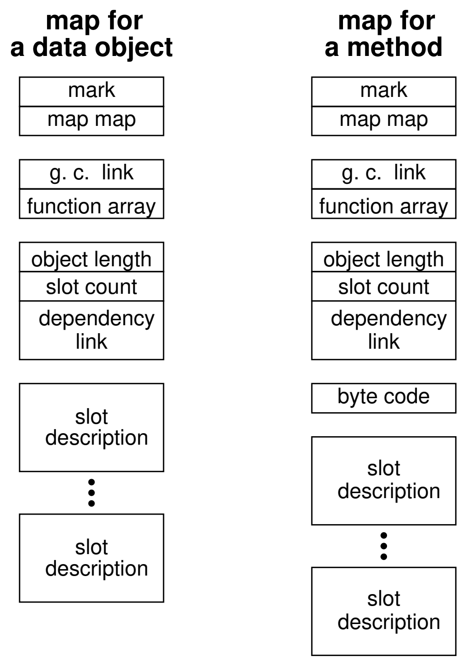
A.2 Object Layout

Each heap object begins with two header words. The first word is the mark word, identifying the beginning of the object. The mark word contains several bitfields used by the scavenger and a bitfield used by the SELF **`_IdentityHash`** primitive. The second word of an object is a tagged reference to the object's map. A SELF object with assignable slots contains additional words to represent their contents. Array objects include their length (tagged as a SELF integer to prevent interactions with scavenging and scanning) before any assignable slot contents. Object arrays include their elements (tagged object references) after any assignable slot contents (afterwards so that assignable slot contents have constant offsets within a given clone family), while byte arrays include an untagged pointer to a word-aligned sequence of 8-bit bytes before any assignable slot contents (segregation of packed bytes parts is described in section 6.1.2).

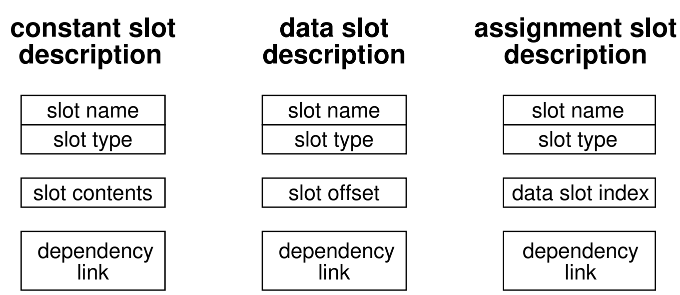


Like other objects, map objects begin with mark and map words. All map objects share the same map, called the *map map*; the map map is its own map. All maps in new-space are linked together by their third words; after a scavenge the system traverses this list to perform special *finalization* of inaccessible maps. The fourth word of a map contains the virtual machine address of an array of function pointers; these functions perform type-dependent operations on objects or their maps.[*]

For maps of objects with slots, the fifth word specifies the size of the object in words. The sixth word indicates the number of slots in the object and the number of slot descriptions in the map. The next two words contain the change dependency link for the map, as described in section 13.2.2. These four words are tagged as integers. If the map is for a method, the ninth word references the byte code object representing the method's source code (byte code objects are described in section 6.2).

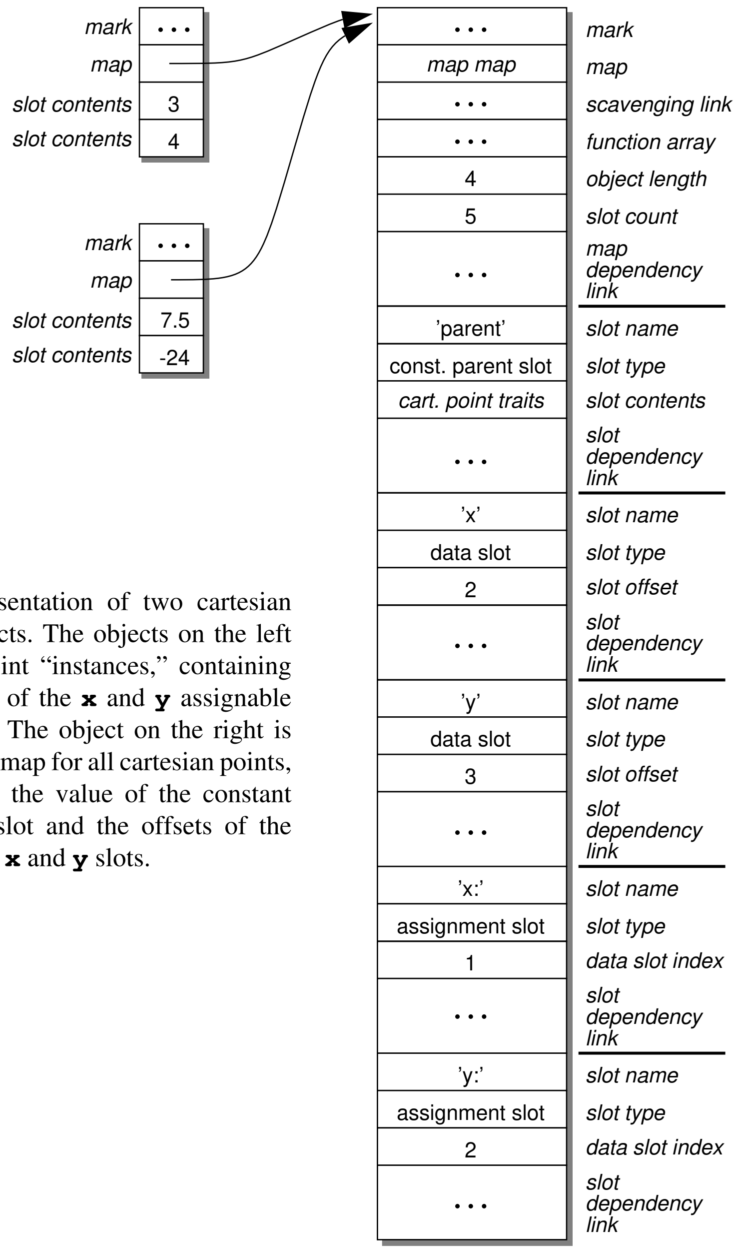


Finally, the map includes a five-word description for each of the object's slots. The first word points to a SELF string object representing the name of the slot. The next word describes both the type of the slot (either constant data slot, assignable data slot, or assignment slot) and whether the slot is a parent slot.[**] The third word of a slot description contains either the contents of the slot (if the slot is a constant slot), the offset within the object of the contents of the slot (if the slot is an assignable data slot), or the index of the corresponding data slot description (if the slot is an assignment slot). The last two words of each slot contain the change dependency link for that slot, as described in section 13.2.2.



The representations of a pair of cartesian points and their map are displayed on the next page.

[*] This function pointer array is the virtual function array generated by the C++ compiler.
[**] In SELF parents are prioritized; the priority of a parent slot is also stored in the second word of the slot description.



The representation of two cartesian point objects. The objects on the left are the point "instances," containing the values of the **`x`** and **`y`** assignable data slots. The object on the right is the shared map for all cartesian points, containing the value of the constant **`parent`** slot and the offsets of the assignable **`x`** and **`y`** slots.