2/15/82

Object Table Format

For Magnolia use complete ~~32~~ 24-bit byte address in object table entry. High order 8 bits may be used for flags. Reference counts must be kept in a seperate array of 8-bit elements.

O.T. entry:

This allows OT entrys to be directly loaded in to 68000 address registers and used to address objects without any shifting or masking. This is possible because 68000 only uses low-order 24 bits on address bus. NOTE: possible problems with future 68000 processors if the assign meaning to high-order 8 bits. In light of this I will assign flag bits starting at high end working ~~down~~ downward. Flags:

Bit	Meaning if set
31	Free OT entry
30	odd byte-length object
29	object contains pointers
28	"monitor bit"
27-24	"unassigned"

Note that only Free entries are "negative"

2/15/82

Cost comparison of code sequences to obtain object address using Xerox (DAD) and Magnolia OT formats.

Assumptions:

OT is an address register containing address of object table.

DØ contains a displacement into OT.

; Xerox format code	cpu cycles	memory cycles
move.l Ø(OT,DØ),D1	18	4
andi.l #ØFFFFFH,D1	16	3
movea.l D1,AØ	4	1
move.w n(AØ,D1.L),D2	14	3
	52	11
if mask in register subtract	8	2
	44	9

Magnolia format:

movea.l Ø(OT,DØ),AØ	18	4
move.w n(AØ),D1	12	3
	30	7

2/15/82

OOP formats

The Xerox format of an object pointer (OOP) consists of a tag bit as the least significant bit and a 15-bit object number or 2's complement integer in the high order 15-bits.

A alternate format for the 68000 places the tag as the most significant bit. This allows the 68000 condition codes to directly reflect whether the OOP is a smallInteger or an object reference. It also allows OOPs to directly index a reference count array.

OOP format Code comparisons 2/15/82

Low-bit Tag

Fetch a OT entry, oop in D0

code	cpu	memory
~~LSL.L #2, D0~~		
ADD.L D0, D0	8	1
MOVE.L d(OT, D0.L), A0	18	4
	26	5

Test if oop is smallInteger, fail if not. Assume oop just loaded into D0

ASR.W #1, D0	8	1
bcc.s fail	8	1
	16	2

As above but also convert to normal integer

ASR.W #1, D0	8	1
bcc.s fail	8	1
	16	2

High-bit Tag

Fetch a OT entry, oop in D0

code	cpu	memory
LSL #2, D0	12	1
MOVE.L d(OT, D0.L), A0	18	4
	30	5

Test if oop is smallInteger, fail if not. Assume oop just loaded into D0

~~bpl.s~~		
bpl.s fail	8	1
	8	1

As above but also convert to normal integer

bpl.s fail	8	1
add.w D0, D0	4	1
ASR.W #1, D0	8	1
	20	3

More Oop format comparisons 2/15/82

store top of context stack into receiver field 1.
Assume receiver object address is in
address register "receiver"

Low Tag Common High Tag

```
        Move.w  6(receiver), D0
LSR.W   #1, D0
BCS.S   1$                          BMI.S   1$
        counting down D0
        :
1$:     move.w  (CSP)+, D0
        Move.w  D0, 6(receiver)
LSR.W   #1, D0
BCC.S   2$                          BPL.S   2$
        next cycle
2$:     count ~~down~~ up d0
        :
```

High Tags saves 16 + 2

Oop Format Conclusions 2/16/82

The high-tag format is slightly more expensive for object table references and for converting smallIntegers to 16-bit integers. It is twice as fast for simple smallInteger tests and 16 cycles (2 μsec) faster for reference counting (memory storePointer: ...).

If context references are not reference counted then most reference counting occurs in the store instance variable byte codes which dynamically occur about 4% of the time. This is less then the frequency of smallInteger arithmetic. With deferred reference count neither oop format appears to have a real advantage of over the other and hence it is perferable to stick with the Xerox format.

68000 Register Assignments 2/16/82

A7 68000 stackpointer
A6 Base address of object table
A5 Address of home context (arg + temps access)
A4 Context SP (Address of top of active Context stack)
A3 IP address of next bytecode
A2
A1
A0

D7 Next cycle (top of interp. loop) address not needed see page 10.
D6
D5
D4
D3
D2
D1
D0

The address of the ~~Refere~~ Reference count table must be in either a A or D register.

Branch to Top of Interpreter Loop (Nextcycle) 2/16/82
Alternatives

The final step of every byte code is a transfer of control to the routine which selects and dispatches the next byte code. Since this is done for each byte code the overhead should be very low. A related issue is the possibility of a ~~X~~ process switch between each byte code. Ideally the possibility of a process switch should add NO overhead when there is not a pending process switch. The follow code sequences for Nextcycle attempt to achieve this.

Solution 1: Address of dispatcher on top of 68000 stack. Address changed to cause process switch.

```
Nextcycle =     RTS                     16+4
  put address must be restored by dispatcher
                PEA dispatcher.P        16+2+2

                Total cost              32+8
```

Next cycle (cont) 2/16/82

Solution 2: Address of ~~top of~~ dispatcher in an Address register (Ax). Change register contents to cause process switch.

Next cycle:	JMP	(Ax)	8 + 2

Total Cost 8+2
plus 1 A register

Solution 3: Address of dispatcher in Data register (Dx). Change register for process switch.

Next cycle:	MOVEA.L	Dx, A0	4+1
	JMP	(A0)	8+2

Total cost 12+3
plus 1 D register

Solution 4: Like 3 but Dx contains offset of dispatcher relative of OT base

Next cycle:	JMP	0(A6, Dx.L)	14+3

Total Cost 14+3
plus 1 D register

Nextcycle (cont) 2/16/82

Solution 5. Address of dispatcher in global variable. Change variable for processes switch.

Nextcycle = ~~JMP dispatcher.S 10+2~~
MOVEA.L ptrDispatcher.S, A0 16+4
JMP (A0) ~~8+4~~ 8+2

Total Cost 24+6

Solution 6. Unconditional branch to top of dispatcher. Change first instruction of dispatcher to a branch is process switch required.

Nextcycle = JMP dispatcher.S 10+2

total cost 10+2
plus self modifying code

---

fastest slowest
2 6 3 4 5 1

#2 is best but requires dedicate A reg (unlikely)
next best is #6 which uses not registers.

O.T. Flags 2/16/82
(continued from Page 1)

Problem: There are 8 available flag bits, 4 were assigned meaning on page 1 and 4 were unassigned. I want to reserve 3 bits for LOOM (leaving 1 unassigned). Inorder to support deferred reference counting I need several additional bits.

See Krasner letter 2/16/82

Alternatives:

1) Remove the ~~FREE~~ FREE bit from the flags ~~pat~~ and put it into the low order 24 bits. Possible encodings are bit 23 set (limits addressable memory to 8 meg), bits 16-23 = FF (limits addressable memory to 16 meg - 64K), or bit 0 set (object may not be allocated on odd boundries, free list complications).

2) One of the bits needed for deferred reference counting is a bit which says "don't count references from this object". Pointer bit can serve this function

Hardware Assist for bytecode dispatch 2/17/82

The minimum 68000 instruction dispatch overhead is 48 cycles and 11 memory references (8+ μsec) (see pages 8-10).
This includes branch to dispatcher:

```
JMP     dispatcher.S                    10+2
```

and actual dispatch code:

```
MOVEQ   #0, D0                          4+1
MOVE.B  (IP)+, D0                       8+2
ADD.W   D0, D0                          4+1
MOVE.W  TABLE (PC, D0.W), A0            14+3
JMP     (A0)                            8+2
                                        48+11
```

Could a hardware assist improve this significantly?

Consider a device which contained a low copy of the dispatch table and the Smalltalk Instruction pointer. The device could then perform the function of fetching the next bytecode, incrementing the IP and returning the dispatch address. It would do this in response to a read from ~~the~~ a hardware assigned location. The 68000 code to do this would be:

```
movea.w  dispatchHardware.S, A0         12+3
JMP      (A0)                           8+2
                                        20+5
```

2/17/82

The above code would replace the branch to the dispatcher at the end of each bytecode implementation.

Since a 68000 address register would no longer be needed to contain the IP that register could be used to contain the address of the dispatch hardware. This would reduce the code sequence to:

```
movea.w  (Ax), A0      8+2
JMP      (A0)          8+12
                       16+4
```

(2 μsec+)

To support multibyte instructions the hardware would have to support reading of the actual byte pointed to by the IP. (Note that if the hardware returned the dispatch hardware on a word read and the actual data byte on a byte read then there would be no penality for reading instruction stream bytes)

The hardware must also support reading and writing of the IP value and process switching. Process switching can be done by force a fixed dispatch address.

2/17/82

Logical discription of dispatch hardware operations.

Build Dispatch Table
Used to load the value into the dispatch table. Since this is only done once the means of doing this isn't very important.

Store IP Value
Stores new value into IP register. Fetches byte pointed to by new IP value and places it into Dispatcher Data register. Uses data byte to index dispatch table and save table value in Dispatcher "Address" register.

only if process switch flag not set

Fetch IP Value
returns value in IP register.

Fetch Data Value
Return value in data register. Increment IP. Fetch byte pointed to by IP into data register. If process switch flag not set load dispatch value into dispatch address register.

2/17/82

Fetch Dispatch Value
Return value in dispatch address register. If process switch flag not set increment IP, fetch byte and new dispatch address. Clear flag

Store Process Switch
~~Store~~ Set process switch flag.
Store new value in dispatch address register.

Description of registers in dispatch hardware

IP 24-bit 68000 address of value in Data Reg.

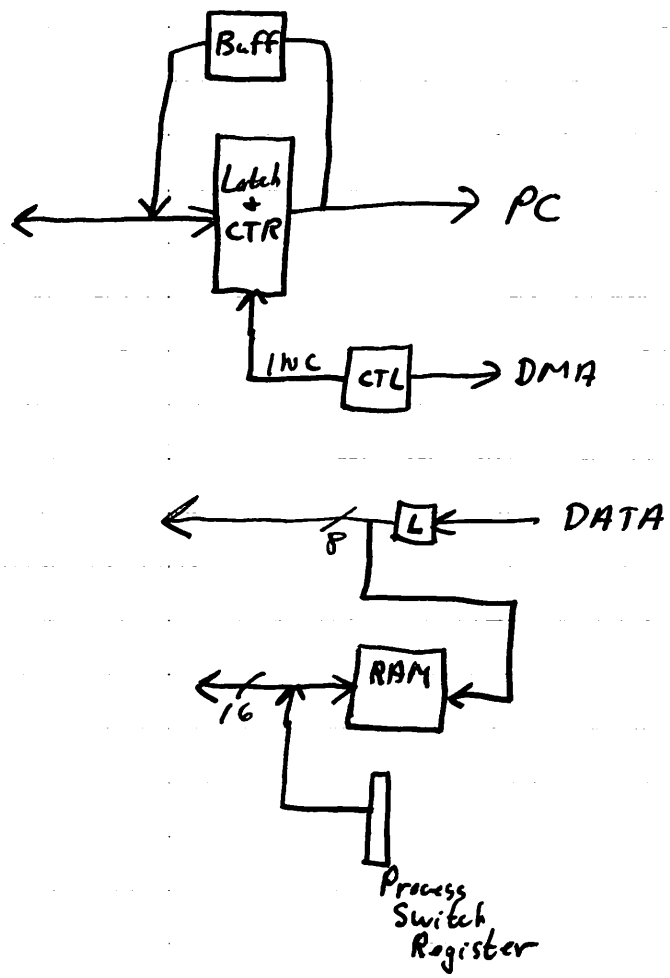
Data Reg 8-bit value of byte addressed by IP

Dispatch Reg 16-bit value returned by Fetch Dispatch Value

Process Switch 1-bit flag indicating Dispatch Reg value "Frozen"

2/17/82

Chip Schnarel's block diagram of dispatch hardware:



~~2/17/82~~ 2/17/82

Caching Contert Stackpointer on 68000

One 68000 address register will be used to contain the address of the top of the active contert's stack. Since 68000 stacks normally grow from high memory to low memory while Smalltalk stacks grow from low memory to high memory we must use addressing modes which are normally ~~are~~ 68000 "pops" for smalltalk "pushes". To accomplish this requires that the "cached stack pointer" actually contain the address of the first free word of the stack.

E.G. CSP = addr(Top of stack) + 2
push x = move.w x, (CSP)+
pop x = move.w -(CSP), x

The value of the SP field of a contert is a smallInteger (i.e. value*2+1) array index (first element is element 1) of the top of the stack. So to convert the value of a SP field to a CSP value:

CSP = addr(contert) + size_in_bytes(objectHeader)
+ size_in_bytes(contert fixed fields)
+ SP - 1

SP = CSP - addr(contert) - size_in_bytes(objectHeader)
- size_in_bytes(contert fixed fields) + 1

Caching Context Instruction Pointer 2/18/82

The instruction pointer field of a context is a smallInteger which is the ~~atum~~ byte array index (one-origined) of the next instruction to be executed. Computing a cached 68000 address value is as follows:

$$CIP = addr(method) + size\_in\_bytes(objectHeader) + (IP/2) - 1$$

$$IP = 2*(CIP - addr(method) - size\_in\_bytes(header) + 1)$$

$$= 2*(CIP - addr(method)) + 2*(1 - size\_in\_bytes(header))$$

Indicating Free OT entries 2/22/82

(see page 11)

Define that bits 16-23 of an O.T. entry (high byte of 24-bit object address field) set to zero indicates a free O.T. entry. This means that Smalltalk object same can not occur within the first 64K of the 68000's address space. But that isn't a problem since the fist 64K is allocated to the interpreter. An advantage of this representation is that it allows a 32-bit varrable to contain either a OOP or an address. ~~Values~~ Values of 65535 or smaller are OOPs all other values are addresses.

```
Form fill:t1 rule: t2 mask: t3

"Modified to recognize cases when
 filling form with white or black so
 that BitBlt without half tone
 may be used"

| newRule ~~newMask~~ |

    newRule ← ~~rule~~.t2
    ~~newMask ← t3.~~
    t2 == Form over ~~if True~~:
        if True: [ t3 == Form white
                if True: [ newRule ← ϕ. ~~newMask ← nil~~]
                if False: [
                    t3 == Form black if True:
                        [ newRule ← 16 ]~~newMask ← nil~~
                        ]
                ]

BitBlt
    .
    .
    .
```