

AM05: Unified Moves

Adam Megacz
megacz@cs.berkeley.edu

September 27, 2006

“FLEET is a one-instruction computer”

– Ivan Sutherland

1 Introduction

What follows is a unified theory of MOVES in which:

- The various proposals for different kinds of moves can be explained as special cases of a single instruction.
- Records can be easily simulated, and the corresponding programming idioms retained.
- Programs (and programmers) do not need to worry about the FLEET-specific upper limit on move counts, which corresponds to the maximum record size.

In the final section I give a specific, precise example of why simple semantics are important for FLEET, and how unified moves give us the power of records without losing that simplicity.

2 One Instruction, Four Forms

Lest we lose sight of Ivan’s original vision, I would like to rephrase the different kinds of moves as instances of a single move instruction. Every MOVE instruction has a numerical count argument from the set $\{0 \dots \text{MAXMOVE}, \infty\}$, where MAXMOVE is specific to a particular FLEET implementation.

In this framework,

- An “unmove” or “kill-move” is a MOVE with count=0.
- A “simple move” is a MOVE with count= 1

- A “counting move” is a MOVE with count= n where $1 \leq n \leq \text{MAXMOVE}$
- A “standing move” is a MOVE with count= ∞

2.1 Syntax Proposal

I propose the following syntax for moving 0, 1, 2, 3, 31337, and ∞ elements from port p on SHIP ship1 to port q on ship2:

```

ship1.p    (0)-> ship2.q // unmove (count=0)
ship1.p    -> ship2.q // single move (count=1)
ship1.p    (2)-> ship2.q // counting move (count=2)
ship1.p    (3)-> ship2.q // counting move (count=3)
ship1.p    (31337)-> ship2.q // counting move (count=31337)
ship1.p    *-> ship2.q // infinite ("standing") move

```

3 Semantics

3.1 $n = 1$ (“simple move”)

In the case where $n = 1$, the MOVE instruction behaves as previously described.

3.2 $1 < n \leq \text{MAXMOVE}$ (“counting move”)

In the case where $1 < n \leq \text{MAXMOVE}$, the MOVE instruction is *semantically identical* to contiguously repeating a count= 1 MOVE instruction n times.¹

Specific FLEET implementations will almost certainly apply optimizations in this case, but from the programmer and compiler’s perspective, there is no difference other than code size.

3.3 $n = \infty$ (“standing move”)

In the case where $n = \infty$, the MOVE instruction goes to the source *and remains there*. While it is at the source, any datums which arrive are immediately sent through the switch fabric to the destination of the $n = \infty$ MOVE.

However, the $n = \infty$ instruction behaves differently from the other forms in one crucial way: other instructions *do not* queue up behind the $n = \infty$ instruction when they arrive at a source.

¹Notice how the previously instituted in-order-delivery requirement makes it easy to explain how counting moves work.

If an $n = \infty$ instruction is present at a source when *any other* instruction arrives behind it, the $n = \infty$ instruction is destroyed. This will require an arbiter².

3.4 $n = 0$ (“token move”)

Motivation: this instruction exists mainly in order to delete an $n = \infty$ MOVE which is already at the source (see above).

When an $n = 0$ instruction arrives at a source, it sends 0 datums to the destination listed in the move instruction, exactly as one would expect. However, in place of datums, the source will synthesize a *token* and sends *it* to the designated destination.

This allows us to condition some other operation on the successful destruction of a standing move – simply use a $n = 0$ move to tear down the standing ($n = \infty$) move, and direct the destination of the $n = 0$ move to the SHIP which needs to be notified when the standing move has been torn down.

The most common use case will typically involve many codebags having a partner “teardown” codebag which dispatches $n = 0$ instructions corresponding to the standing moves in the primary codebag. The tokens generated by those $n = 0$ instructions will be directed to some sort of “token counter”, which will dispatch the successor codebag when it receives a number of tokens equal to the number of standing moves to be torn down.

4 Simplicity Matters

4.1 Simulating Records

I would like to note that MAXMOVE will probably be a fairly small number, approximately 8, for exactly the same reasons why the proposals for records have a size limit of 8. In fact, most record-driven code can be translated in a very straightforward way into counting-move code where the counts are set to the “record” lengths.

However, unlike records, the assembler can easily support “records” of arbitrary lengths by simply translating a move with $n > \text{MAXMOVE}$ into a collection of $n = \text{MAXMOVE}$ instructions and possibly a “remainder” move. This lets the programmer (and the compiler) effectively ignore the value of MAXMOVE unless it becomes a performance concern.

This last point bears repeating: *because the semantics of “unified moves” are simple and regular, there is an automated, meaning-preserving transformation to reduce the*

²however, any switch fabric will already need to have at least $O(\log \text{NUMSHIPS})$ arbiters on each path through the fabric, so one more is not likely to be a problem

maximum required move-size. In contrast, the complex (and still not yet fully specified) semantics of records do not immediately suggest any such meaning-preserving transformation.

4.2 Different SHIPs Can Have Different MAXMOVES

One of the unfortunate consequences of supporting records is that every SHIP needs to support the record interface.

However, with counting moves, only the SHIPs which are the *source* of multi-word transfers need the additional circuitry to handle starting and ending the transfer.

Furthermore, the MAXMOVE limit can be different for different SHIPs, allowing large and complex SHIPs to support very large MAXMOVES by including a (physically bulky) counter. Tiny ships can have a MAXMOVE of 1, effectively freeing them from dealing with counting moves³.

Most importantly, all of these parameters can be completely hidden from the programmer *and even the compiler writer*. The assembler (or even the fetch unit) can break up large moves into contiguous sequences of small moves without any concern whatsoever about damaging the meaning of the user's code.

□

³all SHIPs must include circuitry to support $n = 0$ and $n = \infty$, although I believe this will be tiny