

# UC Berkeley Computer Science

**Subject:** ZOMA: The Return of the Standing MOVE  
**Date:** October 8, 2006  
**From:** Ivan Sutherland  
**UCIES** #2006-is36

## References:

UCIES# 2005-is02: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2005  
UCIES# 2005-is03: Defining Some SHIPs, Ivan Sutherland, 24 August 2005  
UCIES# 2006-is23: FIFO Register File, Ivan Sutherland, 30 June 2006  
UCIES# 2006-is24: Parallel Switch Fabrics, Ivan Sutherland, 10 July 2006  
UCIES# 2006-is25: Instruction Sequence in FLEET, Ivan Sutherland & Igor Benko, 19 July 2006  
UCIES# 2006-is30: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2006  
UCIES# 2006-is31: Some SHIPs, Adam Megacz and Ivan Sutherland, 24 August 2006  
UCIES# 2006-is32: A Record Store Implementation, Ivan Sutherland, 6 September 2006  
UCIES# 2006-is34: The FLEET Architecture (slides), Ivan Sutherland, 15 September 2006  
UCIES# 2006-is35: Some SHIPs (slides), Ivan Sutherland, 20 September 2006  
UCAM# 2006-am05: Unified Moves, Adam Megacz, 27 September 2006

## INTRODUCTION

On Wednesday, 27 September 2006, the class staged a revolt against records and demanded return of the standing MOVE. Over the summer I had eliminated standing MOVEs for what I believed to be good and sufficient reason, namely that they are hard to kill. The former kill mechanism involved an END symbol in the data. The standing MOVE was to send the END symbol as its last dying act. I have come to believe that FLEET's transport mechanism should be agnostic about the data it handles; the meaning of data should be entirely up to the SHIPs. Thus, my objection was to the END symbol and not actually to standing MOVEs.

I had replaced counting MOVEs and standing MOVEs with Records. The central issue about Records is who controls data grouping. With a counting MOVE the program states explicitly how many words move together, whereas with a Record, the number of words to move is implicit in the record itself.

Counting MOVEs require a different program to move a different amount of data, whereas Records permit the same program to handle different amounts of data. There's also a minor implementation efficiency to gain from Records because the words of a Record move require only a single destination address.

During the memorable 27 September class several people expressed regret at the loss of the standing MOVE. It served, they correctly point out, as a way to "wire

---

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley.. The ideas contained herein are freely available for any academic purpose.

up” FLEET into a special purpose pipeline processor. To counter my objection of the difficulty of getting rid of standing moves, Adam Megacz explained his alternative kill strategy (see AM05). He proposed that any new instruction arriving at a source should replace a standing MOVE. Moreover, the class objected to the complexity introduced by records.

I stood in front of the class wondering whether I should resist what turned into a ground-swell of opinion from smart students. Did I want to accept their ideas or use my own? I decided that I was here at Berkeley precisely to listen and learn, and so FLEET changed that very day. Only time will reveal the wisdom of that change.

## ZOMA

There are now four types of MOVE instructions. A MOVE instruction can move Zero, One, Many or All data from a source to a destination. ZOMA says it all.

A MOVE instruction with a *zero* count creates a brand new token and sends it to the chosen destination. Such MOVE instructions can serve to kill standing MOVES.

A MOVE instruction with a count of *one* sends one data element. This may be just the “count of one” case of a counting MOVE. COPY instructions with counts larger than one have little meaning and may clog the switch fabric. We may choose to forbid them.

A MOVE instruction with counts larger than one but smaller than some value called *maxmove* sends the indicated number of data elements as soon as they become available at the source. Such an instruction remains active at the source until it has moved the indicated number of data elements and then vanishes. Such “counting moves” may serve applications like graphics involving short vectors.

A “Standing MOVE” instruction remains active until replaced. Any MOVE instruction arriving at the source where a Standing MOVE is active replaces the standing MOVE. The new instruction may have a count of zero, in which case it immediately creates a token and sends it to its indicated destination. Such a token signals that the standing MOVE is no longer active. The new instruction may have some count other than zero, in which case it takes over transmission of data from the source. The new instruction may be another standing MOVE, in which case it redirects subsequent data to a new destination.

## IMPLEMENTATION

An OutBox capable of handling a counting MOVE must contain a counter. Although the capacity of the counter might differ for different instances of OutBox, simplicity demands that any OutBox with a counter be able to handle a count up to *maxmove*. Not all OutBoxes need contain counters. In the first implementation of the simulator, all OutBoxes will be the same.

An OutBox capable of handling a standing MOVE must contain an arbiter. The OutBox may have space to hold two MOVE instructions, the active one and its potential replacement. If a counting move is underway, the arbiter must decide, for each an every data element, whether to process that data element or whether a potential replacement is ready for use. The two inputs to the arbiter are thus the “data ready” signal from the SHIP’s output and the “potential replacement ready” signal. Not all OutBoxes need handle standing MOVEs.

## IMPLICATIONS

The need for an arbiter to implement standing MOVEs indicates, or is the result of, indeterminacy in the replacement action. How many data elements did the standing MOVE send before another MOVE replaced it? The number of data elements sent depends on exactly when the new instruction interrupted the flow of data. The details of the interruption depend on delays that may vary. It will be interesting to see what precautions programmers will use when replacing a standing MOVE.

## RECORDS

In addition to its role in moving data, the Record served to group data elements together. The fact that a word was the “last” word in a Record was evident at both the source and the destination. The counting MOVE instruction omits this property; the count is known only at the source and not at the destination; every data element including the last appears the same at the destination.

The grouping implied by records made it possible to COPY a record with a single COPY instruction. Although this cost considerable complexity in the equipment to handle records (see is32) it seemed a useful programming concept.

Some SHIPs made use of the grouping provided by records. For example, the Fetch SHIP used the length of its code block descriptor Record to provide a variety of functions. For another example, the Arithmetic SHIP treated its command record as a short program. If the command record was too short for the data records, the Arithmetic SHIP used the command record over again. I hope that we can retain some of this flavor in the Arithmetic SHIP.

Remember that although formal records are gone, FLEET’s register file is made of FIFOs. To remind us of that, let us call it the FIFO file. My notion includes about 16 FIFOs, each 8 elements long. The program will retain detailed control of the content of the FIFO file, using counting MOVEs to put data into and take data from the FIFOs. Note that because the FIFOs have pipeline interfaces, a pair of standing MOVE instructions can couple two independent FIFOs into a longer FIFO if required.

Programs may choose to use a FIFO in several ways. One use will record successive values of short vectors in a FIFO. For example, the three coordinates of a point in space might appear sequentially in a FIFO. A counting MOVE with a count of

three could then send a point from the FIFO to the arithmetic element. Another use will store unrelated variables, such as the variables internal to a loop, in a FIFO. For example it may be useful to store three memory pointers and a count in successive locations in a FIFO. The program could send these variables to the arithmetic element with a MOVE with a count of four.