

# UC Berkeley Computer Science

**Subject:** Record SHIPs  
**Date:** August 1, 2006  
**From:** Ivan Sutherland, Igor Benko and Mark Greenstreet  
**UCIES** #2006-is26

## 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# 2005-is23: FIFO Register File, Ivan Sutherland, 30 June 2006  
UCIES# 2005-is24: Parallel Switch Fabrics, Ivan Sutherland, 10 July 2006  
UCIES# 2005-is25: Instruction Sequence in FLEET, Ivan Sutherland & Igor Benko, 19 July 2006

## INTRODUCTION

This memo offers some thoughts about possible record SHIPs. Please recall that a MOVE instruction can include a count of the number of words to move as a group. Such instructions permit FLEET to pass small groups of words, called “records”, quickly from a source to a destination. Because the words of a record follow each other, the record not only can use the large throughput offered by the switch fabric but also costs only a single latency period. For now we think of a record as 8 words or fewer. This memo offers some functions to show how processing records might work.

Conventional computers include an “OP CODE” in each instruction. Heretofore, FLEET’s design also assumed that bits to select which operation to perform on data will be embedded in the instruction stream. Our earlier conception of FLEET selected the operation by details of destination addresses rather than as a separate OP CODE in each instruction. In effect, the OP CODE bits were embedded in the destination address.

## OP CODES AS DATA

This memo also takes a fresh look at OP CODES. Conventional machines use an OP CODE embedded in each instruction to tell the arithmetic and logical unit (ALU) what to do. The OP CODE bits tell the ALU what to do through an implicit command input to the ALU. Commands for the ALU are, in effect, literals embedded in the instruction stream. The ALU command takes up bits in each instruction.

We now propose to separate OP CODES entirely from the instruction stream, treating ALU commands as data. Thus a SHIP for arithmetic and logical operations with two data inputs will have a third input, its “command” input. The command input will tell the ALU SHIP what to do with the data. Such a SHIP will operate on its data only when both data inputs and the command input have received values.

Providing a separate command input to arithmetic elements has four implications. First, FLEET can deliver a record of commands to an arithmetic element as well as the input data records. This command record permits a SHIP to apply different actions to different elements in the data records. For example, a SHIP might form the sum of the first pair of data elements, the difference of the second pair, the average of the third pair, etc.

---

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.

The second implication of a separate command input is that FLEET can reuse command constants during each iteration of a loop. The command constants need no longer occupy space in the instruction stream; they are data.

A third implication is that a program can compute a command value. This change removes the distinction between “fixed” and “variable” shifts. In a conventional computer, the amount of a “fixed” shift is specified in the instruction stream, whereas the amount of a “variable” shift is a data input to the ALU. With command inputs, this distinction rests entirely on whether the shift command is a literal used as is or modified before use.

A fourth implication is that because there are more bits to specify the ALU’s operation, we can offer ALUs with a greater variety of functions. The tables at the end of this memo suggest some possibilities from which to choose. Moreover, the same command values given to different SHIPs may result in different actions.

A fifth, and possibly bad, implication of separate command inputs is that the command constants must enter FLEET as literals. Command values will occupy literal space and will require set-up instructions and internal storage space. The cost of set-up and storage depends on how many different command values the program uses and how they are stored. This memo shows that treating commands as another form of data can simplify some repetitive operations. Unfortunately, that simplicity may make non-repetitive operations more cumbersome.

## **RECORDS AS DATA ELEMENTS**

Because moving groups of words, called records, in FLEET can be fast, programmers may wish to treat the variables for a loop as components of a single record. For example, three memory addresses and a count might form a four-component record. Instead of processing these variables separately, the program might treat such a record as a unit. A matching record of constants could define the stride values for the memory addresses and the increment for the counter. A suitable command record would tell an ALU how to process each element of the data records. Each iteration of the loop would use three counting MOVE instructions to deliver the variable record, the constant record, and the command record to the ALU. The command record would apply the strides in the constant record to the addresses and increment the count by the appropriate step. The housekeeping part of such a loop would be simple and fast.

Simple operations on records can also provide multiple-precision arithmetic. Let us store a multiple-precision number as a record with least significant part (LSP) first. Given this form, the ALU can insert the carry from the least significant part into a more significant part to do multiple precision arithmetic. The corresponding command record would specify how to link the words in the record by generating, saving, or using the carry bit. Omitting the carry link between selected words of the record would permit several multiple-precision numbers to cohabit the same record as, for example, a multi-precision complex number.

## **LONG VECTORS**

The ability to move a few data elements as a group can also assist in processing long vectors. FLEET can process a long vector as a series of records. For example, FLEET might treat a vector of 66 elements as eight records of eight components each and one final record with two components. By processing eight elements from the vector at a time FLEET can reduce latency and gain throughput.

## A TAXONOMY OF PROCESSING

It seems useful to make a taxonomy of types of SHIPS that could process records. A taxonomy needs axes. The number of data record inputs (0-3) and the number of data record outputs (0-2) seem suitable as two axes. Recall that all such SHIPs will also have a command record input. For a third axis, let's consider how much information passes from one component to the next. For this axis the values are: none, a Boolean result, a few bits of result, and one or more full words that might be parameters.

Some entries in the resulting taxonomy are relatively dull. The entry with zero data inputs and zero data output can form only a bit bucket for commands. The entry with one or more data inputs and zero data outputs can be either a bit bucket for the data inputs or some kind of output device. Similarly, the entry with one or more data outputs and no data input can be either some kind of input device or a source of data that depends on the command. Suitable commands could produce constants, random numbers, date or time of day, and so forth.

A class of operations not covered by this taxonomy involve structural changes to the records. For example, a SHIP might interdigitate the components of two input records to form an output record twice as long. Another SHIP might reverse the sequence of the entries in the record. Another SHIP might pack a record of characters into a shorter record of integers, or unpack a record of integers into a longer record of characters. Yet another ship might produce a collection of addresses according to some parameters that specify strides. We omit such operations here, focusing instead on cases with equal lengths for the command record, the data input records and the data output records.

The next pages offer some ideas for possible functions, fitting them into the taxonomy. Each page shows the number of input records (1 – 3) and the number of output records (0 – 2) for a different choice of the third axis. We assume that the length of the data records and the command records are equal.

**NOTHING CARRIED ALONG THE RECORD, ELEMENTS ARE SEPARATE**

	1 output	2 outputs
1 input	absolute magnitude, complement, negate, increment, decrement, copy output all zeros, output all ones output random number table look up fixed shift or rotate bit reversal, tally = count ones wordwise parity = # of 1s is odd CRC parity bits normalize, output mantissa normalize, output exponent Boolean out if (+ - 0) 1,0,-1 out if (+ - 0) increment and test (+ - 0)	sum and difference normalize, and output both mantissa and exponent = a part of fixed to floating conversion + or - Absolute Magnitude
2 inputs	add, subtract, average saturating add, subtract bitwise logic: and, or, xor, etc. variable shift and rotate select the greater, lesser compare >, =, < Boolean out compare >, =, < with 1,0,-1 out copy A ignore B, copy B ignore A	compare and swap swap or copy
3 inputs	Boolean A selects B or C three input add saturating three input add A + B but not > C nor < -C Muladd: A*B+C bitwise parity, bitwise majority masked insertion copy A, copy B, copy C ignore the others	Boolean A swaps B and C bitwise parity and majority = carry save add divide step (what is this?)

**BOOLEAN CARRIED ALONG THE RECORD**

	1 output	2 outputs
1 input	multiple precision one bit shift multiple precision increment, decrement multiple wordwise parity = # of 1s is odd multiple precision tests with Boolean outputs	
2 inputs	multiple precision add, sub multiple precision saturating add, subtract multiple precision comparison copy A ignore B, copy B ignore A compare first components >, =, < and copy all of chosen record	compare first components >, =, < and swap or copy all
3 inputs	multiple precision three input add multiple precision three input saturating add, subtract	

**MANY BITS CARRIED ALONG THE RECORD – often first component of record**

	1 output	2 outputs
1 input	<p>fill output record with first input value copy component indexed by the first component multi precision shift or rotate several reverse the order of the components rotate the order of the components multi precision normalize, output mantissa multi precision normalize, output exponent bit position of the first one in record multi precision tally = count ones lengthwise logical ops e.g. bit by bit lengthwise parity</p>	<p>multi precision normalize, output both mantissa and exponent, a part of fixed to floating conversion</p>
2 inputs	<p>multi precision variable shift and rotate Boolean output from multi precision comparison multi precision select the greater, lesser compare first multi precision components &gt;, =, &lt; and copy all of chosen record multi precision output 1,0,-1 for &gt;, =, &lt; dot product w/running sum output max value, min value, location of max component count words, count zero words floating point arithmetic exponent and mantissa are component pairs interval arithmetic upper bound, lower bound are component pairs</p>	
3 inputs	<p>select A or B or C according to test of first components of A and B</p>	