

UC Berkeley Computer Science

Subject: Instruction Sequence in FLEET
Date: July 19, 2006
From: Ivan Sutherland and Igor Benko
UCIES #2006-is25

References:

UCIES# 2005-is02: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2005
UCIES# 2005-is14: FLEET – A One-Instruction Computer, Ivan Sutherland, 5 December 2005
UCIES# 2006-is23: FIFO Register File, Ivan Sutherland, 30 June 2006
UCIES# 2006-is24: Parallel Switch Fabrics, Ivan Sutherland, 10 July 2006

BACKGROUND

A continuing problem in FLEET is how to impose sequence on an otherwise parallel machine. FLEET, as described in the 5 December 2005 memo, is almost entirely concurrent, leaving to the programmer achievement of robust sequencing by data flow or tokens. According to the 5 December definition, FLEET was free to issue instructions from a code bag into the instruction pool in any sequence. FLEET was also free to execute instructions from the pool in any sequence, limited only by the availability of data and congestion in the switch fabric. The instruction pool avoided any distinction between newly entering instructions and instructions left over from previous code bags.

The 5 December 2005 definition of FLEET offered only one sequence guarantee: Instructions with the very same source and the very same destination must preserve the sequence of the data items that pass from that source to that destination. This guarantee allows a program to move a vector of values from a source to a destination and preserve the sequence of the elements in the vector. It fails to guarantee absence of extraneous data entering the same destination from some other source. This limited guarantee is easy to implement if data move from that source to that destination along the same path in the switch fabric.

However, this limited guarantee leaves open the questions of “vectorization” and “de-vectorization.” Vectorization: How does one put components in sequence into a vector in the first place? And, de-vectorization: How does one access the separate components of a vector? These tasks are akin to parallel to serial conversion and serial to parallel conversion. In some sense they convert between the space domain of the address space and the sequence domain of data elements that “follow” one another.

Multiplexer and de-multiplexer SHIPs can victories and de-victories. A Multiplexer SHIP has several destination addresses. When each has been filled, such a SHIP can serve as the source for the corresponding vector. A de-multiplexer SHIP does the opposite task. Given a vector of values at its input it delivers the components of that vector at separate source addresses.

Although such SHIPs provide a mechanism for building and accessing vectors, they do so at extra cost in communication. Can we make a simpler mechanism for building and accessing vectors?

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.

PURPOSE

Igor and I have been thinking about changes to FLEET that might simplify sequencing in various ways. Perhaps, while retaining the simplicity of FLEET, we can assure the programmer that some instructions will issue before others. What value might such a guarantee offer? What would it cost.

This memo explores the addition of two sequencing means for FLEET. First, the fetch and issue mechanism might issue instructions in their sequence in memory. Second, the generalized “trunk”, called the “Crossover and Casting Mechanism” in memo 24, might maintain some sequence control over instructions with multiple destinations.

INSTRUCTION ISSUE

The Parallel Switch Fabrics memo, 24, illustrates the use of multiple switch fabrics for different types of data. Instructions that access integer sources, for example, pass through an integer instruction horn on their way to the integer sources. Instructions that access booleans pass through a separate boolean instruction horn. Instructions are thus categorized by the data type of their source register.

Let us define “instruction issue” as the process of placing an instruction in its proper instruction horn. The instruction pool consists of all instructions already in an instruction horn, or waiting at a source for data. Thus instruction issue is the process of placing an instruction in the pool.

INSTRUCTION HORNS

A simple instruction horn will have only one input and only one path from that input point to each of its source locations. Such a horn must preserve the sequence of instructions that access the same source location. For such a simple instruction horn, the sequence in which data are taken from any source is the same as the issue sequence for instructions with that source.

The illustration in the multiple data path memo suggests that instructions for different data types might be issued concurrently into four separate simple instruction horns. Such a structure would preserve the sequence of instructions for integer sources, the sequence of instructions for boolean sources and so on.

A more complicated instruction horn might accept more than one instruction at a time. One such instruction horn might contain two complete instruction horns with a demand merge at each source location. Such an instruction horn pair could issue two instruction in parallel to any source location. In particular, it could issue two instructions with the same source location in parallel. Although, it would offer higher throughput than the simple horn, I cannot see how it could preserve instruction sequence, even for instructions with the same source address.

Another form of instruction horn might divide the source address space into more than one part. For example, instructions with even source addresses might go into one horn while instructions for odd source addresses would go into another horn. Such a mechanism would preserve sequence for instructions with the same source address while also providing greater throughput than the simple instruction horn. In effect it treats even and odd source addresses in the same way previously proposed for separate instruction horns for different data types.

GAINS FROM PRESERVING SEQUENCE

Suppose that there are four values stored in a FIFO whose output is source address A. Four instructions of the form:

`A => B; A => B; A => B; A => B;`

will move that vector of eight values to destination B, preserving the sequence of the values. The sequence of values is preserved by the switch fabric independent of the sequence in which the instructions issue, because the instructions are identical.

However, absent some instruction sequencing rules, the four instructions

`A => B; A => C; A => D; A => E;`

might issue in any sequence and thus move the four components of the initial vector into destinations B, C, D, and E in any sequence.

On the other hand, if we can preserve the sequence of instructions for the same source address, the same four instructions

`A => B; A => C; A => D; A => E;`

will faithfully put the first component into B, the second into C, the third into D and the fourth into E. Thus preserving the sequence of instruction issue for the same location provides de-vectorization.

THE INSTRUCTION SEQUENCE

It has been common to declare the instruction sequence in a code block to be the numeric sequence of the memory locations in which the instructions appear. This seems an obvious choice, but there are two other alternatives that appear to be attractive, making three in all for us to consider.

One alternate sequencing technique marks the instructions with their intended sequence. With this technique, the instructions

`A.1 => B; A.4 => E; A.3 => D; A.2 => B;`

carry markers, here denoted as `.#` that indicate the intended sequence. They appear unsorted in the line above to make clear that it is the decimal notation prevails over the lexical sequence. Moreover, interspersed but unmarked instructions might issue in any sequence.

The second sequencing technique uses only the sequence of code bag issue. According to this plan, the four instructions above would have to appear in separate code bags, code bags interpreted in sequence, to perform their intended purpose. Such code bag sequencing could easily be enforced by ensuring that all instructions from one code bag issue before any issue from the next code bag. In some sense this alternative puts the onus of sequencing on the instruction fetch and issue mechanism.

LIMITATIONS – DESTINATION SEQUENCE NOT GUARANTEED

Sequencing instructions by source address guarantees that source data are launched into the switch fabric in sequence. The source fabric can guarantee that such data will not bypass other data moving from the same source to the same destination. The source fabric fails, however, to make any other delivery guarantee.

In particular, imagine moving a vector from A to B with the four instructions

`A => B; A => B; A => B; A => B;`

Now suppose that a fifth instruction, `X => B`, issues concurrently. Although we can be sure that the value of X will enter the existing four-component vector, its position in the resulting

five-component vector is indeterminate. I know of no simple rules for the switch fabric that will “fix” this problem.

Thus, although sequencing can simplify access to the components of an existing vector, it cannot help in building the vector in the first place. We will still need a SHIP with multiple inputs to transform data distinguished only by address into a vector sequence.

SEQUENCING IN THE TYPE-CAST MECHANISM

A mechanism to guarantee that a certain MOVE instruction has at least reached the crossover and type-cast mechanism might help with destination sequencing. Let us suppose that we are willing to separate two instructions:

A => D; and B => D;

into two separate code bags. How can we know that the second code bag is sufficiently delayed that A and B arrive at D in that sequence?

One way is to sense the arrival of A at destination D. Such an arrival can release a token that, in turn, can release the second code bag. This method has proven of value for sequencing the flow of data between pipeline interfaces. When each data element arrives, it releases a token that notifies the source to send the next datum. However, need we wait until the data arrive at their destination? Instead, we might include something in the crossover and type cast mechanism that could give use earlier notice.

The crossover and type conversion mechanism could provide some sequence guarantee for delivery to multiple destinations. For example, it could guarantee something about release to multiple destinations. With such a guarantee, one could send integer data to integer destination D and as a second destination for the same instruction send the same data, converted to a token, to another destination that would release another code bag.

Thus the pair of code bags:

bag1: A => D, X; bag2: B => D;

would deliver A and B to D in that sequence if delivery of the token to X is required for release of bag2.

It appears relatively easy to provide such a guarantee in the crossover network. Like the problem of preserving sequence in the instruction horns, design of parallel type cast implementations must avoid demand merge after separation the multiple destinations.