

UC Berkeley Computer Science

Subject: Scatter and Gather
Date: August 2, 2006
From: Ivan Sutherland
UCIES #2006-is27

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-is26: Record SHIPs, Ivan Sutherland, Igor Benko & Mark Greenstreet, 29 July 2006

INTRODUCTION

This memo addresses the problem of distributing values from a record to disparate locations and of assembling disparate values into records. I think of these as the “scatter” and “gather” tasks. Scatter converts sequence to location; gather converts location to sequence.

It appears attractive for FLEET to process groups of data elements as records. Moving a record through the switch fabric can be quick and efficient when the source has the entire record ready and the destination can receive the entire record. The entire record can pay only one latency period and can make use of the high bandwidth of the switch fabric. With records, a few instruction bits can do a lot of useful work.

Memo ucb25, Instruction Sequence in FLEET, offers a guarantee of sequence for instructions that use the same source. The guarantee of sequence provides a simple scatter mechanism. Because the elements of a record appear at the same source in sequence, successive MOVEs from that source can send successive elements of the record to known destinations. The sequence of destinations in the MOVE instructions sets the places where successive records go. Without the sequence guarantee for sources we could scatter the elements of a record, but where each element would go would be indeterminate.

The gather task is harder. FLEET avoids making guarantees about the arrival sequence of data from different MOVE instructions that target the same destination. It must avoid destination sequence guarantees, because there is no way of knowing when data elements emerge at their sources. Thus even if instructions with a common destination were issued in sequence, the values might arrive in any sequence.

ENSURING ARRIVAL SEQUENCE

There are several ways to establish the sequence in which data elements arrive at a destination. Each way to establish sequence involves using a token to delay the actions of later MOVE instructions until earlier MOVE instructions are complete, or at least complete enough to guarantee the sequence.

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.

Using a Pipeline Interface: The pipeline interface structure defined in an earlier memo makes data available at a source only upon receipt of a token. Suppose that there are three data sources, **A**, **B** and **C**, each with a pipeline interface. Each source will have a corresponding token destination that I will call **a**, **b** and **c**. Thus **A** and **a** form a pipeline interface, and likewise **B** and **b** and likewise **C** and **c**. Source **B** will deliver data to a waiting MOVE instruction only upon receipt of a token at **b**, and **C** will deliver data only upon receipt of a token at **c**.

Let us suppose that we wish to assemble elements from **A**, **B** and **C**, in that sequence, into a record at destination **D**. Let us suppose, moreover, that destination **D** is a pipeline destination. As a pipeline destination it has a token source, **d**, that produces a token for each data item delivered to **D**. Thus the bag of instructions:

A => D; B=> D; C=> D; (token) => a; d => b; d => c; d => bit bucket
will deliver values from **A**, **B** and **C** to **D** in sequence. Note that the first three instructions can do nothing until the tokens moved by the last four instructions arrive. Operation might have been faster had the instructions been issued in the order that they will execute:

(token) => a; A => D; d => b; B=> D; d => c; C=> D; d => bit bucket;
The token produced at **d** by the arrival of **A** releases **B** and likewise the token produced at **d** by the arrival of **B** releases **C**.

Unlike the data moves which do the gather task, the token moves:

d => b; d => c; d => bit bucket
do a scatter task for tokens to **b**, and **c** and the bit bucket. FLEET guarantees that such a scatter will act in sequence.

Notice that this set of instructions requires seven complete trips through the switch fabric, three for data and four for tokens. At least six of these can happen only in sequence. We may be able to get the same effect with less delay as described below.

Using multiple destinations: FLEET offers a sequence guarantee for multiple destinations that might save time. Suppose that only **A**, **B** and **C** are pipeline interfaces and **D** is not. Then the instructions:

(token) => a; A => D, b; B => D, c; C => D;
will perform the same gather task. Moreover, although I have listed these instructions in the order that they will execute, correct behavior depends only on the flow of the tokens and remains independent of the sequence in which the instructions issue.

This bag of instructions works because FLEET's cast mechanism converts the value from **A** into a token for destination **d**. Moreover, FLEET guarantees that this token can enter the destination horn only when the corresponding value from **A** is on its way through the destination horn to **D**.

This bag of instructions may be faster than the pipeline interface example because the tokens appear half way through the switch fabric at the cast mechanism in the trunk. Each token need pass only through the destination part of the switch fabric to reach its destination and release the next value. Moreover it can do this concurrently with data delivery. Thus this plan eliminates three instructions, the source part of three token moves, and overlaps the destination part of the token moves with data delivery.

Of course, this bag of instructions cannot stall if a destination is already occupied. Thus the Pipeline Interface plan, though slower, remains safer.