

UC Berkeley Computer Science

Subject: Indirection for Memory Read and Write
Date: October 28, 2005
From: Ivan Sutherland
UCIES #2005-is11

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-is04: A Dozen Problems, Ivan Sutherland, 6 September 2005
UCIES# 2005-is05: Notation Questions, Ivan Sutherland, 12 September 2005
UCIES# 2005-is06: Some Ideas About Notation, Ivan Sutherland, 13 September 2005
UCIES# 2005-is07: Literals for FLEET, Ivan Sutherland, 20 September 2005
UCIES# 2005-is08: Function or Addressing, Ivan Sutherland, 20 September 2005
UCIES# 2005-is10: More About Literals, Ivan Sutherland, 26 October 2005

INTRODUCTION

We've been talking a lot about indirection in the switch fabric. The question is whether a MOVE instruction must always fetch from the same source in the switch fabric and deliver to the same destination in the switch fabric. Another way of posing the questions is to ask whether switch fabric addresses are treated as a data type.

Greg Gibeling proposed names for three classes of indirection during our 24 October class. I shall try to record here some of the discussion that followed.

THREE CLASSES OF INDIRECTION

1st class indirection. Indirection is first class if switch fabric source and destination addresses are real data objects that can be manipulated by FLEET. Using such objects, a FLEET programmer might construct a MOVE instruction. There would be a SHIP which, given a source and a destination address, would deliver the MOVE instruction thus generated to the switch fabric for execution.

2nd class indirection. Indirection is second class if FLEET can manipulate source addresses or destination addresses but not both. If destination addresses are data objects, a special SHIP might forward a data value to a specified destination. If source addresses are data objects, a special SHIP might fetch data from a specified source. The programmer would be prevented from making a general MOVE instruction, instead being confined to a variable source or variable destination interpreted by special SHIPs.

3rd class indirection. Indirection is third class if FLEET is prevented from manipulating switch fabric addresses at all. Instead, there may be special SHIPs of two types. One type of special SHIP delivers data to a selected one of a few outputs depending on an integer, character, Boolean or token input. Standing MOVE instructions at those outputs then forward the data to chosen locations. Such a SHIP is like the "case" statement seen in many programming languages.

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 other type of special SHIP takes input data from one of a few selected inputs depending on a data value input. Such a SHIP is a multiplexer that selects from amongst several inputs. If the inputs happen to be different code bag descriptors, such a SHIP can create a conditional branch to a chosen code bag.

MEMORY READ AND WRITE

The subject of indirection came up again in connection with memory read and memory write SHIPs. Nemanja Isailovic suggested a form for memory read and memory write SHIPs that I'll describe briefly here. See also UCIES# 2005-is03, Defining Some SHIPs.

MEMORY READ

The memory read SHIP accepts some values at its pipeline input interface. These values include at least a pointer to main memory. When it is ready for another pointer and associated values, it emits an acknowledge token at its input interface. At its output interface it delivers some values each time it receives a token. The values it delivers include at least the information read from memory.

Most modern memory systems are capable of processing multiple read requests in pipeline fashion. Some memory systems keep the memory read requests in sequence, delivering values in the same sequence as the corresponding read requests. Some memory systems, however, take advantage of the differing proximity of different parts of memory to deliver values out of sequence.

For out-of-sequence delivery, the memory read SHIP must know what to do with the values as they return. Nemanja proposed that the memory read SHIP be given a switch fabric destination as part of each memory request. In other words, the input values for a memory request would include both the address of the data in memory and the switch fabric address of the destination to which the data from memory are destined.

This proposal has Greg's 2nd class form of indirection. We are forced to treat switch fabric destination addresses as a data type. The memory read SHIP can send data to a designated destination. Later on I'll suggest an alternative that avoids treating destination addresses explicitly.

There's a nice interlock of this proposal with the literals idea described in UCIES #2005-is10. Just as the fetch mechanism delivers literals automatically, so the memory read SHIP delivers data automatically to a switch fabric destination.

However, there's a critical difference between Nemanja's memory read SHIP proposal and the literals mechanism. For literals the destination is designated at compile time. Nemanja's memory read SHIP, on the other hand, gets a destination defined only at run time. That's what makes Nemanja's proposal fit into Greg's second class of indirection. A proposal that avoids this problem appears at the end of this memo.

MEMORY WRITE

Nemanja also proposed a form for the memory write SHIP. He proposed that the memory write SHIP have an input interface that takes in three values: a value to write, a memory pointer indicating where to write it, and a code bag descriptor. When the three data elements have all arrived at the input interface and the write SHIP is ready for another

input set, it issues a token at its input interface. This token means that the memory write SHIP is ready to be told what to do next; it does NOT say that the write has been done.

The output interface of the memory write SHIP must give some indication that the write has actually been done, or at least that it will be done with high certainty. In the initial description of SHIPs, #2005-is03, I suggested using a token output at the output interface for this purpose.

Nemanja claims that writing into memory is a common final act for many code bags. Therefore, he argues, a more useful output from the memory write ship would be a code bag descriptor. Indeed, the very code bag descriptor delivered to the input interface with the memory request.

This proposal fits in Greg's third class of indirection. The programmer nowhere manipulates switch fabric addresses. Instead, the programmer manipulates a code bag descriptor that will, in turn, fetch appropriate instructions for taking the next action.

POSTPONING MOVE INSTRUCTIONS

Here's rather a strange idea to avoid dealing with switch fabric destination addresses. The next section shows how we might use it to enhance Nemanja's memory read SHIP. The idea depends on a new use of a source location.

Let us review how the switch fabric normally does the instruction $X \Rightarrow Y$. First, the switch fabric uses the source address to send the instruction to the geographic place of the source, X. There the instruction waits for data to appear. The switch fabric may or may not have "used up" the bits of the source address, X, while moving the instruction to the source. In any case, the destination address bits, Y, remain in tact for use in delivering the data. Notice that this destination address has arrived at the source location of the SHIP.

Because the switch fabric destination address, Y, is located at the SHIP's source, the SHIP could copy the bits of Y and save them for later use. The rest of the instruction is no longer useful, and may be destroyed. Surely we can design a SHIP's source terminal that will recognize when an instruction has arrived, copy parts of it for later use, and destroy the rest.

When such a SHIP is ready to send something to the saved destination, it can re-construct the MOVE instruction using the saved destination bits, Y. It can then issue suitable data along with the reconstituted MOVE into the switch fabric for execution.

A PROPOSAL FOR A MEMORY READ SHIP

A memory read ship could use this technique to postpone execution of MOVE instructions until appropriate data are ready. We must make a SOURCE for the memory read SHIP of this special type. Strangely enough, that SOURCE will be a part of the SHIPs input interface. In that regard it's much like the token source that's often a part of an input interface.

Let us include such a source address in the Memory Read SHIP's input interface. For each read request, the programmer must deliver to this source address a MOVE instruction that later on will move data to the destination intended for this memory read. Instead of completing that MOVE instruction immediately, however, the memory read SHIP gobbles up the MOVE instruction, saving at least it's destination address for later

use. The memory read SHIP will associate such a destination with each read request and generate appropriate synthetic MOVE instructions as the data emerge from memory.

This proposal contemplates a non-standard use of a source address, namely a special source that postpones use of the instruction. Such source addresses don't just deliver data for the MOVE instruction to carry. Instead, such a source "eats" the MOVE instruction itself, saving at least it's destination for later use.

In return, we avoid the problem of having to deal with switch fabric destination addresses as a data type. Never does a destination addresses pass through the switch fabric as data; destination addresses remain inside instructions. However, the special source form enables a SHIP to construct its own MOVE instructions and execute such MOVE instructions only when appropriate.

A memory control SHIP that can deal with out-of-sequence memory reads must associate a destination address with each read operation. I've described a way to capture such addresses. How to associate them with the memory read operations remains up to the designer of the memory control SHIP.

To use such a SHIP a programmer must deliver two things to its input interface: 1) a memory address pointer and 2) a MOVE instruction. The MOVE instruction targets the destination for the memory data, but goes to a source location that is part of the SHIP's input interface. When the memory read SHIP has both items, it will deliver a token on its input interface token source to indicate that it has accepted the required information and is ready for the next set of data.

When the memory read SHIP has completed its memory operation it will construct the MOVE instruction needed to deliver the data. However, in keeping with the form our output interfaces, it will deliver the data and the instruction to the switch fabric only after it gets a token at the token destination of its output interface.