

UC Berkeley Computer Science

Subject: More About Literals
Date: October 28, 2005
From: Ivan Sutherland
UCIES #2005-is10

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

PURPOSE

This memo records some ideas that came out of the recent class discussions. There is only one key idea here – literals can be delivered automatically to a destination, which turns a literal into a kind of MOVE instruction with a destination address but no switch fabric source address. This idea obviates the need for a Literals SHIP.

A CHANGE IN THINKING

The problem is how to get literals into circulation. Our initial thinking involved a special LITERALS SHIP. Literals would move, we thought, to the literals SHIP as part of the instruction fetch process. From there, ordinary MOVE instructions would deliver the data to its final destination.

The new idea is to eliminate the LITERALS SHIP entirely by associating a destination with each literal. Thus literals would not only be fetched by the instruction fetch mechanism, but also inserted into the switch fabric by it for automatic delivery.

FOUR PARTS OF A MOVE

Igor drew a picture on the board that described how a MOVE instruction executes. He indicated two of the steps using a red and a green arrow. We started talking about the two actions as “Igor’s red arrow” and “Igor’s green arrow,” names worth preserving.

There are four parts to a move instruction. First, the instruction must pass to the physical location of the source data. We’ll call this step “Igor’s red arrow”. Second, until data become available at that source the instruction must wait there for the data. Such an instruction is still in the “instruction pool” waiting to execute. The instruction pool is distributed geographically and includes all those places where an instruction may wait for data. Third, the instruction, or at least its destination address, carries the data through the switch fabric to the physical location of the destination. We’ll call this step “Igor’s green arrow.” The instruction, or at least any parts of it not required to decode the destination, waits at the

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.

destination for space to become available there. Finally, when there's space at the destination the data are delivered, along with any remaining bits of the destination address.

I find it useful to think of the instruction as "carrying" the data through the switch fabric. Although the switch fabric may consume bits of the instruction as they control its path, it may also be useful to carry such "used" bits along with the instruction, and possibly other meta-information about the instructions, e.g. a sequence number, all the way to the destination, if only to help in debugging.

LITERALS ARE MOVE INSTRUCTIONS

Igor pointed out that the red arrow step is unnecessary for a literal. The fetch mechanism can insert the value of the literal into the switch fabric directly without having to decode a "source" for the value. The source for a literal is the instruction memory or instruction cache, and is implied by the nature of a literal. Thus the literal can avoid the red arrow step that sends an instruction to a source.

But the literal still needs a destination. Therefore, let us associate in memory at least one switch fabric destination address with every literal. A part of the process that fetches literals will simply deliver each literal to the switch fabric along with the associated destination address. The switch fabric will complete the "green arrow" part of the MOVE.

By associating a destination with each literal, we turn a literal into a special form of MOVE instruction that has a source VALUE rather than a source ADDRESS. From the point of view of the switch fabric, this makes literal MOVE instructions less indirect than ordinary MOVE instructions. Ordinary MOVE instructions must seek data at a source address, as indicated by Igor's red arrow. A literal MOVE instruction already has the "source" data, and thus need not seek data from some switch fabric source.

ENCODING LITERALS

This still leaves open the question of how to encode literals in memory. I offer two encodings.

1) Sequential literals. Literals are stored sequentially in memory, each with a value and one or more destinations. Such literals differ from ordinary instructions only in that the switch fabric source address is replaced with a literal value. They are treated as concurrent, i.e. the literals in any group may be delivered in any sequence.

Depending on the type of literal, e.g. integer, character, Boolean, etc., the spacing of such literal MOVE instructions may be uneven. We could pad out shorter literals to make them all the same size, thus making it easier to start processing such a "bag" of literal MOVEs in the middle of the bag.

We may wish to treat a set of literals as a "bag" of literal instructions. Like regular MOVE instructions, the literal instructions are all concurrent. Thus one might have two forms of code bag descriptor, one of which identifies ordinary MOVE instructions, and the other identifies a set of literal MOVE instructions.

2) Pointer into a table. Literal MOVE instructions are stored with ordinary instructions and have the same length as ordinary MOVE instructions. Literal MOVE instructions can be interspersed with ordinary instructions because all are the same length. In a literal MOVE instruction, the bits that might otherwise specify a switch fabric source address specify instead a displacement in a literals block for this bag of instructions.

This has the advantage of holding constant the length of MOVE instructions and separating from them the values that may possibly have variable length. Our existing plan for the “code bag descriptor” includes a separate pointer to the literal data. We leave it up to the fetch unit to fetch the literals themselves along with the instructions and to launch the literal MOVE instructions into the switch fabric.

We might combine the pointer to the literals block with the memory limit of the code bag itself. Using a single pointer to serve double duty like this saves bits but has two disadvantages. First, code bags may not share literals area. Second, a code bag cannot overlap another code bag.