

UC Berkeley Computer Science

Subject: Parallel Switch Fabrics
Date: July 10, 2006
From: Ivan Sutherland
UCIES #2006-is24

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

INTRODUCTION

FLEET has several data types. This memo proposes a separate switch fabric for each data type with a crossover network connecting them at the trunk. Separating the switch fabrics allows tokens and integers, for example, to move concurrently. A suitable instruction interpretation mechanism may also issue instructions concurrently, thus hiding, for example, the time of a token instruction inside the time of an integer move instruction.

The crossover network serves several functions. First, it broadens the trunk to speed the flow of information through concurrent channels. Second, it connects any source to any destination. Third, it casts one data type into another, converting them as they pass through according to rules. This memo offers an example of some such rules. Converting data types permits the programmer to send, for example, an integer to an integer destination and also to a token destination to initiate some new process.

DATA TYPES

For this memo I shall deal with only the data types listed below:

TOKEN
BOOLEAN
INTEGER
CODE BAG DESCRIPTOR (CBD)

The ideas presented here generalize to other data types not considered here, including CHARACTER, MEMORY POINTER, LONG, FLOAT, and so forth. To retain simplicity, this memo treats only a few data types, leaving extensions to the future.

THE IDEA – Figure 1

Figure 1 shows a FLEET with separate horns and funnels for four data types. The SHIPs, green in the figure, are split so as to appear in two places: their outputs, which are sources, appear as the tall green rectangle at the left side of the drawing and their inputs, which are destinations, appear as the tall green rectangle at the right side of the drawing. The dotted lines at the extreme left and right of the tall rectangles representing SHIPs indicate where the two rectangles join.

The instruction fetch mechanism, red in the figure, appears as a tall rectangle at the left of the figure. It delivers each instruction into one of the instruction horns. The instruction fetch mechanism uses the type of the source address of each instruction to select the horn to use for it. Instructions with a token source, for example, pass into the token instruction horn, shown near the top of the figure. The instruction fetch mechanism may deliver several different instructions concurrently to several of the instruction horns.

The black squares in the figure are called “instruction and data rendezvous” (IDR) units. Each IDR unites an instruction with its data, and delivers the data and its destination addresses into the source funnels. The output from an IDR occurs only after it has both an instruction and data. The IDR can hold an instruction at the data source until data for that instruction become available.

The IDR boxes also deal with repeated and standing instructions. Each IDR contains a mechanism to store standing instructions, forwarding duplicate destination data with each new data value presented by the SHIP. Each IDR contains a counter for repeated instructions so that it will forward only as many data values as required by the repeat count. Each IDR also contains circuits that detect the OOB values that will kill standing instructions. The “instruction pool” includes the IDR boxes as well as the instruction horns.

The sets of three black dots, ellipsis marks, indicate that although the IDR boxes appear in groups of three, there are more of them than shown. The number of outputs from each instruction horn corresponds to the number of sources of that type of data. Each source has an instruction horn output, an IDR box and a source funnel input.

The IDR boxes deliver data and destination addresses into the source funnels. Because the data for each funnel differ in width, the funnels likewise differ in the number of bits in their data paths, although the figure fails to show that difference. Each source funnel carries data values and the destinations for those data values together towards the trunk. Each source funnel contains arbiters so that it delivers its inputs one at a time to its output on a first-come-first-served basis.

The crossover and type adjustment box in the center of the figure replaces the “trunk” seen in earlier diagrams. A later section will describe its action in more detail. It uses the destination addresses delivered by the source funnels to route each data value to its appropriate destination horn or horns. It delivers individual data items and their addresses to their destination horns, duplicating data if required.

The destination horns deliver data to the destinations which are the inputs to the SHIPs. The ellipsis marks at the right side of the figure indicate that although the figure shows only three destinations for each data type, there will be as many outputs from the destination horns as there are destinations for that type of data.

CROSSOVER AND CASTING

The crossover and casting mechanism does three things. First, it deals with delivery of a single datum to multiple destinations. Recall that each MOVE instruction may have as many as three destinations. Thus each datum that arrives from a source funnel may have one, two, or three associated destinations. The crossover and casting mechanism duplicates the data as required and forwards it to its destinations. Thus a single data value arriving at an input to the crossover and casting mechanism may result in up to three output values from it.

A second task for the crossover and casting mechanism is to deliver data values to the proper destination horn. For example, let us consider what would happen if a programmer wished to use an integer value not only as an integer, but also as a token to initiate some action. In this case, the integer would arrive at the output of the integer funnel with two destinations, one for an integer destination and the other for a token destination. Given such a destination pair, the crossover and casting unit will deliver an output to both the integer and the token destination horns for concurrent delivery to the two destinations.

A third task for the crossover and casting mechanism is data type conversion or “casting.” In the example of the previous paragraph, what actual value should go into the token destination horn to represent the integer there? Because as a token, only the existence of the integer matters and not its actual value, converting an integer to a token is easy – the integer value vanishes. It’s harder to understand some other casts. For example, how should a boolean value appear at an integer destination? The crossover and type casting mechanism includes FLEET’s built in casting mechanisms.

We can represent the rules for type casting between N types as an N by N matrix of rules. Table 1 proposes such a set of rules for type casting in the four data types considered by this memo. A larger table will be required to accommodate more data types.

| Table 1: Some type casting rules | | | | | |
|---|-----------|--------------|----------------------------|--------------------------------|-------------------------------|
| from | to | token | boolean | integer | CBD |
| token | | SAME | FALSE | ZERO | ONE STORED CBD |
| boolean | | TOKEN | SAME | if(TRUE) then ONE else ZERO | ONE OF TWO STORED CBDs |
| integer | | TOKEN | if (>0) TRUE else FALSE | SAME | THEEE STORED CBDs by -,0,+ |
| CBD | | TOKEN | FALSE | SEE TEXT | SAME |

Let me share some of my reasoning in choosing the entries of this table. The diagonal entries require no casting at all. I have chosen arbitrarily to treat FALSE, ZERO, and TOKEN as a group; each of them converts symmetrically into the others. My other arbitrary choices are to cast TRUE into ONE and all positive non-zero integers into TRUE.

I dodged the problem of casting into code bag descriptors (CBDs) by assuming a special SHIP that stores at least three CBDs for this purpose. A token casts into a particular CBD which the program must deliver to a special SHIP associated with the

casting mechanism. A boolean casts into one of two CBDs, and an integer casts into one of three CBDs according to its value with respect to ZERO. The programmer must likewise deliver the values of these CBDs to the special SHIP.

I don't know how best to cast CBDs into integers. We can use this cast to extract from the CBD the locations in memory to which it refers. If the CBD format is shorter than the integer format, we could just copy the bits as if nothing had happened. It may be valuable, however, to do some offset calculation to find the actual location of the instructions and literals in memory. If there are too many bits in a CBD for an integer to hold, we can cast the CBD into a vector of integers. No doubt we should provide somewhere a mechanism, probably a SHIP, that will produce a CBD from such a vector.

CROSSOVER AND CAST UNIT – Figure 2

The crossover and cast unit consists of several layers as shown in Figure 2. Data elements and their three destination addresses enter at the left of the figure. They first enter two-way branch circuits shown as small triangles at the left of the figure. These two-way branch units forward the data and destination addresses to either one or the other of their outputs or forward duplicate copies to both outputs. If all of a data element's destinations are of the same type as the input channel, the data and the addresses go directly to the two-way merge elements shown as small triangles at the top of the figure. If all are of a type that differs from the input channel, nothing goes directly to the output merge elements, but instead the data and its destinations go to the three-way branch units shown as the larger triangles near the left of the figure. If some destinations are of the same type and some of a different type, duplicate copies of the data and destinations go both directly to the merge elements at the top of the figure and to the three-way branch units.

The three-way branch units at the left of the figure likewise forward inputs to any of their outputs or can forward duplicate copies to more than one of their outputs. If all destinations are the same, data go only the corresponding output. If there are than one destination type, duplicate copies of the data and destinations go to multiple outputs.

The squares in the middle of the figure represent the cast units. These units modify the data according to the cast rules chosen for the system. Each such cast unit changes the value of the data but not the destinations. The data output width of each cast unit is appropriate to the form into which the data are cast. The cast units differ in complexity depending on the casting rules that each implements.

The three-way merge units shown as the larger triangles at the top of the figure combine their inputs on demand, using arbiters. Likewise, the two-way merge units represented as the smaller triangles at the top of the figure also combine their inputs on demand. Thus the merge action at the top of the figure is on a first-come-first-served basis.

Finally the data reach the top of the figure. If the data are destined for more than one type of destination, the steps previously described will have duplicated the data and sent it to more than one output channel. The casting mechanism will ensure that the value at each output channel is of a suitable form.

Now, however, it is possible that the data must go to more than one destination of a given type. For example, consider a data element with two integer destinations. It will arrive as a proper integer on the integer output channel with two integer destinations. There it will pass into the larger rectangular boxes at the top of the figure labeled "duplicate." If there remains more than one address, these units duplicate the data and send it out more than once as required.

LATENCY IN CROSSOVER AND CAST

Notice that the path through the crossover and cast network is shortest for values that go only to an output channel of the same type as the input. Such values avoid casting and pass only through a two-way branch and a two-way merge on their way to the duplicate box at the top of the figure.

Of course, data that must change from one form to another must pass through the appropriate cast unit. In addition, however, to get to the proper cast unit, it passes through a three-way branch and a three-way merge. Thus data to be cast will have substantially longer latency than cast-free data.

In the design of Figure 2, I have chosen to minimize the latency for cast-free data at the cost of additional latency for data that require casting. Other choices are, of course, possible. Consider that the three-way branch and merge units probably use two layers of two-way branch and merge. Instead of a two-way merge followed by a three-way merge, one might have used a symmetric four-way merge. Similarly, the branch mechanism might have been symmetric as well.

I chose to illustrate the un-balanced branch and merge to make a point. If there is one path that is commonly used, it is possible to reduce its latency at the expense of the latency of other paths. I prefer the scheme shown here not only because it illustrates an unbalanced tree, but also because it might easily be expanded to more data types.

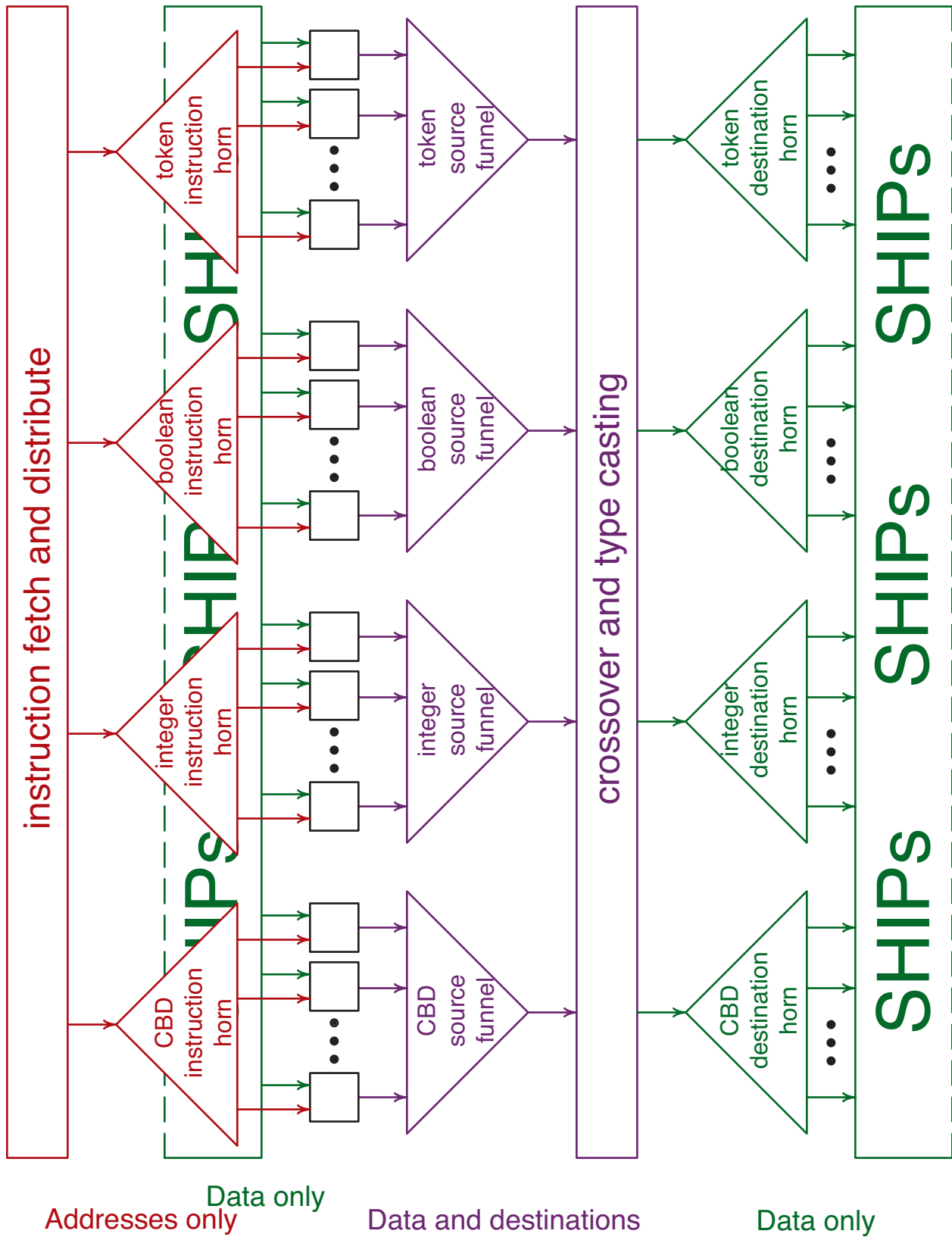


Figure 1: Multiple horns and funnels

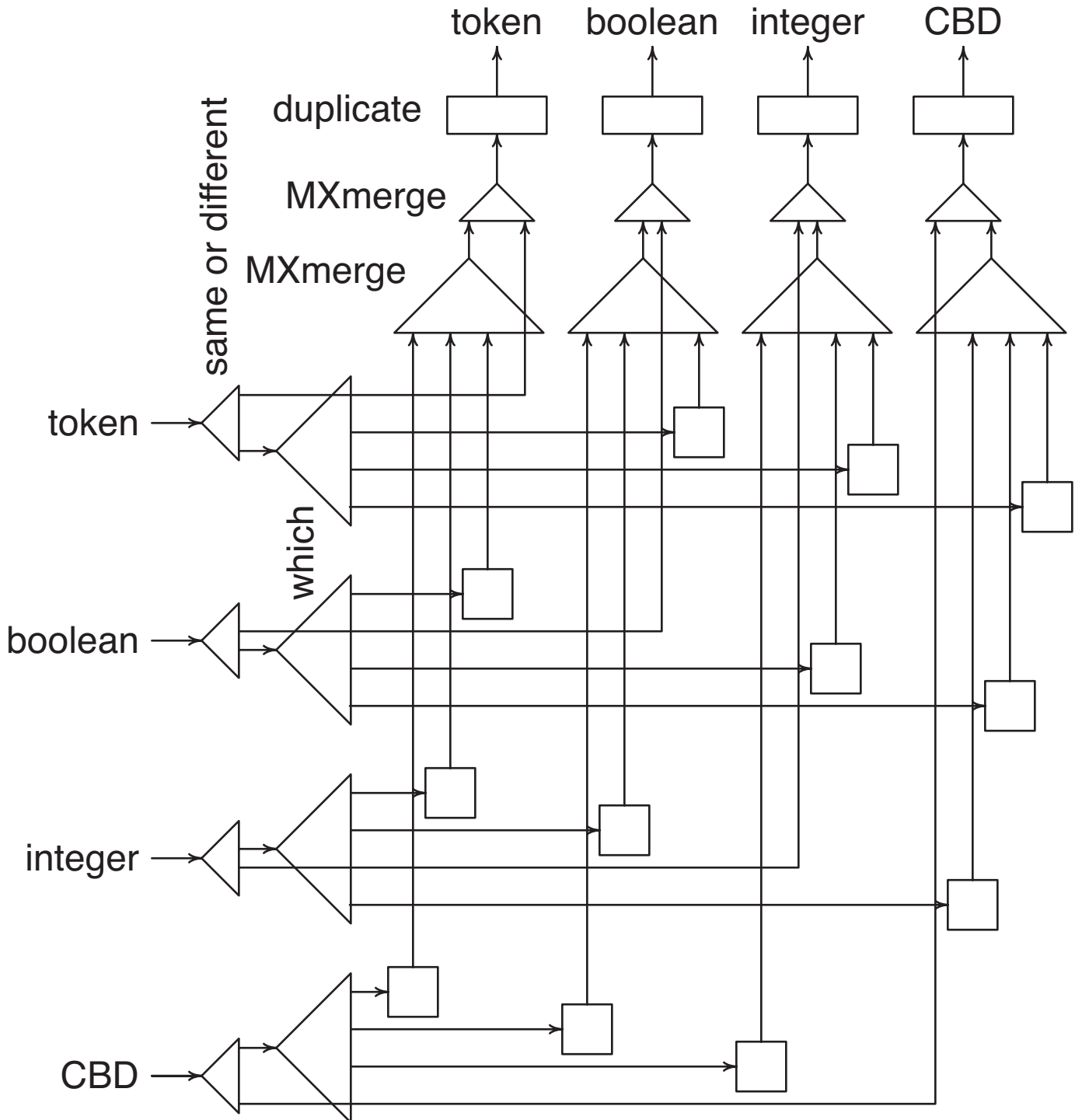


Figure 2: The crossover and cast unit