

# Opcode Ports

Adam Megacz

August 7, 2007

## **Abstract**

This memo defines a relationship between boxes which allows the programmer to deliver a data to one box and a compile-time constant to the other box.

## 1 Motivation

Quite often, a single ship will be capable of performing many different operations. A good example is the Alu ship. This ship can add, subtract, compare, and do other things as well.

In such situations, the usual approach is to give the ship an “opcode port” to which one of several special values is sent. Each of these special values designates one of the operations. For example, 1=add, 2=subtract, and 3=compare.

While writing Fleet programs, we realized that these opcodes were causing two undesirable side effects:

1. They were adding bloat to code bags, in the form of additional literals
2. When a single ship is shared by two logical tasks within a code bag – for example, alternating between adding and comparing – we found that we had to send many tokens around in order to ensure that the correct opcode would arrive along with the correct datum. These tokens added even more code bloat, switch fabric traffic, and – most importantly – unnecessary serialization.

## 2 Who Knows The Answer?

To solve this problem, we asked ourselves this question: “when a value arrives at the Alu’s input, *who knows* if it should be added, subtracted, or compared?”

After reviewing many programs, the answer turns out to be *the instruction which sent the data from its outbox*. It appears that when such situations arise, it is usually possible to point to a `sendto` instruction in the program which always sends data items that are to be added, or to point to some other `sendto` instruction whose data items are always compared.

Therefore, we would like to have a way to send a small value – typically

only a few bits wide – from the *instruction stream* at the sender to the opcode inbox near the recipient inbox.<sup>1</sup>

### 3 The Solution: Opcode Boxes

In order to implement this plan, we introduce the concept of an “Opcode Box”. Being an opcode box is sort of like being a “little brother” or being a “big sister” – it’s only meaningful with respect to someone/something else.

In Fleet assembly and when naming ship ports, we will adopt the following convention:

- Every box name must begin with either “in” or “out” (both case sensitive).
- Opcode boxes names must begin with “in” and must end with the two characters “Op” (case sensitive).
- *Only* opcode box names may end with the two characters “Op” (case sensitive).
- For every opcode box, there must be some other box which has the same name but without the last two characters Op. This “other box” is also allowed to have an out prefix rather than an in prefix.<sup>2</sup>

We say that a box “is the opcode box of” some other box. For example, “inDataOp is the opcode box of inData.” We also say that “this box’s opcode box is that box.” For example, “inData’s opcode box is inDataOp.”

This convention has the desirable property of being self-documenting; it is immediately obvious which boxes have opcode relationships with each other, and what the “direction” of the relationship is.

---

<sup>1</sup>We previously solved this problem with something called “virtual ports.” They were ugly and gross, and we will not mention them ever again.

<sup>2</sup>however, both may not exist – for example, if there is a box named inDataOp, then inData and outData may not *both* exist

## 4 Writing Assembly Code Using Opcode Boxes

Wherever a box address is allowed to appear in Fleet assembly code, the programmer may include an opcode. For example, consider this program:

```
outPort: sendto inData
```

The program above will transmit a single datum from outPort to inData.

However, if the programmer would also like to have the opcode 3 arrive at inDataOp, the programmer could write:

```
outPort: sendto inData(3)
```

Note that tokens may also bear opcodes; for example:

```
outPort: ack outData(3)
```

In this situation, a *token* would arrive at outData, whereas a *data item* would arrive at inDataOp.

Other memorandums and documents may describe new methods for defining and using constants. Any such mechanisms are valid within the parenthesis. For example, if some future memo XQ999 were to decree that “FF+!!!..,?” was the optimal way to describe the constant for addition, then the following program would be valid:

```
outPort: sendto inData(FF+!!!..,?)
```

However, this is unlikely.

## 5 Behavior Guarantees

Note that for a given box/opcodebox pair, we now have three ways of sending data:

1. Send data to inData
2. Send data to inDataOp

3. Send data to `inData` and a constant value to `inDataOp`

The only guarantee this document makes is the following: when two different sources both send a data item using technique #3 above, the two data values will enter the data fifo at `inData` in the same order in which their corresponding constant values enter `inDataOp`'s data fifo.

## 6 Implementation

The following section is *non-normative*. It is merely a suggestion to assist in understanding the opcode mechanism. Actual Fleet processors need not work as described below, although this is one example of a valid implementation.

Perhaps the most straightforward way to implement a switch fabric is using binary (two-way) horn elements, stripping a single bit off of the destination address at each horn element. In such a switch fabric, the opcode box functionality can be implemented by doing the following:

- Ensure that the data fifos of `inData` and `inDataOp` are each reachable only through a single horn element, and that they share that horn element. Another way of describing this is that they share a common "leaf node" of the data horn.
- Modify this horn element so that it examines *two* bits rather than one. If the bits are:
  - 00 – send the datum to `inData`
  - 01 – send the datum to `inDataOp`
  - 1X – send the datum to `inData`, and send bit X plus any remaining unused address bits to `inDataOp`