

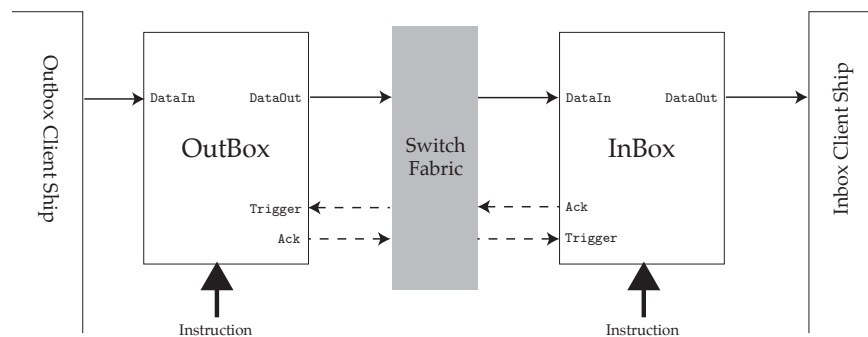
# AM10: Boxes

Adam Megacz and Ivan Sutherland

November 8, 2006

## 1 Inboxes and Outboxes

Inboxes and Outboxes are now identical structures. Each takes an instruction from a source address on the instruction horn. Each has a data input and data output, as well as a token input called `Trigger` and a token output called `Ack`. The only difference between inboxes and outboxes is how they connect to their client ships and the switch fabric.



Inboxes and Outboxes

An inbox is present on every data input to every ship, and an outbox is present on every data output. Raw token inputs and outputs do *not* have boxes attached to them; examples of these include the “done” token emitted by a memory ship and the “release codebag” token input to a fetch unit.

At first glance this may seem to add a great deal of complexity. However, as will be explained later in this memo, the resting state of a Fleet has all inboxes configured to be transparent, so the programmer can ignore inboxes entirely unless she explicitly configures them to perform some action.

## 2 Instruction Ordering

Recall that if a codebag contains two or more instructions which have the same source address, those instructions are guaranteed to be executed at their source ship in the order in which they appear in the codebag.

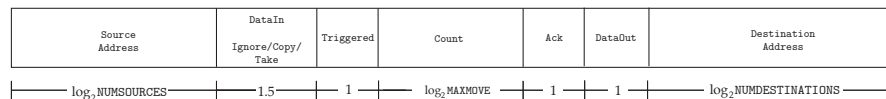
This property is important for understanding the usefulness of some instruction forms. In particular, many instruction forms are only useful as a “barrier” in the stream of instructions being sent to a particular ship.

## 3 Instruction Fields

An instruction tells a box what **action** to take. This action involves possibly copying or consuming a datum from its DataIn input and possibly sending that datum on its DataOut output. The instruction may also specify whether or not the box must wait for a token on its Trigger input before performing this action, and whether it should emit a token on its Ack output to announce that the action has been completed.

The count field (formerly ZOMA) can now hold only values  $1..MAXMOVE$  and  $\infty$ . A count=0 is no longer valid; its role has been subsumed by the other instruction fields.

A conceptual Fleet instruction format is shown below.



Anatomy of a FLEET Instruction

Note that the bits need not occur in this order. Indeed, the bits are likely to be coded more efficiently, since the DataIn field has three possible values, and not all combinations of DataIn, Ack, DataOut, and Trigger are valid.

The new fields include:

- **Trigger**  
If set, the box will wait for a token on its Trigger input and consume it before taking action.
- **DataIn**  
If set to Ignore, the box will not wait for a value to arrive on its Data input before taking action. If set to Take, the box will wait for a value to arrive on its Data input and will take action by consuming it. If set to Copy, the box will wait for a value to arrive on the data input, take action by copying it, and will then *leave the value there*.

- **DataOut**  
If set, the data read from `DataIn` will be emitted on `DataOut` as part of the box's action. In the case of an outbox, where the `DataOut` port sends data to the switch fabric, the datum will be emitted with a destination address equal to the destination address in the instruction. If this bit is not set, no value will be emitted on the `DataOut` port when the box takes action. If a datum is emitted, the box does not complete taking action until the emitted datum has been accepted by the ship or switch fabric.
- **Ack**  
If set, the box will emit a token on its `Ack` output after taking action. This token will be emitted with a destination address equal to the destination address in the instruction.

### 3.1 Count Field

The remainder of this document focuses on describing the behavior of various instruction forms under the assumption that the `count` field is set to 1.

Instructions with `count=n` where  $n < \infty$  behave in a manner indistinguishable from  $n$  identical, consecutive instructions with `count=1`.

Instructions with `count= $\infty$`  behave as an instruction with `count=1` which is not consumed from the instruction input until some other instruction arrives. When a second instruction arrives, the `count= $\infty$`  instruction disappears, and the second instruction is presented to the box. This requires an arbitration between the box's request for an instruction and the arrival of an instruction following a `count= $\infty$`  instruction.

## 4 Instruction Forms

The table below gives opcode names for each combination of possible values for DataIn (along the top) and DataOut/Ack (along the side).

Each of the 10 instructions in the table below comes in two forms: a triggered form in which the Trigger field is set, and a non-triggered form in which the Trigger field is not set.

	DataIn Ignore	DataIn Copy	DataIn Take
	nop	wait	discard
Ack	nop+ack	wait+ack	discard+ack
DataOut		copy	move
DataOut +Ack		copy+ack	move+ack

Opcodes for Instruction Forms

Note the empty cells in the lower left hand corner. These correspond to instructions that attempt to emit data without first accepting data as input and are therefore meaningless.

## 5 Instruction Tutorial

The behavior of the instructions is completely specified by the fields described in the previous sections, their possible values, and the meaning of those values.

However, for the reader's benefit, this section gives a sentence or two of prose explaining each of the 20 combinations in detail. Nicknames for some instruction combinations are also included in parenthesis.

- nop ("no operation")  
This instruction does nothing. It does not even wait for some event to take place. Rarely useful.

triggered nop (“discard trigger”)

This instruction discards a token from the Trigger input.

- nop+ack (“announce instruction execution”)

This sends a token when the instruction executes. This is useful for tearing down standing moves because it produces a token to signal that the standing move has been torn down. The token might go to the Fetch unit to start another code bag.

triggered nop+ack (“token reflector”)

This waits for a token on the Trigger input and retransmits (“reflects”) it to another destination as an acknowledgement. Example use case needed.

- wait (“wait for data”)

This operation stalls the instruction stream until a value appears at the data input. The value is undisturbed. Example use case needed.

triggered wait (“wait for token and data”)

This operation stalls the instruction stream until a value appears and a token arrives at the Trigger input. The data value is undisturbed; the token is consumed. No output is produced. Example use case needed.

- wait+ack (“poll”)

This instruction sends out a token when a value becomes available. The value is undisturbed.

triggered wait+ack (“triggered poll”)

This instruction sends out a token when a value becomes available and a token is present on the Trigger input. The value is undisturbed but the token is consumed. This instruction can be chained across multiple boxes to detect the presence of data on all of the boxes.

- copy

On an outbox, this is the “copying move” described earlier. On an inbox, this creates a “sticky” data input which is fed to the client ship more than once. Useful for duplicating outputs or supplying constants as inputs.

triggered copy

Like a copy, but the box will wait for and consume one token from the Trigger input for each copy of the datum it produces. Useful for providing constants on *all* inputs of a ship – the Trigger input acts as a rate control.

- copy+ack

Not meaningful for an outbox. On an inbox this behaves like a copy instruction, but emits a token *for each copy of the datum* that the ship consumes.

triggered copy+ack

Not meaningful for an outbox since outboxes have only one output. Not

meaningful for an inbox since inboxes have no trigger input.

- `discard (data)`

Waits for and discards one datum from the data input.

`triggered discard (data)`

Waits for a datum and a Trigger token, then discards both.

- `discard+ack (“convert data to token”)`

Waits for a datum, discards it, and sends a token. This could be thought of as “converting” the datum into a token and sending the token on to some other destination.

`triggered discard+ack`

This sends out a token when it gets both data and a trigger token. Both data and trigger token are consumed.

- `move`

This is the “simple” move previously described.

`triggered move`

This instruction will consume a datum and token, and will pass the datum on to the data output.

- `move+ack`

Not meaningful for an outbox. This provides flow control on an inbox; it emits a token each time the ship consumes a datum.

`triggered move+ack`

Not meaningful for an outbox since outboxes have only one output. Not meaningful for an inbox since inboxes have no trigger input.

## 6 How the Inbox and the Outbox Differ

So far we have seen how alike the inbox and the outbox are. Now let’s look at their differences.

The outbox has, in fact, only one output port that serves as output for both `DataOut` and `Ack`. Thus, for the outbox, the bottom row of the table must be omitted because `DataOut` and `Ack` are mutually exclusive.

For the outbox, the trigger input can provide for the flow control of a pipeline interface. The `Copy` instruction in the middle column sends out the data from the ship but never tells the ship about the copy, and so the data value remains undisturbed at the ship’s output for use by a subsequent instruction.

The inbox has, in fact, no Trigger input.<sup>1</sup> Thus, for the inbox, there is no “triggered” form for the instructions; only the 10 non-triggered forms are

---

<sup>1</sup>this is still up for debate – it seems that `triggered copy` is useful on an inbox

valid.

For the inbox, the four boxes in the lower right corner of the table: `copy`, `move`, `copy+ack`, and `move+ack` are the most useful. The forms with `+ack` let the inbox deliver tokens back to a sender. The `copy` form at the inbox makes the data sticky, obviating the need to re-send constants to a ship. The `ack` token appears only after the ship has accepted the data.

## 6.1 Inbox Initialization

A Fleet boots up with a standing ( $\text{count}=\infty$ ) `move` active at every inbox. This tells all inboxes that until further notice they should simply consume data from the switch fabric and pass it along to the ship without waiting for a `Trigger` token and without producing an `Ack` token.

By convention, a programmer may assume that the Fleet is in this state when invoked from unknown code, and should return the inboxes to this state before invoking unknown code.

This convention serves two purposes:

- It simplifies Fleet programming by letting the programmer ignore inboxes unless they are needed for some task.
- It ensures that non-standing moves ( $\text{count}<\infty$ ) can be performed with only a *single* instruction, which is sent to the outbox. Without this convention, two instructions would be required for every move – one sent to the outbox and one to the inbox.

## 7 Some Comments About Multiple Inboxes

Consider a ship like the Adder ship that has three inputs, A B and C. Suppose, further, that the ship waits until all three inputs are available and then uses and acknowledges all three. Each of the inputs has an inbox. Suppose further, that a `move+ack` instruction is in each of these inboxes. When do the inboxes emit tokens?

The inbox emits an `Ack` token when the ship actually accepts the data proffered to it by the inbox. The `Ack` token is, in effect, a form of the ship's acknowledgement of receiving data. The Adder ship with three inputs will acknowledge all three of its inboxes concurrently, permitting them each to emit a token. The three tokens will appear concurrently. and we can easily send these three tokens to three different pipelined data sources. However, suppose we need only one token, should we send the other two to the bit bucket?

No. To get only a single token from three inboxes connected to concurrent inputs of the same ship, give two of the inboxes a simple move and give the third a move+ack. The third inbox will emit a token to indicate that the ship has acknowledged all three inputs.

## 8 Example Program

This is the very lowest level at which Fleet can be programmed – explicit move instructions to individual box ports. A forthcoming macro layer will make it easy to set up these flow-controlled connections with minimal effort.

```
// take the receiver inbox out of pass-through mode, then release GO
GO
      -> fetch.codebag
nop+ack receiver -> fetch.release

GO: {
  // send 3 data without waiting for a token
  move sender.data -[3]-> receiver.data

  // send arbitrarily more data, but wait for a token each time
  triggered move+ack sender.data -[*]-> receiver.data

  // route receiver tokens back to sender
  move receiver.ack -[*]-> sender.trigger
}

STOP: {

  // eat three trigger tokens at the sender; third one releases CLEANUP
  triggered nop sender.out -[2]-> bitbucket
  triggered nop sender.out      -> fetch.token
  CLEANUP                       -> fetch.codebag
  NEXT_CODE_BAG                  -> fetch.codebag

  CLEANUP: {
    // tear down standing move on receiver; use token to release successor
    nop+ack receiver.ack      -> fetch.token
  }
}
```

Note that if only one datum is to be in flight, then GO does not need to be a separate codebag – the nop+ack receiver token can be used to prime the pump. This trip through the fetch unit can also be avoided if a “token multiplier” ship is available which takes the nop+ack receiver token as input and produces 3 tokens as output.