

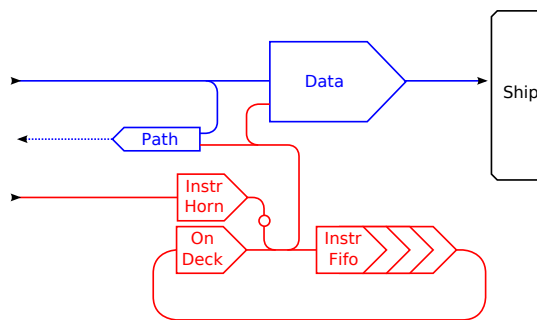
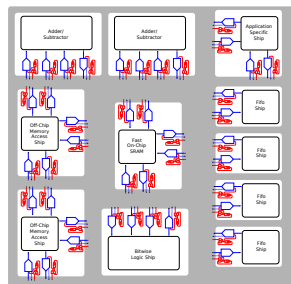
The FleetTwo Dock

April 17, 2008

Abstract

Changes:

- 17-Apr Made all instructions except `setOuter` depend on `0LC>0`
Removed ability to manually set the C flag
Expanded predicate field to three bits
New literals scheme (via shifting)
Instruction encoding changes made at Ivan's request (for layout purposes)
Added summary of instruction encodings on last page
- 07-Apr removed "+" from "potentially torpedoable" row where it does not occur in Execute
- 06-Apr extended `LiteralPath` to 13 bits (impl need not use all of them)
update table 3.1.2
rename S flag to C
noted that `setFlags` can be used as `nop`
- 29-Mar removed the L flag (epilogues can now do this)
removed `take{Inner|Outer}LoopCounter` instructions
renamed `data` instruction to `literal`
renamed `send` instruction to `move`



1 Overview of Fleet

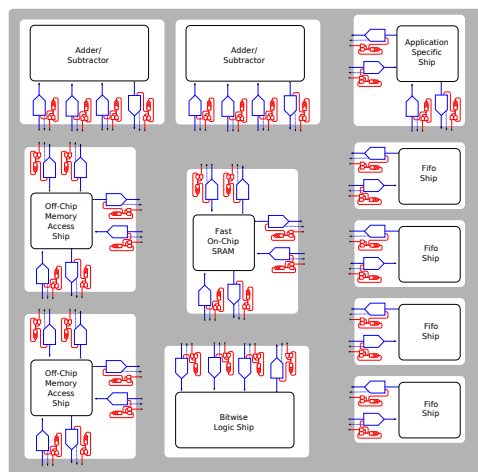
A Fleet processor consists of a *switch fabric* with several functional units called *ships* connected to it. At each connection between a ship and the switch fabric lies a programmable element known as the *dock*.

A *path* specifies a route through the switch fabric from a particular *source* to a particular *destination*. The combination of a path and a single word *payload* is called a *packet*. The switch fabric carries packets from their sources to their destinations. Each dock has two destinations: one for *instructions* and one for *data*. A Fleet is programmed by depositing packets into the switch fabric; these packets' paths lead them to the instruction destinations of the docks.

When a packet arrives at the instruction destination of a dock, it is enqueued for execution. Before the instruction executes, it may cause the dock to wait for a packet to arrive at the dock's data destination or for a value to be presented by the ship. It may present a data value to the ship or transmit it for transmission to some other destination.

When an instruction sends a packet into the switch fabric, it may specify that the payload of the packet is irrelevant. Such packets are known as *tokens*, and consume less energy than data packets. From a programmer's perspective, a token packet is indistinguishable from a data packet with a unknown payload.

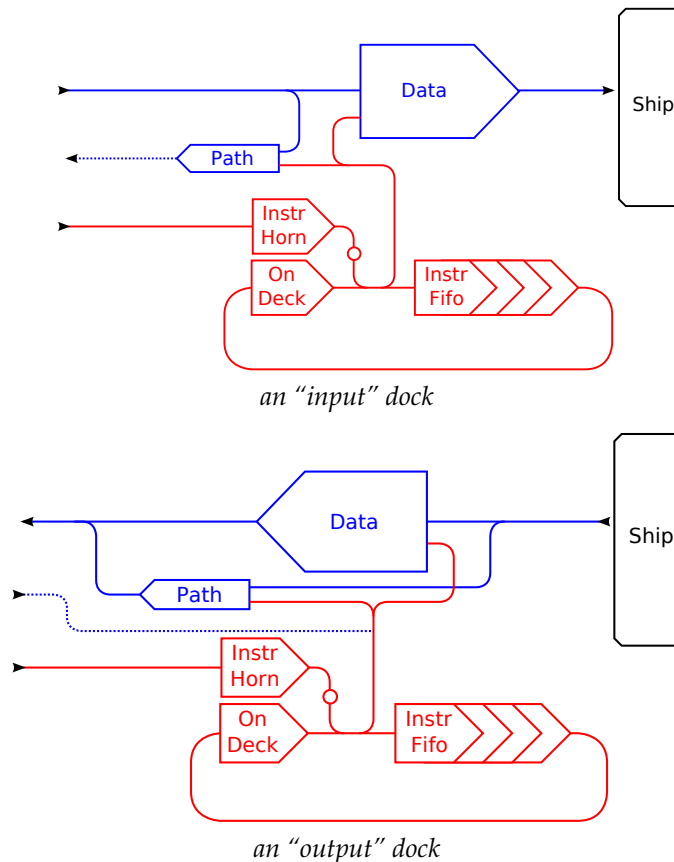
In the diagram below, the red wires carry instructions and the blue wires carry data; the switch fabric (gray area) carries both. Notice that the red (instruction) wires do not contact the ships. This is an advantage: ships are designed without any consideration for the instructions used to program their docks.



Overview of a Fleet processor; gray shading represents a packet-switched network fabric; blue lines carry data, red lines carry instructions.

2 The FleetTwo Pump

The diagram below represents a *programmer's* conceptual view of the interface between ships and the switch fabric. Actual implementation circuitry may differ substantially. Sources and destinations that can send and receive only tokens – not data items – are drawn as dashed lines.



The term *port* refers to an interface to the ship, the *dock* connecting it to the switch fabric, and the corresponding sources and destinations on the switch fabric.

Each dock consists of a *data latch*, which is as wide as a single machine word and a *pump*, which is a circular fifo of instruction-width latches. The values in the pump control the data latch.

Note that the pump in each dock has a destination of its own; this is the *instruction destination* mentioned in the previous section. Note that unlike all other destinations, there is no buffering fifo guarding this one. The size of these fifos are exposed to the software programmer so he can avoid deadlock.

3 Instructions

In order to cause an instruction to execute, the programmer must first cause that instruction word to arrive in the data latch of some output dock. For example, this might be the “data read” output dock of the memory access ship or the output of a fifo ship. Once an instruction has arrived at this output dock, it is *dispatched* by sending it to the *instruction port* of the dock at which it is to execute.

Each instruction is 26 bits long, which makes it possible for an instruction and an 11-bit path to fit in a single word of memory. This path is the path from the *dispatching* dock to the *executing* dock.



Note: the instruction encodings below are simply “something to shoot at” and a sanity check to make sure we haven’t overrun our bit budget. The final instruction encodings will probably be different.

All instruction words have the following format:



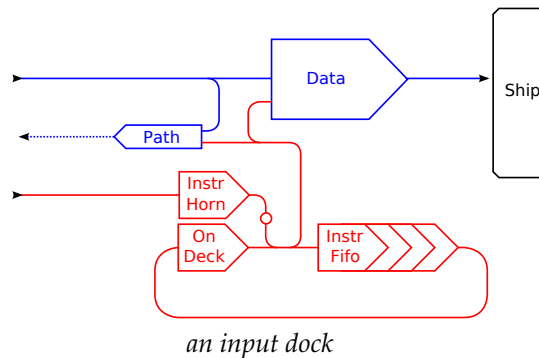
Each instruction word is called a *micro instruction*. Collections of one or more micro instruction are known as *composite instructions*.

The I bit stands for *Interruptible*. The OS (“One Shot”) bit indicates whether or not this instruction is part of an outer loop. Both of the preceding bits are explained in the next section.

The abbreviation P stands for *predicate*; this is a two-bit code that indicates if the instruction should be executed or ignored.

3.1 Life Cycle of an Instruction

The diagram below shows an input dock for purposes of illustration (behavior at an output dock is identical).

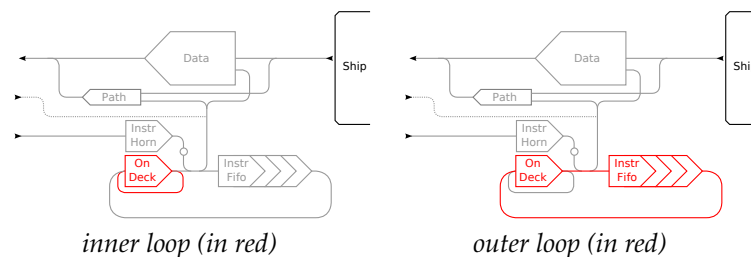


Note the circle on the path between “instr horn” and “instr fifo”; this is known as “the hatch”. The hatch has two states: sealed and unsealed. When the machine powers up, the hatch is unsealed; it is sealed by the tail instruction and unsealed whenever the outer loop counter is set to zero (for any reason¹).

When an instruction arrives at the instruction horn, it waits there until the hatch is in the unsealed state. The instruction then enters the instruction fifo. When an instruction emerges from the instruction fifo, it arrives at the “on deck” stage, where it may execute.

3.1.1 Inner and Outer Loops

A programmer can perform two types of loops: *inner* loops of only one micro-instruction and *outer* loops of multiple micro-instructions. Inner loops may be nested within an outer loop, but no other nesting of loops is allowed. The paths used by inner loops and outer loops are shown below:



Each type of loop has a counter associated with it: the ILC counter for inner loops and the OLC counter for outer loops. The inner loop counter applies only

¹this includes OLC being decremented to zero, a `setOuter` with a literal field of zero, a `setOuter` which copies a zero from the data register to OLC, or the occurrence of a torpedo

to certain “inner-looping” instructions (see the table below for details). When such an instruction reaches On Deck, if its predicate is true it will execute a number of times equal to $ILC+1$, and leave $ILC=0$ after executing. Non-inner-looping instructions and instructions whose predicate is false do not decrement ILC .

The outer loop counter applies to all instructions *except* the instruction `setOuter` with $OS=1$, because such instructions are needed to reset the outer loop counter after it becomes zero. However, predicated `setOuter` with $OS=0$ is useful for resetting the loop counter in the middle of the execution of a loop.

3.1.2 On Deck

The table below lists the actions which may be taken when an instruction arrives on deck:

	Outer-Looping ($OS=0$)					One-Shot ($OS=1$)				
	move	literal	setFlags	setInner	setOuter	move	literal	setFlags	setInner	setOuter
Wait for hatch sealed, then IF0 w/ copy of self	+	+	+	+	+	-	-	-	-	-
Potentially torpedoable	P+I	P+I	P+I	P+I	P+I	P+I	P+I	P+I	P+I	PI
Execute	P+	P+	P+	P+	P+	P+	P+	P+	P+	P
Inner-looping	P+	-	-	-	-	P	-	-	-	-

+	Only if $ILC>0$ (ie ILC is positive)
P	Only if predicate is true
P+	Only if predicate is true and $ILC>0$
PI	Only if predicate is true and $I=1$.
P+I	Only if predicate is true and $ILC>0$ and $I=1$.

Note: a non-one-shot instruction may *execute* before the hatch is sealed, but may not *fill IF0* before the hatch is sealed. The instruction will not vacate On Deck until both of these tasks are complete, so the second non-one-shot instruction in a loop will not execute until the hatch is sealed, *but the first instruction will.*

3.1.3 Torpedo

There is a small fifo (not shown) before the latch marked “Instruction Horn”; after the tail instruction seals the hatch, any subsequent instructions will queue up in this fifo until the hatch is unsealed. This is typically used as storage for a “loop epilogue” – a sequence of instructions to be executed after a torpedo arrives or the outer loop counter expires.

Each dock has a fourth connection to the switch fabric (not shown), called its *torpedo destination*. Anything (even a token) sent to this destination is treated as a torpedo. Note that because this is a distinct destination, instructions or data queued up in the other destination fifos will not prevent a torpedo from occurring.

The dock also has a *torpedo acknowledgment path latch*, which stores the path along which a token should be sent when a torpedo strikes. How this latch is set is yet to be determined.

When a data item or token arrives at the torpedo destination, it lies there in wait until On Deck holds a potentially torpedoable instruction (see previous table). Once this is the case, the torpedo causes the inner and outer loop counters to be set to zero (and therefore also unseals the hatch)² and sends a token along the path stored in the torpedo acknowledgment path latch.

3.2 Flags

The pump has three flags: A, B, and C.

- The A and B flags are general-purpose flags which may be set and cleared by the programmer.
- The C flag is known as the *control* flag, and it is set every time the data latch takes on a new value. At outboxes its value is determined by the ship; at inboxes its value is copied from an unused address bit in the destination to which the received value was sent.

Many instruction fields are specified as **three-bit predicates**. These fields contain one of eight values, indicating if an action should be taken unconditionally or conditionally on one of the A, B, or C flags:

- | | |
|------------------------|------------------------|
| • 000: if A is set | • 100: if C is set |
| • 001: if A is cleared | • 101: if C is cleared |
| • 010: if B is set | • 110: unused |
| • 011: if B is cleared | • 111: always |

²it is unspecified whether the torpedoed instruction is requeued or not; this may or may not occur, nondeterministically. It is the programmer’s responsibility to ensure that the program behaves the same whether this happens or not. We think that this will not matter in most situations.

4 Instructions

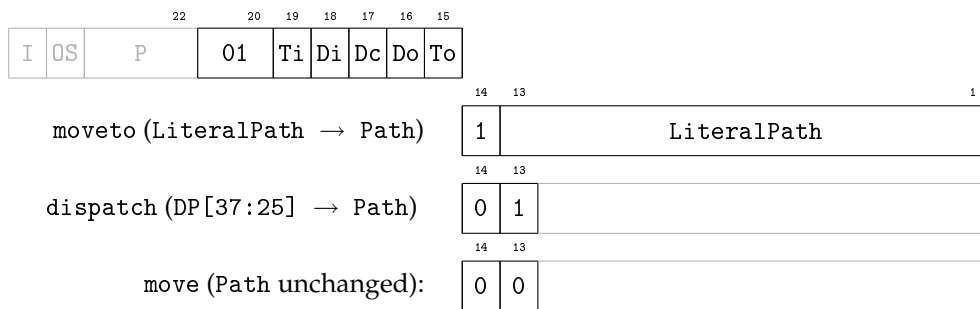
Here is a list of the instructions supported by the dock:

```

move (variants: moveto, dispatch)
literal
setFlags
setInner
setOuter
tail

```

4.1 move (variants: moveto, dispatch)



- Ti - Token Input: wait for the token predecessor to be full and drain it.
- Di - Data Input: wait for the data predecessor to be full and drain it.
- Dc - Data Capture: pulse the data latch.
- Do - Data Output: fill the data successor.
- To - Token Output: fill the token successor.

The data successor and token successor must both be empty in order for a move instruction to attempt execution.

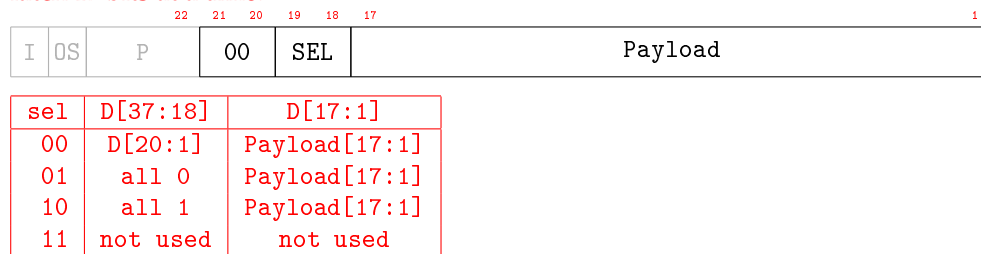
The inner loop counter can hold a number $0..MAX$ or a special value ∞ . If ILC is nonzero after execution of a move instruction, the instruction will execute again, and ILC will be latched with $(ILC == \infty ? \infty : \max(ILC - 1, 0))$. When the inner loop counter reaches zero, the instruction ceases executing.

4.2 literal

Each literal instruction carries a payload of 17 bits. When a literal instruction is executed, this payload is copied into the least significant 17 bits of the data latch, and the remaining most significant bits of the data latch are loaded with either:

- All zeroes
- All ones
- The value formerly in the least significant bits of the data latch

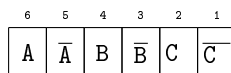
In this manner, large literals can be built up by “shifting” them into the data latch 17 bits at a time.



4.3 setFlags

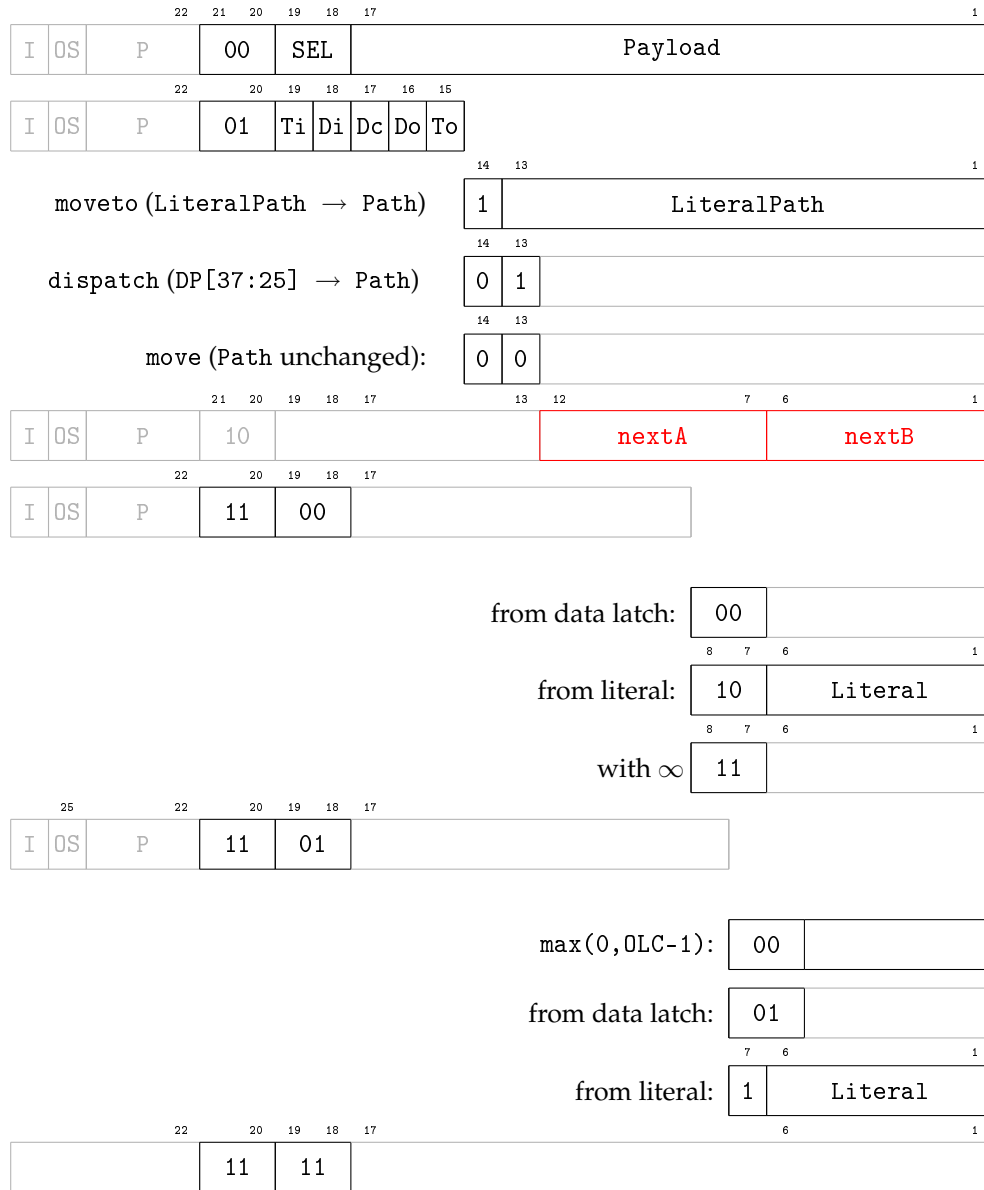


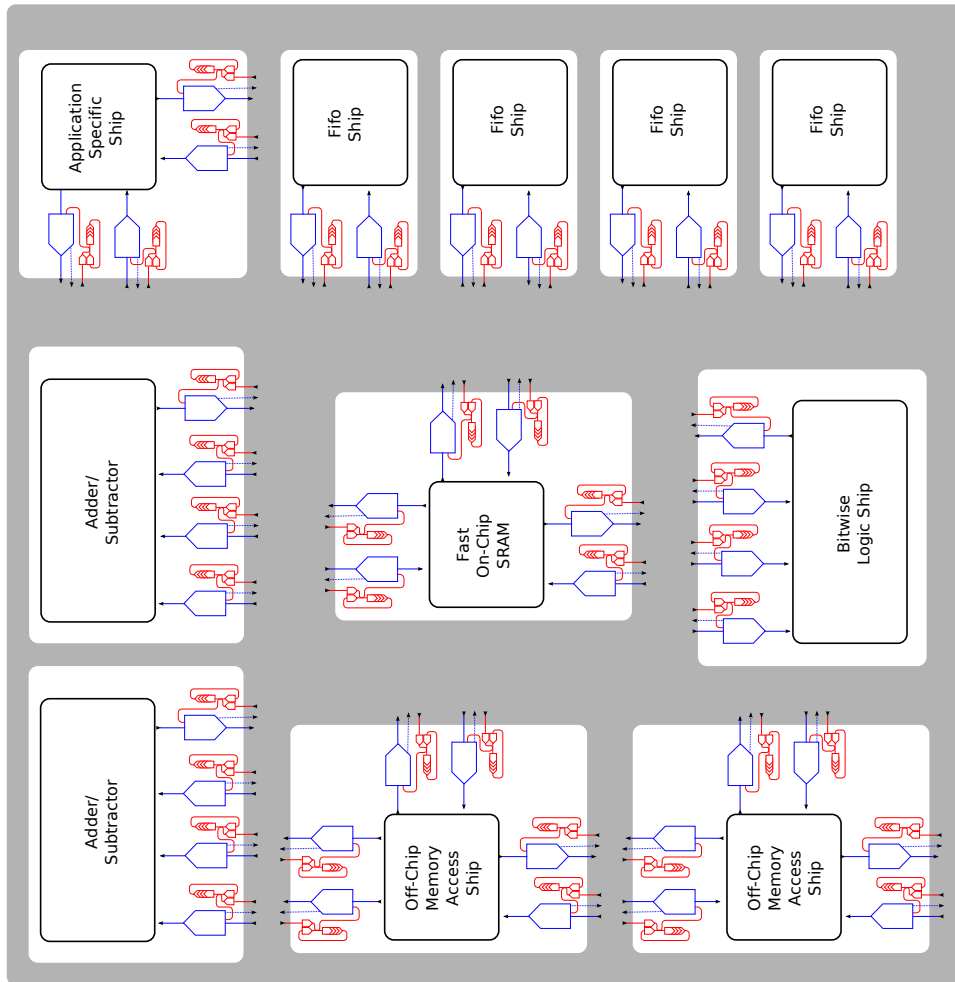
When this instruction executes, the flags are updated according to the nextA and nextB fields; each specifies the new value as the logical OR of zero or more inputs:



Each bit corresponds to one possible input; all inputs whose bits are set are ORed together, and the resulting value is assigned to the flag. Note that if none of the bits are set, the value assigned is zero. Note also that it is possible to produce a 1 by ORing any flag with its complement, and that setFlags can be used to create a nop (no-op) by setting each flag to itself.

Instruction Encoding Map





Output Dock

