

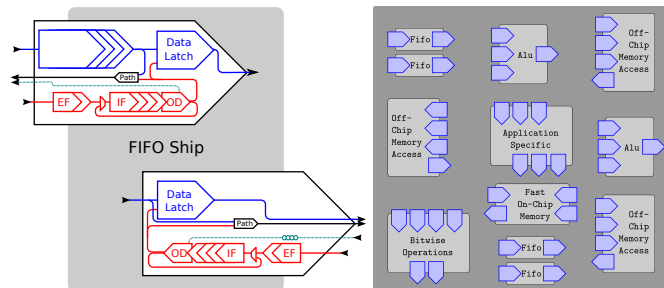
AM33: The Marina Docks

Adam Megacz

August 29, 2009

Abstract

This document describes the Docks on the Marina test chip.



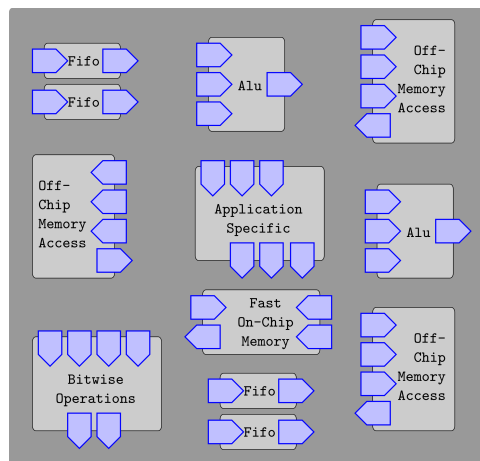
1 Overview of Fleet

A Fleet processor is organized around a *switch fabric*, which is a packet-switched network with reliable in-order delivery. The switch fabric is used to carry data between different functional units, called *ships*. Each ship is connected to the switch fabric by one or more programmable elements known as *docks*.

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 to be delivered 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 instruction packets into the switch fabric with paths that will lead them to the instruction destinations of the docks at which they are to execute.

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. When an instruction executes it may consume this data and may present a data value to the ship or transmit a packet.

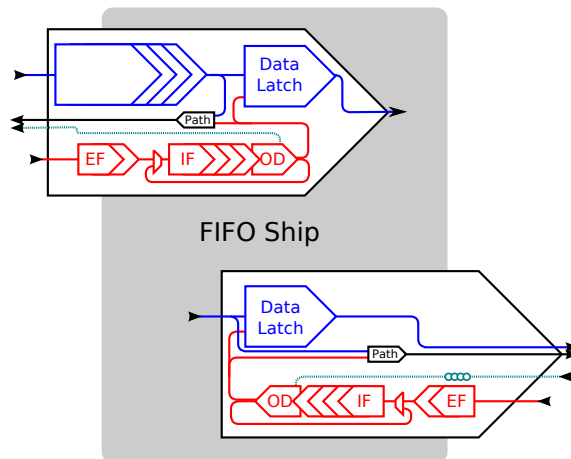
When an instruction sends a packet into the switch fabric, it may specify that the payload of the packet is irrelevant. Such packets are known as *tokens*, and consume less energy than data packets.



Overview of a Fleet processor; dark gray shading represents the switch fabric, ships are shown in light gray, and docks are shown in blue.

2 The Marina Dock

The diagram below represents a conceptual view of the interface between ships and the switch fabric; actual implementation circuitry may differ.



An “input” dock and “output” dock connected to a ship. Solid blue lines carry either tokens or data words, red lines carry either instructions or torpedoes, and dashed lines carry only tokens.

Each dock consists of a *data latch*, which is as wide as a single machine word and a circular *instruction fifo* of instruction-width latches. The values in the instruction fifo control the data latch. The dock also includes a *path latch*, which stores the path along which outgoing packets will be sent.

Note that the instruction fifo in each dock has a destination of its own; this is the *instruction destination* mentioned in the previous section. A token sent to an instruction destination is called a *torpedo*; it does not enter the instruction fifo, but rather is held in a waiting area where it may interrupt certain instructions (see the section on the *move* instruction for further details).

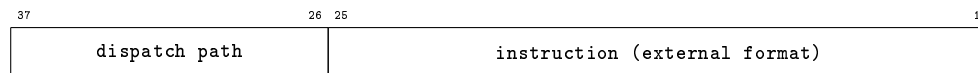
From any source to any dock’s data destination there are two distinct paths which differ by a single bit. This bit is known as the “signal” bit, and the routing of a packet is not affected by it; the signal bit is used to pass control values between docks. Note that paths terminating at an *instruction* destination need not have a signal bit.

3 Instructions

In order to cause an instruction to execute, the programmer must first arrange for 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 destination* of the dock at which it is to execute.

There are two instruction formats, an *external format* described in this section and an *internal format* described in the last section of this memo.

Each instruction is 25 bits long, which makes it possible for an instruction and an 12-bit path to fit in a single word of memory. This path is the path from the *dispatching* dock to the *executing* dock.



Note that the 12 bit `dispatch path` field is not the same width as the 13 bit Immediate path field in the move instruction, which in turn may not be the same width as the actual path latches in the switch fabric. The algorithm for expanding a path to a wider width is specific to the switch fabric implementation, and may vary from Fleet to Fleet. For the Marina experiment, the correct algorithm is to sign-extend the path; the most significant bit of the given path is used to fill the vacant bit of the latch. Because the `dispatch path` field is always used to specify a path which terminates at an instruction destination (never a data destination), and because instruction destinations ignore the sign bit, certain optimizations may be possible.

3.1 Loop Counters

A programmer can perform two types of loops: *inner* loops consisting of only one move instruction and *outer* loops of multiple instructions of any type. Inner loops may be nested within an outer loop, but no other nesting of loops is allowed.

The dock has two loop counters, one for each kind of loop:

- OLC is the Outer Loop Counter
- ILC is the Inner Loop Counter

The OLC applies to all instructions and can hold integers 0..MAX_OLC (63).

The ILC applies only to move instructions and can hold integers 0..MAX_ILC (63) as well as a special value: ∞ . When ILC=0 the next move instruction executes zero times (ie is ignored). When ILC= ∞ the next move instruction executes until interrupted by a torpedo. After every move instruction the ILC is reset to 1 (note that it is reset to 1, *not to 0*).

3.2 Flags

The dock has four flags: A, B, C, and D.

- 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 may be set by the move instruction based on information from the ship or from an inbound packet. See the move instruction for further details.
- The D flag is known as the *done* flag. The D flag is *set* when the OLC is zero immediately after execution of a `set o1c` or `decrement o1c` instruction, or when a torpedo strikes. The D flag is *cleared* when a `set o1c` instruction causes the OLC to be loaded with a nonzero value.

3.3 Predication

All instructions except for `head` and `tail` have a three-bit field marked P, which specifies a *predicate*.



The predicate determines which conditions must be true in order for the instruction to execute; if it is not executed, it is simply *ignored*. The table below shows what conditions must be true in order for an instruction to execute:

Code	Execute if
000:	D=0 and A=0
001:	D=0 and A=1
010:	D=0 and B=0
011:	D=0 and B=1
100:	Unused
101:	D=1
110:	D=0
111:	always

3.4 The Requeue Stage

The requeue stage has two inputs, which will be referred to as the *enqueueing* input and the *recirculating* input. It has a single output which feeds into the instruction fifo.

The requeue stage has two states: UPDATING and CIRCULATING.

3.4.1 The UPDATING State

On initialization, the dock is in the UPDATING state. In this state the requeue stage is performing three tasks:

- it is draining the previous loop's instructions (if any) from the fifo
- it is executing any "one shot" instructions which come between the previous loop's tail and the next loop's head
- it is loading the instructions of the next loop into the fifo.

In the UPDATING state, the requeue stage will accept any instruction other than a tail which arrives at its *enqueueing* input, and pass this instruction to its output. Any instruction other than a head which arrives at the *recirculating* input will be discarded.

Note that when a tail instruction arrives at the *enqueueing* input, it "gets stuck" there. Likewise, when a head instruction arrives at the *recirculating* input, it also "gets stuck". When the requeue stage finds *both* a tail instruction stuck at the *enqueueing* input and a head instruction stuck at the *recirculating* input, the requeue stage discards both the head and tail and transitions to the CIRCULATING state.

3.4.2 The CIRCULATING State

In the CIRCULATING state, the dock repeatedly executes the set of instructions that are in the instruction fifo.

In the CIRCULATING state, the requeue stage will not accept items from its *enqueueing* input. Any item presented at the *recirculating* input will be passed through to the requeue stage's output.

When an abort instruction is executed, the requeue stage transitions back to the UPDATING state. Note that abort instructions include a predicate; an abort instruction whose predicate is not met will not cause this transition.

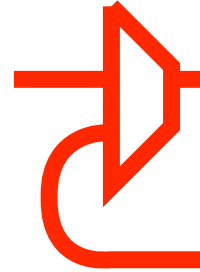
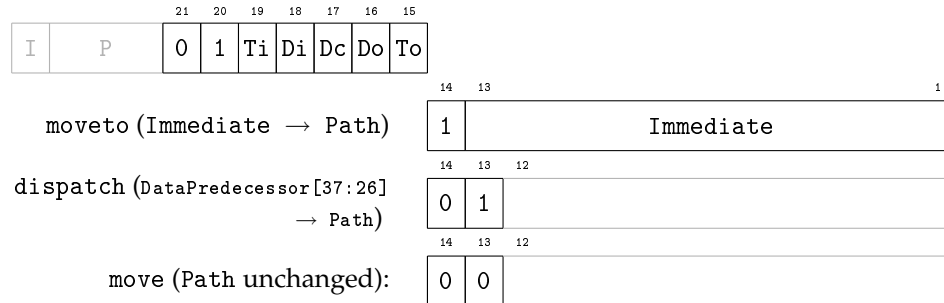


Figure 1: *the requeue stage*

4 Instructions

4.1 move



- 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 I bit stands for Immune, and indicates if the instruction is immune to torpedoes.

Every time the move instruction executes, the C flag is be set:

- If the dock is an *output* and the instruction has the Dc bit set, the C flag is set to a value provided by the ship.
- Otherwise, if Ti=1 at any kind of dock or Di=1 at an input dock, the C flag is set to the signal bit of the incoming packet.
- Otherwise, the signal bit is set to an undefined value.

The `flush` instruction is a variant of `move` which is valid only at input docks. It has the same effect as `deliver`, except that it sets a special “flushing” indicator along with the data being delivered.



When a ship fires, it must examine the “flushing” indicators on the input docks whose fullness was part of the firing condition. If all of the input docks’ flushing indicators are set, the ship must drain all of their data successors and take no action. If some, but not all, of the indicators are set, the ship must drain *only the data successors of the docks whose indicators were not set*, and take no action. If none of the flushing indicators was set, the ship fires normally.

4.3 shift

Each `shift` instruction carries an immediate of 19 bits. When a `shift` instruction is executed, this immediate is copied into the least significant 19 bits of the data latch, and the remaining most significant bits of the data latch are loaded with 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 19 bits at a time.



The Marina implementation has an unarchitected “literal latch” at the on deck (OD) stage, which is loaded with the literal *at the time that the shift instruction comes on deck*. This latch is then copied into the data latch when the instruction executes.

4.4 abort



An `abort` instruction causes a loop to exit; see the section on the Requeue Stage for further details.

4.5 head



A `head` instruction marks the start of a loop; see the section on the Requeue Stage for further details.

4.6 tail



A `tail` instruction marks the end of a loop; see the section on the Requeue Stage for further details.

Errata

The following additional restrictions have been imposed on the dock in the Marina test chip:

Both Docks

1. A Marina dock initializes with the ILC, OLC, and flags in an indeterminate state.
2. The instruction immediately after a move instruction must not be a set flags instruction which utilizes the C-flag (the value of the C-flag is not stable for a brief time after a move).
3. If a move instruction is torpedoable (ie it has the I bit set to 0), it *must* have either the Ti bit or Di bit set (or both). It is not permitted for a torpedoable move to have both bits cleared.

Dock with Ivan's Counter (non-stretch)

1. A torpedoable move instruction must not be followed immediately by a set olc instruction or another torpedoable move.
2. This document specifies that when a torpedoable move instruction executes successfully, the D flag is unchanged. In Marina, when a torpedoable move instruction executes successfully, it causes the D flag to be set if the OLC was zero and causes it to be cleared if the OLC was nonzero. Thus, in the following instruction sequence:

```

head;
[*] set olc=1;
    send token to self:i;
[T] recv token;
[*] send token to self;
[T] recv token;
[*] abort;
tail;

```

Will leave the D flag *set* on Marina, whereas a strict implementation of this document would leave it cleared.

In practice, this distinction rarely matters.

Dock with Kessels Counter ("stretch")

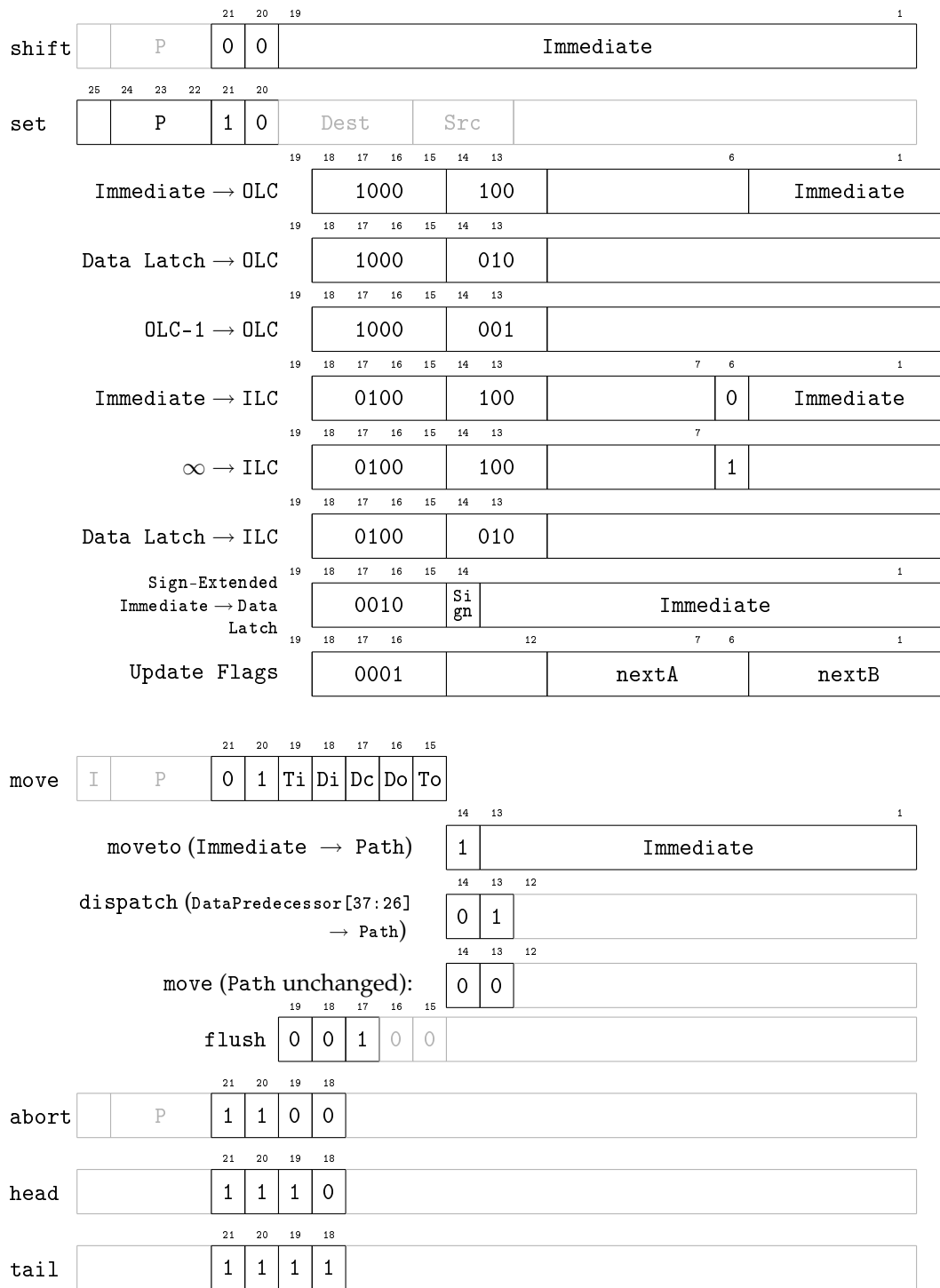
With the Kessels counter, the D-flag *is exactly equal to* the zeroness of the OLC; it cannot be "out of sync" with it.

1. Every "load OLC" instruction must be predicated on the D-flag being *set*. This is a sneaky way of forcing the programmer to "run down" the

counter before loading it, because Kessels' counter does not support "unloading."

2. Every "decrement OLC" instruction must be predicated on the D-flag being *cleared*. This way we never have to check if the counter is already empty before decrementing.
3. The instruction after a torpedoable move must not be predicated on the D-flag being *set* (it may be predicated on the D-flag being *cleared*. This is because, while the move instruction is waiting to execute, the D-flag will be cleared, and the predicate stage believes that it can skip the instruction even though `do[ins]` is still high (I think this is dumb).

External Instruction Encoding Map



Internal Instruction Encoding Map

Marina Instructions in main memory occupy 37 bits. Of this, 11 bits give the path to the dock which is to execute the instruction; thus, only 26 of these bits are interpreted by the dock.

It is easiest to design the OD and EX stages of the dock if the control bits supplied there are mostly one-hot encoded. Moreover, due to layout considerations there is very little cost associated with making the instruction fifo 36 bits wide rather than 26 bits wide.

Due to these two considerations, all 26-bit instructions binary-coded-control instructions are expanded into 36-bit unary-coded-control instructions upon entry to the instruction fifo. This section documents the 36-bit unary-coded-control format.

Predicate Field

The Predicate field, common to many instructions, consists of a six-bit wide, one-hot encoded field. The instruction will be **skipped** (not executed) if **any** condition corresponding to a bit whose value is one is met.



For example, if bits 31 and 34 are set, the instruction will be skipped if either the B flag is cleared or the A flag is set. Equivalently, it will be executed iff the B flag is set and the A flag is cleared.

Set Flags

Each of the FlagA and FlagB fields in the Set Flags instruction gives a truth table; the new value of the flag is the logical OR of the inputs whose bits are set to 1.

