

AM33: The FleetTwo Dock

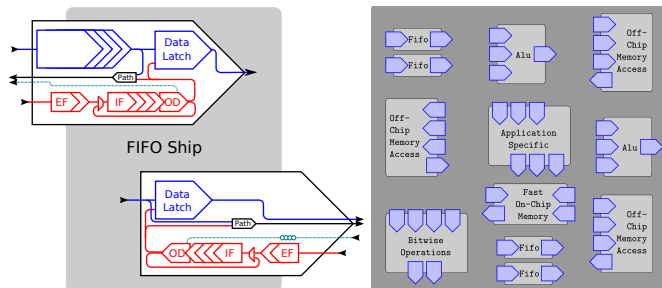
Adam Megacz

January 5, 2009

Abstract

Changes:

- 05-Jan Fixed a one-word typo
- 02-Jan Added head instruction
- Lengthened external encoding of tail instruction by one bit
- Added abort instruction
- Removed OS field from instructions
- Renamed the Z-flag (olc Zero) to the D-flag (loop Done)
- 19-Dec Updated diagram in section 3 to put dispatch path near MSB
- Changed DP[37:25] to DP[37:27]
- Added note on page 4 regarding previous
- 14-Nov Roll back "Distinguish Z-flag from OLC=0"
- Clarify what "X-Extended" means
- Change C-bit source selector from Di to Dc
- 07-Nov Distinguish Z-flag from OLC=0
- Add flush instruction
- Change I bit from "Interruptable" to "Immune"
- 20-Sep Update hatch description to match IES50
- 28-Aug Note that decision to requeue is based on value of OLC *before* execution
- Note that decision to open the hatch is based on value of OS bit



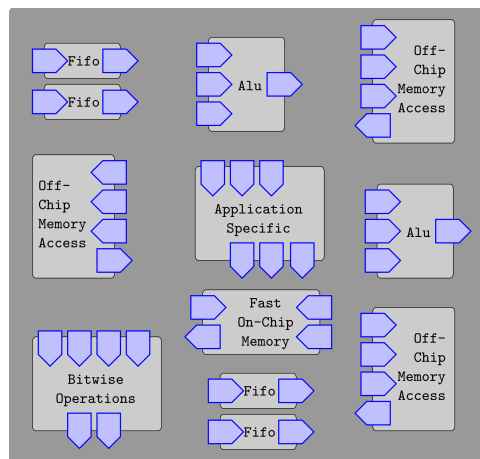
1 Overview of Fleet

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 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.

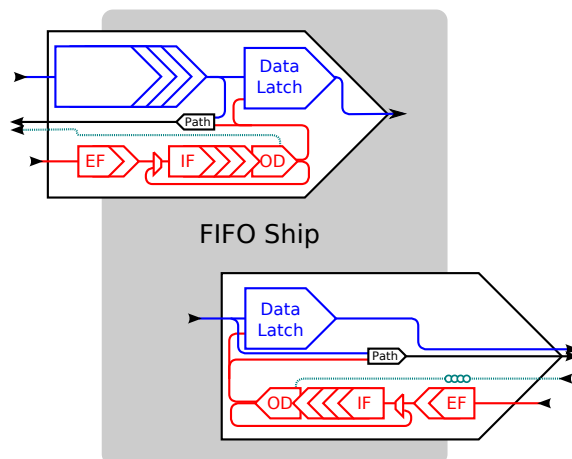
When an instruction sends a packet into the switch fabric, it may specify that the payload of the packet is irrelevant. Such packets are known as *tokens*, and consume less energy than data packets.



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 FleetTwo 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.

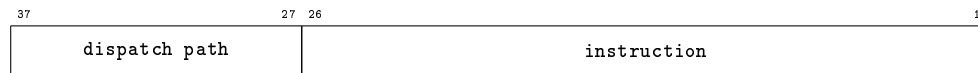
Note that the instruction fifo in each dock has a destination of its own; this is the *instruction destination* mentioned in the previous section. A token sent to an instruction destination is called a *torpedo*; it does not enter the instruction fifo, but rather is held in a waiting area where it may 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.

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 that the 11 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 is not specified by this document.¹ In particular, 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 signal 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 $0..MAX_OLC$.

The ILC applies only to move instructions and can hold integers $0..MAX_ILC$ as well as a special value: ∞ . When $ILC=0$ the next move instruction executes zero times (ie is ignored). When $ILC=\infty$ 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*).

¹for the Marina experiment, the correct algorithm is to sign-extend the path; the most significant bit of the given path is used to fill all vacant bits of the latch

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 olc` or `decrement olc` instruction, or when a torpedo strikes. The D flag is *cleared* when a `set olc` 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.

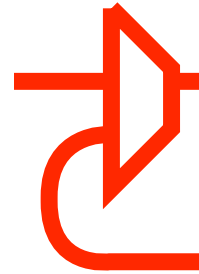


Figure 1: *the requeue stage*

4.2 set

The `set` command is used to set or decrement the inner loop counter, outer loop counter, and data latch.

		26	25	24	23	22	21	20															
			P	1	0	Dest				Src													
Immediate → OLC		19	18	17	16	15	14	13							6	1							
		1000				100										Immediate							
Data Latch → OLC		19	18	17	16	15	14	13															
		1000				010																	
OLC-1 → OLC		19	18	17	16	15	14	13															
		1000				001																	
Immediate → ILC		19	18	17	16	15	14	13				7	6	1									
		0100				100							0	Immediate									
∞ → ILC		19	18	17	16	15	14	13				7											
		0100				100							1										
Data Latch → ILC		19	18	17	16	15	14	13															
		0100				010																	
Sign-Extended Immediate → Data Latch		19	18	17	16	15	14											1					
		0010				Si	gn	Immediate															
Update Flags		19	18	17	16	12						7	6	1									
		0001										nextA		nextB									

The FleetTwo implementation is likely to have an unarchitected “literal latch” at the on deck (OD) stage, which is loaded with the possibly-extended literal *at the time that the set instruction comes on deck*. This latch is then copied into the data latch when a `set Data Latch` instruction executes.

The Sign-Extended Immediate instruction copies the Immediate field into the least significant bits of the data latch. All other bits of the data latch are filled with a copy of the bit marked “Sign.”

Each of the `nextA` and `nextB` fields has the following structure, and indicates which old flag values should be logically ORed together to produce the new flag value:

6	5	4	3	2	1
A	\bar{A}	B	\bar{B}	C	\bar{C}

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 `set Flags` can be used to create a nop (no-op) by setting each flag to itself.

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 FleetTwo implementation is likely to have 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.

Instruction Encoding Map

