A Simpler View of Fleet

October 24, 2007

Abstract

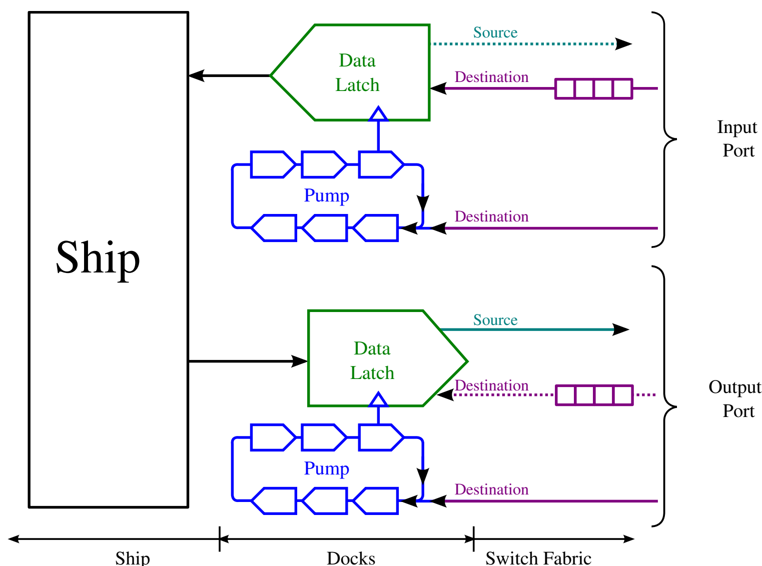
This document attempts to describe Fleet and its semantics as briefly as possible, with no assumption of prior technical knowledge about the architecture.

1 Introduction

This document is intended to be a self-contained description of Fleet's semantics.

A Fleet processor consists of a number of functional units, which are called *ships*. Example ships include adders, multipliers, storage fifos, on-chip "scratch-pad" memories, interfaces to off-chip memory, and comparison units.

These ships are connected to each other by a *switch fabric*. The points at which the ships join to the switch fabric are called *docks*, which are the focus of most of the programmer's attention.



Each dock can be either an *input* dock (used to accept data into the ship) or an *output* dock (used to send data from the ship out to another ship via the switch fabric). As an example, an adder ship might have three docks: two input docks for operands and one output dock for a result.

The main component of each dock is a data latch (or register), capable of holding one machine word[1]. An apparatus called the *pump* is responsible for controlling the data latch: the pump decides when the data latch should capture a new value from its input and when it should present its current value to its output.

The programmer controls the pump by issuing instructions through the switch fabric. These instructions are sent over the same network used for data, and each instruction word is the same size as a data word.

[1] the chip Sun is currently fabricating uses 37-bit words

Each dock has three connections to the switch fabric. One of these is for delivering instructions to the pump, one is for inbound data, and one is for outbound data. In the case of input docks, the outbound data connection can carry only *tokens* – zero-bit data items used for sequencing operations. The same is true of the inbound data connection at an output dock.

Tokens are semantically identical[2] to words whose value is undefined – any word can act as a token, and a token can act as a word – although the value of its bits will be undefined.

Each connection to the switch fabric is either a *source* or a *destination*. A *path* is a sequence of bits specifying a route through the switch fabric, from a specific source to a specific destination. Items sent along a given path will arrive in the order in which they were sent.

2 Fleet Programs

Actual Fleet machine code is specified in terms of extremely primitive *micro-operations* executed directly by the pumps[3]. These micro-operations were chosen to make the hardware as simple as possible while retaining as much flexibility as possible – but with no attention paid to ease of programmability or readability of programs. For purposes of compilation and program analysis, it is convenient to define a somewhat higher-level assembly language which has an obvious mapping to micro-operations.

3 Instructions

3.1 Atomic Instructions

Atomic instructions are the most primitive component of a Fleet program. The available atomic instructions are:

`AtomicInstruction ::=`	`literal(i)`	load literal (immediate) into data latch
	`?ship`	wait for value from ship, store in data latch
	`?fabric`	wait for value from fabric, store in data latch
	`_ship`	wait for value from ship and drop it
	`_fabric`	wait for value from fabric and drop it
	`!ship`	offer value to ship, wait for it to accept
	`!fabric(p)`	offer value to fabric, wait for it to accept
	`!token(p)`	offer token to fabric, wait for it to accept

[2]there is a difference in power consumption, however
[3]further details can be found here

At an input dock, the `?fabric` instruction waits for a word to arrive from the switch fabric and overwrites the contents of the data latch with the new word. The `!ship` instruction offers the value in the data latch to the ship, and waits for the ship to accept it. The `_fabric` instruction waits for a word to arrive from the switch fabric and discards it.

At an output dock, the `?ship` instruction waits for the ship to offer a word, and overwrites the contents of the data latch with the new word. The `!fabric(p)` instruction offers the value in the data latch to the switch fabric, to be sent along path `p`. The `_ship` instruction waits for the ship to offer a value, but discards it.

In both input docks and output docks, the `_fabric` instruction waits for a value to arrive from the switch fabric, but ignores the actual value. In both input and output docks, the `!token(p)` instruction offers a token to the switch fabric, to be sent along path `p`.

Both the `!token(p)` and `!fabric(p)` instructions take an optional path argument. If this path is specified, then the word or token being written to the fabric is sent to the destination specified by that path. If no path is specified, then *the most significant bits of the data latch* are used as a path[4]

The `literal(i)` instruction carries a literal[5] and overwrites the value in the data latch with that literal.

3.2 Complex Instructions

A sequence of simple instructions may be repeated a small number of times[6], or it may be repeated forever.

```
Instruction ::=  AtomicInstruction
                 repeat_i{AtomicInstruction*}
                 forever{AtomicInstruction*}
                 interrupt(i)
```

If the pump is executing a `forever{}` loop, the only way to terminate that loop is by issuing an `interrupt(i)` instruction. The parameter given to the interrupt instruction *must exactly match* the number of instructions in the loop to be interrupted[7] – in effect, the programmer must know what loop is being interrupted in order to interrupt it correctly.

The `interrupt()` instruction must only be used to interrupt `forever{}` instructions – never `repeat`$_i$`{}` instructions. It is an error to issue an `interrupt()`

[4]This feature is extremely powerful, but also very difficult to reason about. It would be okay to exclude "pathless sends" from any sort of analysis; all interesting programs can be written without it, although sometimes at a very heavy performance penalty.

[5]currently up to 17 bits, sign-extended

[6]currently limited to 2^6

[7]there is a small chance that this requirement may be lifted, but it is unlikely

instruction to a pump which is not currently executing a `forever{}` instruction.

3.3 Execution

When an instruction arrives at a pump, it is simply appended to the program currently executing at the pump.[8] If a pump has executed all of its instructions, it simply waits for more instructions to arrive.

3.4 Restrictions and Requirements

It is an error for any instruction to arrive at a pump while it is executing a $\texttt{repeat}_i\texttt{\{\}}$, and it is an error for any instruction other than an `interrupt()` to arrive at a pump while the pump is executing a `forever{}`.

There are buffers between the switch fabric and each data latch destination. These buffers are of finite size[9], and it is an error to ever exceed this capacity.

The circular fifo which holds a pump's currently-executing program is of finite size[10]. It is an error to send an instruction to a pump when the pump's instruction fifo is full. If instruction sequences longer than this limit are required, there is a fairly simple code transformation that will cause them to be dispatched in chunks; however, these long instruction sequences must be statically identifiable within the code.[11]

[8] there is a rather complex micro-instruction mechanism for getting `repeat{}`'s – which are actually many words long – to be appended atomically. Fortunately this can be ignored at the program analysis level.

[9] the exact size is currently being debated

[10] currently 8 instructions

[11] it may be worth introducing a separate assembler-level construct for these "batched instruction sequences"