1st Class Instructions for FLEET

Greg Gibeling
Wednesday October 4th, 2006
GDG01 – 1st Class Instructions for FLEET

1.0 Introduction

In the original specifications for FLEET discussed, debated and implemented throughout the 2005/2006 school year at UC Berkeley, FLEET retained an element of strong data typing at the architectural level. This feature was in strong contrast to most existing ISAs which make no distinction between various data types.

With the advent of IES30[] data typing of most operands has been abandoned, in favor of the more common method of allowing the various operators to determine and interpret the types of their data. This update works hand in hand with the removal of standing moves and out of band values.

However, as FLEET drifts slowly towards a more common architecture, with respect to data types, it has inherited one of the primary flaws of these architectures, which has been arbitrarily imposed by tradition: namely that instructions and data should be separate. In a single-cycle, pipelined or even out of order processor design, this distinction may be viewed as beneficial as it removes some circularity and dependence between the instruction fetch and decode logic, and the execution logic. Yet it does require, or at least imply, that instruction and data cache should be separate, requiring duplicate storage structures.

While the duplicate caches are a small price to pay given the availability of silicon area, and the separation of fetch and decode from execution is beneficial, FLEET does not share this optimistic outlook. First, because FLEET is intended to be highly concurrent, hopefully far more than the standard 4-way superscalar designs which are currently practical, and second because FLEET is centered around the cost of moving data through a switch fabric, rather than ignoring the multiplexers that would normally handle the same work, there is actually a substantial positive cost to separation of code and data.

The costs include:

- Duplicate switch fabrics for routing instructions and data. (Section 2.1 The Switch Fabrics)
- Logic to distinguish and decode more data types. (Section 2.2 Literals)
- Inflexibility of the fetch logic. (Section 2.3 Sequencing & Fetching)
- Inability to sequence instructions without false data dependencies. (2.4 Passive Reads & Tokens)

2.0 Existing Design

This section outlines the current design of FLEET with respect to a number of the less-well understood implications of our current thinking. It attempts to make fewer assumptions that are generally made in class, and to point out flaws in current thinking without regard to solutions. Solutions are presented in section 3.0 1st Class Instructions.

2.1 The Switch Fabrics

First of all, the separation of instructions and data has thus far led to the assumed existence of duplicate switch fabrics. The first, the 'I' fabric carries instructions from a data structure (a mapping of SHIP source ports to lists of instructions) known as the instruction pool, to the SHIP sources. Because of the ordering guarantee outlined in IES30[], this fabric must perform either oblivious routing, or ensure that only one instruction per SHIP source is active at any one time. Once an instruction has arrived at a source it is buffered, perhaps only briefly waiting for the data which it is intended to move. This data is then routed to its destination over what has been assumed to be a standard packet switched second, or 'D' switch fabric.

In general this means that both an instruction on its way to a source, and data on its way to a destination are assumed to be represented as standard <destination, payload> packets in an obliviously routed packed switched network, and that there are two such identical networks.

This scheme has the drawback that each switch fabric is likely to be costly and perhaps complicated, not to mention that silicon layout will be complicated by the overlap of the two.

2.2 Literals

Literals thus far in the design of FLEET have remained a source of contention and uncertainty. Various schemes for their introduction to a running processor have been discussed, ranging from special "literal bags" which are collections of literals loaded form memory with each code bag but which could be shared by various code bags, all the way to a special form of the move instruction.

In the end, both of these solutions have significant drawbacks. The use of literal bags entails the fetching of literals possibly from another memory location, resulting in likely instruction cache problems, not to mention coherence and consistency overhead in the event of literal sharing. The use of special "literal move" instructions however introduces a kind of instruction which is no longer particularly well defined in light of section 2.1 The Switch Fabrics, as they must presumably start off as instructions in the 'I' fabric, but must then be moved over to the 'D' fabric for delivery.

2.3 Sequencing & Fetching

One of the most pervasive problems encountered throughout the effort to write useful FLEET programs during spring 2006, was the lack of sequencing guarantees. This led to various schemes were the loading of code bags was triggered by the completion of an operation, often through explicit, and otherwise false, data dependencies.

IES30[] resolved many of these issues by adding a source sequence guarantee; that all instructions within a single code bag which take data from a single source will be executed in the order in which they appear in memory. While this scheme deserves significant credit for reducing the sequencing problems, it does not eliminate them.

A normal processor provides roughly two guarantees, one for source sequencing as above, and a similar one for destination sequencing. Without the destination sequencing guarantee there must still be times when a program will require explicit token based sequencing, and as a result becomes very convoluted and slow with many trips through the

'D' fabric. The assembly of records [IES31] comes to mind as a particular problem.

Furthermore, the source sequence guarantee may be too strong for many programs, or at least many code bags. In general there are likely to be a significant number of situations where no such guarantee is needed, or perhaps where no such guarantee is useful. A SHIP which outputs the same data repeatedly, and a SHIP which is part of an occupancy one, self-sequenced loop are perhaps the most obvious two.

2.4 Passive Reads & Tokens

In IES31[] there are a number of paragraphs outlining the design of base literal generation SHIPs, ones which generate 'true', 'false' or '0', not to mention timestamps or random numbers. For all of these SHIPs two designs are considered which are generally classified in the I/O peripheral community as active and passive read. Active read designs are those which imply some action upon read, i.e. those designs in IES31[] which can produce an output value at any time, a random number generator which generates a new value each time it is read, or a stack SHIP which pops any value read from its output. Passive read designs perform roughly the same function but separate the act of, for example reading the current random number, from the act of generating a new one, which must now be controlled by a token.

In general I/O peripheral designs make careful use of both schemes, often using active read where buffering is implied such as a stack SHIP. In FLEET as currently described however, tokens and the exclusive use of passive reads are practically dictated by the lack of destination sequence guarantee and the loose concurrency provided by code bags.

This problem is alleviated by the source sequence guarantee in that a stack SHIP could peek or pop it's top depending on whether a move or copy instruction was received, but the situation for a random number generator, or worse, a timestamp generator is completely untenable without a positive, sequence-able trigger input such as a token.

3.0 1st Class Instructions

In the previous sections, a number of problems were presented, and a couple of assumptions questioned, or at least brought to light as questionable. This section presents a single solution which seems, at this time, to resolve a number of design, programming and implementation problems.

3.1 A Definition

IES30 purports to eliminate data types from the FLEET specification, and yet there remain two: data and instructions. As written, and as assumed in more common ISAs, instructions are a sort of second class type of data. This assumption has far-reaching implications which in general are the cause of many of the problems outlined in section 2.0 Existing Design. Further problems include the separation of I and D cache, not to mention the prohibition against self-modifying code, which was the basis of a lively discussion in Fall of 2005.

By truly removing data type separation, and viewing all data within a FLEET as nothing more than bits interpreted by whatever SHIP or switch fabric they are handed too we can greatly simplify the design,

programming and implementation of FLEET.

A brief description of the life cycle of these new instructions will guide the following discussion. First an instruction must be read from memory, an operation which in this model can be safely left to a standard memory access SHIP rather than a specialized fetch SHIP. Perhaps the only specialized operation is that the data read from memory should be inject directly into the switch fabric. Because the instructions are represented in memory and in the switch fabric in identical formats this is a trivial operation. These instructions or packets, will include a destination header and a payload. Presumably most of them will be identical to the old move instructions, whereby the destination of the instruction will be the source of the move (or a special port on the same SHIP) and the payload of the instruction will be the destination of the move, which the source should attach to whatever data is waiting.

3.2 The Switch Fabric

This is by far the most straightforward step: by merging our ideas of data and instructions, we can entirely eliminate the use of separate switch fabrics. While this does increase the size of remaining switch fabric, which may very well grow in super-linearly with respect to the number of SHIP ports it must service, the resulting simplicity should more than repay this cost.

Furthermore while it is currently assumed that the switch fabric is a simple merge and fan-out design, this may not always be the case. By reducing a FLEET to a single, seemingly uniform, switch fabric, and leaving its implementation for later we can greatly increase the odds of an efficient implementation.

In this case, the switch fabric is reduced a single packet-switched network, which can carry variable length payloads; it need make no distinction between instructions and data. Some may be made in the case of e.g. the Bit Bucket, or other such optimization cases, but in the abstract this is a far simpler design. Furthermore, with the advent of some tricks described in the below sections, it is likely that the oblivious routing requirement will be rendered unnecessary and cumbersome, further simplifying matters. As a final point a packet-switched network is a relatively common entity which it is easy to build and experiment on, whereas the custom switch fabrics described in section 2.1 The Switch Fabrics, are likely to be highly specialized.

3.3 Data Typing, Literals & Fetching

By reducing all instructions to a simple data packet with a destination address and a payload, it is easy to implement literals as a packet whose payload is a literal, rather than the destination address of a move, as described in section 3.1 A Definition.

In addition there is no longer any need to separate literals as a special case. This obviates all debate over the use of literal bags, special literals SHIPs or event the need for a special form of the move instruction.

Furthermore, the use of a standard memory read SHIP rather than a specialized fetch ship will allow the programmer to determine the timing and method of the fetch ordering. This implies that source sequencing is no longer a meaningful property, but rather one which a programmer can choose. It

is possible that the first code bag loaded upon reset could setup the functional equivalent of the fetch SHIP as outlined in IES30. However it should be equally possible to create a far less constraint fetch system in the event of highly concurrent code.

Overall the removal of instructions as a distinct data type simplifies a number of the design points of FLEET as well as ending some of the more contentious debates among the designers, implementers and programmers.

3.4 Sequencing, Passive Reads & Tokens

The primary reason for the existence of tokens and the use of passive reads, which often imply the existence of a token to perform some related action, is to control the sequencing of operations.

The primary drawback of this approach is that it rests on the introduction of data-dependencies in the form of tokens to stand in for what in a common ISA would be a control-dependence. Not only does this scheme obscure the meaning of tokens somewhat, but it also increases the number of move instructions required to execute a fixed amount of work, as the tokens must traverse the switch fabric, adding to the latency of the operation.

With the advent of first class instructions, instructions become sequence-able. That is to say, there can exist SHIPs which control the timing of the injection of instructions into the switch fabric based on the completion of operations. With this capability in hand, we can now use the arrival of instruction packets, that is packets whose payload is a destination address to be added to some data, to perform the same sequencing function that tokens previously provided. Now, all SHIPs which output tokens upon completion should instead provide a packet (instruction) pass-through interface, whereby an instruction accepted at a special destination would be passed unmolested out of a source upon completion of an operation.

Furthermore, because instruction packets now represent the control dependencies in a very well defined form, they can be used in active reads, whereby the delivery of an packet will tell, e.g. a timestamp generator both to read the current time, and where to send it.

4.0 Conclusion

By removing the distinction between data and instructions in the switch fabric of FLEET, we allow instructions to be a 1st class datatype. In doing this we simplify the overall architecture by removing a redundant switch fabric, and the need for a special fetch SHIP. This also reduces the cost related to the source sequence guarantee, by allowing the user to selectively invoke it. In addition the issues which have perennially plagued the literals and token sequencing mechanisms become simplified in a unified way, which has the added benefit of completely obviating the need for SHIPs which generate a fixed literal upon the receipt of a token.

Overall first class instructions simplify a wide range of design, programming and implementation problems by relying on the well understood mechanism of packet switched networks to control both sequencing, communication and control.