

UC Berkeley computer science

Subject: Introducing sequential behavior in Fleet
Date: September 2005
Authors: Igor Benko
UCIB#: 2005-ib05

References:

- [1] UCIBIS 2005-is02: Fleet – a one instruction computer, by Ivan Sutherland, August 2005
- [2] SML 2005-0253: Deadlocks in Fleet – problem characterization, by Benko, Furber and Clark, March 2005
- [3] SML 2000-0768: FLEETZero: An Asynchronous Switching Experiment, by Jones, Coates, Lexau, Fairbanks, and Sutherland

FOREWORD

This memo started as a presentation of one idea for introducing sequentiality into an instruction issue mechanism for Fleet. The idea was the concept of a logical clock. Even though the idea was far from polished, I decided it was valuable to record the idea and possibly sharpen it during the writing process. As I was writing the memo, I was following my train of thought, got stuck a number of times, and ended up writing a memo that describes a whole collection of unpolished ideas. I still believe the exercise was worth undertaking, not only for the sheer benefits of what is recorded in this memo but for the benefit of at least five more memos that got started while I was writing this memo.

BASE MODEL: A POOL OF CONCURRENT INSTRUCTIONS

Figure 1 shows an abstract view of Fleet processor architecture. Note that the core elements of Fleet remain in the picture: Ships are connected via a switch fabric, where each ship serves as both a source of data and a destination for data. The switch fabric enables data transfer between ships. That is, each move instruction that the instruction issue unit releases into the switch fabric transfers data between a source ship and a destination ship. Note that execution of an instruction is limited to the data transfer itself. Any data transformation that may happen inside a ship is a part of operation of the ship. As a result of such an operation, a new data item may be available at the ship's source port. One can say that data transformation takes place as a side effect of data movement.

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley. The ideas contained herein are freely available for any academic purpose.

Ivan Sutherland proposed that Fleet architecture have a pool of concurrent instructions. The instruction issue unit issues these instructions in the switch fabric in a concurrent manner. This means that some of the instructions may still reside in the pool and some may already be dispatched into the switch fabric. The programmer has no control over the order of the timing of instruction dispatch. When the instruction fetch unit adds a new instruction into the pool, the new instruction becomes concurrent with all of the instructions that are already present in the pool and may even execute ahead of some of the instructions that have been added to the pool at an earlier time.

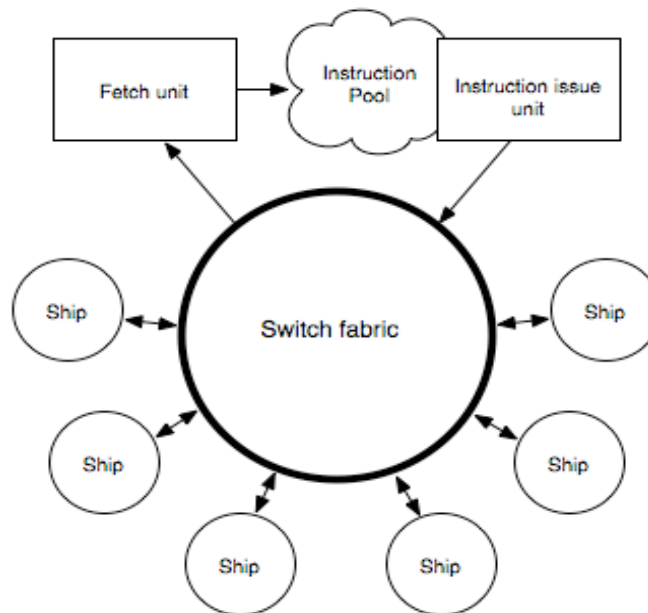


Figure 1: An abstract view of Fleet architecture

Such a pool of concurrent instructions allows for a very large degree of parallelism. A number of move instructions may execute in parallel, and along the way, a number of ships may be performing their operations, potentially transforming data. At the same time, new instructions may join the instruction pool and start their execution.

Let me remark that the introduction of a pool of concurrent instructions turns Fleet into a variable issue computer where the user is in charge of dynamically creating bags of instructions that are to be issued together.

Note that we have not introduced any mechanisms that would introduce a sequence in the otherwise parallel behavior of Fleet. The lack of a sequencing mechanism is an important issue that we must address, because the ability to ensure a sequence is critical when we are dealing with operations that depend on each other's results. Moreover, unless we have a major breakthrough in the area of reasoning about

concurrent programs, we will need a sequencing mechanism to enable writing Fleet programs and reasoning about their properties.

PASSING INSTRUCTION BAGS TO THE FETCH UNIT

Ivan Sutherland proposed a mechanism for introducing sequencing that requires that ships decide when instructions should be added to the instruction pool. More precisely, when a ship has, internally, completed its operation, it passes an instruction bag descriptor to the fetch unit. The fetch unit fetches the instructions described in the bag and adds them to the instruction pool. We use the term *code bag*, because there is no implied order inside the bag, and multiple instances of the same instruction may appear in a bag. How does a ship know what instructions should execute next? A ship may, in addition to the data that needs to be processed, receive one or more code bag descriptors, which it would later pass to the fetch unit.

To illustrate this concept, let us hypothesize a “compare-jump” ship illustrated in Figure 2. The ship may receive two numerical inputs, X and Y, and two code bag descriptors, IDX and IDY. The ship makes the smaller of the two numerical values available on ship’s output S. Moreover, if $(X < Y)$, then code bag descriptor IDX is moved to ship’s output IDS, otherwise code bag descriptor IDY is moved to ship’s output IDS. That is, the ship selects the smaller of the two numerical inputs and passes along the corresponding code bag descriptor.

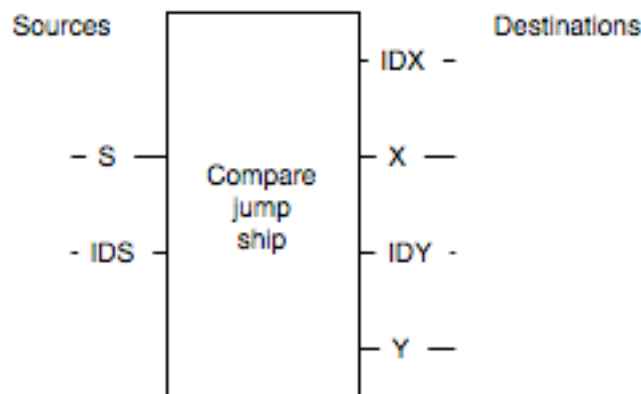


Figure 2: Compare-jump ship

The result of the comparison provided in output S may have a role in subsequent computations. More importantly, moving code bag IDS to the fetch unit causes fetch and execution of the chosen code bag.

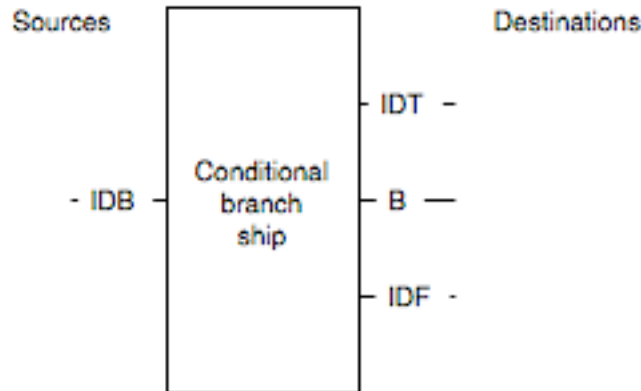


Figure 3: Conditional-branch ship

One can, of course, imagine many variations on such a ship. For example, we could have a conditional-branch ship that would, on the destination side, take one Boolean input B and two code bag descriptors IDT and IDF as shown in Figure 3.

Depending on the value of B, such a ship moves either IDT or IDF to its output IDB. Namely, if B is true, then IDT makes its way to the ship's output, otherwise IDF moves to the output. After IDB moves to the instruction fetch ship, the corresponding instructions are loaded into the instruction pool and are eventually executed concurrently with other instructions in the pool. Such a ship implements a conditional branch instruction: If a flag is true, then one block of code is executed, otherwise execution continues in another code bag. Note the explicit nature of passing of the Boolean flag into the conditional-branch ship. In common processors of today, conditional jump instructions use such a flag in an implicit manner.

PASSING CODE BAG DESCRIPTORS INTRODUCES SEQUENTIALITY

Note that passing a code bag descriptor between ships introduces sequentiality. For example, imagine a data transforming ship with a simple interface shown in Figure 4: two inputs, one for data and one for code bag descriptor, and two outputs, one for data and one for the code bag descriptor. The ship will not make the data available on the output until it receives both the data and the code bag descriptor on the input side.

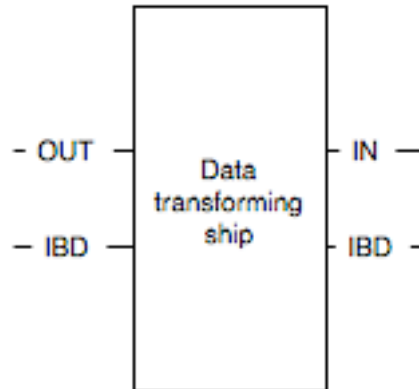


Figure 4: Data transforming ship

One can imagine a sequence of ships that perform a sequence of operations on a data item, transforming it along the way. Only when the whole sequence of operations is preformed will a new block of instructions enter Fleet. In such a case, we can connect a series of ships such as the one from Figure 4 in a virtual pipeline by issuing a set of move instructions that move the data along the pipeline from the output of one ship to the input of the subsequent ship in the pipeline. Note that, provided that these ships are not being used for other operations, such a set of instructions can be issued all at once. The instructions will however, execute in the sequence defined by the pipeline. Note also that along with the data-move instructions, we must issue the set of accompanying instructions that would take care of moving the appropriate code bag descriptor through the pipeline. We can see that such a setup provides us with a mechanism to control when a new code bag descriptor is passed into a fetch unit, and, consequently, when the corresponding instructions are issued into the Fleet switch fabric. We, of course, assume that, for performance reasons, the fetch unit may have the capability to prefetch code bag descriptors, either as a consequence of an explicit hint or due to a built-in prediction mechanism.

We can introduce a variety of modifications to the setup described above. For example, there may be no need to waste energy on lugging a code bag descriptor all the way along the pipeline. It is sufficient that the last ship in the pipeline has a code bag descriptor available when it has completed its operation. We should be careful, however, in proposing such optimizations. Note that the setup above assumed that all of the ships have the same interface expecting both data items and code bag descriptors. If only the final ship in the pipeline is to provide a code bag descriptor, then we need to change interfaces for the rest of the ships by removing the input and output for the code bag descriptor. Such a change, again, comes with an additional complication: now only certain ships can serve as the final stage of a virtual pipeline. Possibly a solution would be to introduce a ship whose sole purpose would be to accept a data item and a code bag descriptor and, once both have arrived, pass them on. Such ships would serve as synchronizers, helping to control issue of new instructions.

Another modification we may consider is that the code bag descriptor and the data item do not need to arrive from the same ship. One could imagine a situation where the data item would arrive from one ship and the code bag descriptor would arrive from another ship. The two separate arrivals would indicate that the two preceding operations have taken place. Possibly, the code bag descriptor itself is a result of a calculation that took place in the originating ship.

Let us remark that in the arrangement described in this section, a code bag descriptor really serves as a token. Only after the code bag descriptor has made its way to the instruction issue unit does the actual contents of the code bag descriptor matter. Therefore, one can consider an optimized scheme where ships pass tokens to each other. Only specific ships would send, upon receipt of a token, a code bag descriptor to the instruction fetch unit.

STANDING INSTRUCTIONS

Ideas described in the previous section introduce the notion of tokens which enable certain ships to send code bag descriptors to the instruction fetch unit. Note that one still needs to issue a move instruction that would transport the code bag descriptor to the fetch unit.

When we deal with repetitive computation such as in signal processing, we expect that repetitiveness in computation may be reflected in the need to issue move instruction in a repetitive manner. Such repetitive process of issuing the same sequences of move instructions may be wasteful. In a more efficient arrangement, the communication paths would be set in advance and the same pattern of moves would take place until an event that indicates termination of such a pattern.

In [1], Ivan Sutherland explores the notion of standing instructions. A move instruction may include a count that indicates how many times in sequence would a source ship send data to the same destination. Thus, by disseminating a set of move instructions with counts, one can program a Fleet processor to repeatedly perform the same set of operations. Ivan proposes two termination mechanisms. In the first mechanism a ship is ready to receive a new instruction when the count associated with the current standing order decreases to zero. The second mechanism makes it possible to "clear" the current standing instruction by sending a special datum to the ship, for example a not-a-number. Details and precise usage of the two mechanisms remain to be ironed out.

The notion of a standing instruction may offer an opportunity to shed new light to the concept of reprogrammable logic. That is, in comparison to any other reprogrammable mechanism I have seen, the concept of standing instruction offer the ability to

reprogram a computation engine at a much higher level, on the fly, and quite possibly with much less complexity.

One challenge in utilizing standing instructions is that one needs to be careful that a fast ship does not plug up the switch fabric by sending too many data items to the destination specified in the standing instruction. If the destination ship cannot consume the data items at the sending rate, the data items will remain in the switch fabric. Depending on the specific architecture of the switch fabric, data items that are stopped in transit may potentially cause Fleet to deadlock. In [2] we discuss possibilities of deadlock in horn-and-funnel style switch fabric akin the one that was used in the Fleet implementation described in [3].

A solution to prevent plug-up of the switch fabric to ensure that the source ship sends data no faster than the destination ship can process it. Ivan Sutherland, in [1], proposes a mechanism that allows us to maintain such alignment between the two ships. The core idea is to use the acknowledge mechanism that we are familiar with from asynchronous circuits. More precisely, when the destination ship is ready to receive more data, it sends an acknowledgment token back to the source ship. Upon receiving the acknowledgment token, the source ship can transmit the subsequent data item towards the destination ship. Ivan proposes that we use tokens to send acknowledgments from a destination to the source. Tokens could travel via the usual Fleet switch fabric, or we could extend the switch fabric by building a separate switch fabric that is responsible for the transport of tokens.

To offset the delay added by the back propagation of a token, Ivan proposes that a programmer primes communication links between destinations and sources with tokens. Possibly, the programmer could insert extra tokens – one to enable sending the first data item, and possibly at least one more to avoid the penalty of the full communication delay from a destination to the source. This schema requires much further investigation and likely a simplification. That is, Ivan and I agree that this schema involves a large degree of complexity that, at the moment, we may not fully appreciate. Some of the sources of complexity are priming the tokens, making sure that tokens are sent back to the correct source, and especially dealing properly with boundary conditions such as terminations of a standing instruction.

SLIDING INSTRUCTION WINDOW

The notion of the pool of concurrent instructions opens up possibilities for concurrency in instruction issue that go far beyond anything I have seen. We can have several ships sending code bag descriptors to the instruction fetch unit, which eventually results in the bag of all code bags being issued concurrently.

A very interesting property of a Fleet with a pool of concurrent instructions is that instructions can be added to the pool in a completely asynchronous manner by any ship that sends a code bag descriptor to the instruction fetch unit. This asynchrony allows different parts of computation to proceed at their own pace and request new instructions as needed. While I marvel at the wide range of possibilities for concurrency, I have concerns about viability of such a system with a very few limitations. In particular, I find myself wondering how one would be able to debug Fleet programs where we have such a large degree of uncertainty about the timing of instruction issue and execution. My experience with concurrency has taught me one consistent lesson: Concurrency is hard and should be used wisely. Consequently, I see us having two options. We may be able to discover a method for reasoning about such a vastly concurrent system, a major feat on its own. Let us recall a lecture by Alan Turing from early fifties, where he eloquently stated that he could imagine that computer circuits can be built without a time sequencing mechanism, namely the clock, but that he cannot imagine how one could reason about such a system, and how one would deal with complexity. I believe Fleet with its pool of concurrent instructions may be presenting us with a similar challenge.

I started writing this memo intending to explore possibilities for introducing a sequencing mechanism that would allow us to simplify programming and debugging Fleet without sacrificing too much of concurrent execution of instructions.

In previous sections we touch the topic of introducing a sequence by passing code bag descriptors from a ship to a ship, and finally, when the new block of instructions can be issued, the last ship in the execution sequence passes the code bag descriptor the instruction fetch unit.

Such a mechanism has a number of weaknesses. We already noted that we may be wasting energy by moving code bag descriptors over and over again through the switch fabric and through ships. Moreover, passing a code bag descriptor through a sequence of ships has quite a linear nature. Not all computations, however, have such a linear nature. To accomplish synchronization of various branches of a computation we would likely require special ships that would synchronize the branches. These ships may not be completely trivial because results of the computation in different branches may influence what code bags need to be issued¹.

Let us consider whether lessons from design of synchronous circuits can help us in introducing sequencing into an asynchronous Fleet processor. Recall that the purpose of the clock is to slice time into regular predetermined intervals. Such discreteness of time allows designers to know exactly when computation may be complete, which is precisely the challenge we face with Fleet. We can easily expand the basic concept of

¹ We require additional insight into the nature of applications that could be targets for Fleet processors. Examples would be JVM running typical web services, and a database.

the clock by relaxing the definition of the clock. For example, the clock does not have to tick at regular predetermined intervals. Instead, we could have a clock signal that still provides global synchronization, but is generated whenever the work assigned to the current slice has been completed. Such a view of the clock brings us a step closer to the concept of asynchronous circuits, where we replace the notion of a single global synchronization signal with multitude of local synchronization signals generated by a local group of circuit components.

So, let us explore whether we can introduce a notion of global synchronization into Fleet and have a gain in simplicity without making too large of a sacrifice in the degree of available parallelism. Let us use the name "logical clock" for the synchronization mechanism that we are about to explore.

To introduce a logical clock, we first need to decide where we could benefit most from introducing discreteness into time. The part of Fleet that gives me most concern is the ability to extend the instruction pool at any point in time. That is, any component can, at least in principle, send a code bag descriptor to the fetch unit at any point in time, causing expansion of the instruction pool. We do not know at what point in time new instructions will appear in the instruction pool, neither do we know whether any of these instructions will execute before any of the instructions that have already resided in the instruction pool.

I question whether we really require such an amount of uncertainty and nondeterminism. I also have a difficult time imagining methods for programming and especially debugging such a nondeterministic system. Therefore, I propose that we examine consequences of introduction of a logical clock into the mechanism of adding instructions to the instruction pool and into making fetched instructions available for execution.

Note that, so far, we have not introduced a distinction between instruction fetch and addition of instructions to the instruction pool. That is, in Fleet as it stands today, we can only supply a fetch unit with a code bag descriptor, and by the virtue of such action, instructions from the code bag would eventually appear in the instruction pool.

I find it useful to have ability to ask a fetch unit, at any time, to prepare additional instructions for inclusion into the instruction pool. Such advance notice provides a buffer between the time of announcement that instructions would be required to the time when the instruction is released for execution. That is, we have a natural time window to prepare instructions for execution, which we should keep. This means that we should introduce discreteness into time at the point of adding new instructions to the instruction pool.

This means that we need determine when is the right time to release additional instructions into the instruction pool. As a starting point, we borrow a decision that is

commonly made in circuit design and assume that the work that a Fleet program needs to accomplish is split into phases. Instructions that define a subsequent phase are released into the instruction pool only when the work that needed to be done by the previous phase was accomplished.

We can illustrate such an approach as a sliding instruction execution window shown in Figure 5. Execution of a program is broken into a sequence of segments. Fleet slides its focus window across the sequence of segment, executing instructions in the segment that is in focus.

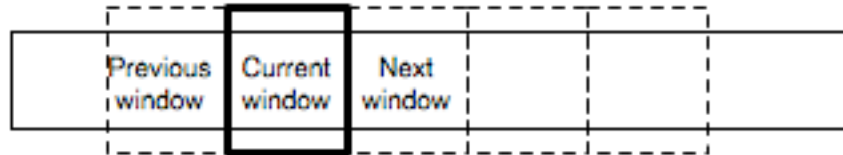


Figure 5: A sliding instruction execution window

How does Fleet know when to move the instruction execution window to the next segment? In other words, how does Fleet know when the work that has to be completed in the current segment has actually been completed? We could, for example, put together a token counting mechanism similar to the one we arrived at in our discussions with Steve Furber: When a move instruction completes, the destination ship sends a token to the logical clock unit. The logical clock unit knows about the number of instructions that need to be completed before the next clock tick takes place. For each token received, the logical clock unit decrements the count, and when the count reaches zero, the clock tick takes place and releases instructions from the subsequent instruction window into the instruction pool.

Note that there may be some flexibility in release of tokens that decrement the instruction count. For example, in a horn-and-funnel switch fabric, such as the one that was used in [3], we could contemplate sending a token as soon as an instruction reaches the trunk, or any trunk if we have more than one. If that feels too adventurous, then we could still send the tokens from the destination ship, but perhaps sooner than when the ship has completed the work. For example, sending a token as soon as the destination has received the data may be a good idea, creating some time slack for transmission of the tokens. Such an early transmission of the token may be especially useful for long-latency ships such as memory access ship.

Note that work that needs to be done in the current instruction window does not necessarily need to equal completion of all instructions in the current instruction window. We could, for example, mark a sub-bag of instructions that need to be completed and use their completion as a logical clock. We could even contemplate a scheme where we could, for each instruction decide in how many instruction windows it needs to complete. For example, a move that triggers memory read could be marked for completion in 10 instruction windows. At the moment, I do not have enough knowledge and experience to

be able to discuss, with any degree of authority, advantages and disadvantages of various schemes.

How can we accomplish delay in issuing instructions into the instruction pool. We could line the fetched instructions in a queue, and, upon the go-ahead signal, empty the queue. What if all instructions that comprise the current instruction window have not been fetched by the time their release into the instruction pool. The only option that I see so far is that we have to delay issue of the subsequent instruction window until all of the instructions from the current window have been issued. If our criteria for issuing the subsequent instruction window require completion of all instructions from the current window, then a premature release of the subsequent instruction window is not possible, because instructions that have not been released have clearly not completed yet.

The possibility that not all instructions from the current instruction window have been fetched by the time of instruction issue makes one question how the current instruction window is constructed. So far, I have tacitly assumed that an instruction window consists of instructions from several instruction bags. That is, I allowed code bag descriptors be sent to the fetch unit at any point in time. This means that some of the instructions in the current instruction window may specify a move from a source to the instruction fetch unit, and that the additive union of all corresponding instruction bags will form the instruction pool for the next execution window.

Note that we can have a tricky situation on our hands if we relax criteria for completion of the current execution window. Assume that one of the move instructions in the current execution window stipulates a move of a code bag descriptor to the fetch unit. Also assume that completion of this instruction is not a part of completion criteria for the current instruction window. That is, we may assume that instruction fetch is a slow operation and we may want to issue a fetch request ahead of time, so that the instructions are ready when we need them. However, if the fetch unit is fast, it may fetch instructions before the current execution window is complete, thus adding a bag of instructions to the subsequent execution window – sooner than we planned to. This can have unforeseen effects. We can prevent such premature instruction issue if we request that completion of fetch moves is always required for completion of the current execution window. At the moment, I cannot say whether this is an efficient way to handle this issue. Maybe an option would be to specify, precisely, for each move instruction, the number of execution windows in which it needs to be completed. Each source would keep track, for the move instruction it is executing, in how many execution windows the result should be released, and would not release the result prematurely.

The paragraphs above emphasizes that it is important that instructions be added to the instruction pool at a correct time, not prematurely, and not too late.

So far, we have assumed that the fetch unit does not release the next batch of instructions into the instruction pool until the logical clock signals the time to move the

execution window. In a horn-and-funnel implementation of the switch fabric, this would mean that no move instruction would enter the instruction horn until the logical clock ticks. This means that instructions have to traverse the instruction horn tree before they reach their sources. One could ask whether we could move instructions closer to sources to shorten the latency that we incur when releasing new instructions to the instruction pool. Ivan argues that we should keep instructions in one central place, so that their release is easy to control. I believe his argument has lots of merit.

However, this memo is about presenting options. Hence, let me describe, very briefly, one other possibility. Suppose we insert a queue in front of each source, and use the queue to accumulate move instructions intended for that source in the subsequent execution window. When the logical clock ticks, we would insert a marker item at the end of the queue and start releasing instructions to the source. When the marker reaches the exit side of the queue, the release of instructions stops and a signal is sent to the logical queue generator. This way we may be able to avoid the latency of travel of an instruction through the source horn. This arrangement, of course, assumes that our criteria for completion of an execution window requires that at the very least, the switch fabric delivers all move instructions to their respective destinations. Relaxing this criterion for completion would, as far as I can see at the moment, very much complicate a schema for accelerated delivery of instructions.

Note that the concept of the logical clock does not necessarily contradict the notion of standing orders [1]. We could, for example, assume that one instance of a standing-order instruction per ship executes in one instruction execution window. We could also, I believe, extend the mechanism for accelerated instruction delivery to be able to handle standing-order instructions. This could mean that the same ship could be used in a Fleet that does not know about standing orders and a Fleet that uses standing orders.