UC Berkeley Computer Science

Subject: FLEET – A One-Instruction Computer
Date: December 5, 2005
From: Ivan Sutherland
Number UCIES #2005-is14

References:

UCIES# 2005-is02: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2005
UCIES# 2005-is03: Defining Some SHIPs, Ivan Sutherland, 24 August 2005
Coates, W., Lexau, J., Jones, I.W., Fairbanks, S., & Sutherland, I.E., "*FLEETzero:* An Asynchronous Switching Experiment," Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2001), April 2001.

PURPOSE

FLEET has evolved during the Fall of 2005. Because my earlier description is no longer adequate, I offer this memo as a fresh summary and overview of FLEET, replacing my memo of 24 August 2005. I hope this new memo may serve as a starting point for readers new to FLEET. I apologize in advance for any incompleteness of the FLEET design. This is research, and so we don't know all the answers yet.

BACKGROUND

Nearly every computer designer has his own favorite computer design. Here is mine. I have dreamed for many years about the ideas offered here and I plan to spend time in the future understanding their implications.

I first got interested in a computer with only one instruction at Caltech in the late 70s where we considered the design for a machine called Our Machine, also known as OM. At Sun Microsystems in the late 90s we built an asynchronous test chip, called FLEETzero, [see reference above] to demonstrate an asynchronous "switch fabric" capable of carrying information from several sources to several destinations using a pipeline switch with very high throughput albeit considerable latency. The question remained whether or not such a switch fabric can form the basis of an interesting computer architecture.

A group of half a dozen very bright UC Berkeley graduate students, a few industrial guests, and I have spent Monday afternoons all Fall discussing FLEET and improving it. I thank them for their lively and stimulating discussion, their programming and design examples, and especially for their energy and enthusiasm. The students wrote a variety of simple programs for the machine to explore its limitations. Those involved were graduate students, Dominic Antonelli, Greg Gibeling, Nemanja Isailovic, Adam Megacz, Andrew Schultz, and Trevor Meyerowitz, and three industrial visitors, Igor Benko and Mike Holenderski from Sun and also Marly Roncken from Intel.

One exciting new idea dominated our thinking. Because its switch fabric can carry several messages concurrently, FLEET is highly concurrent instead of sequential. To think about FLEET we abandoned "blocks" of instructions because they carry an implied internal

---

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.

sequence. Instead, we think of "bags" of concurrent instructions, called "code bags," that are entirely free of internal sequence. FLEET executes such instructions concurrently.

The change from sequential to concurrent execution of instructions has profound implications that are the main content of this memo. First, the program counter and jump instructions vanish. They are replaced by a data type called a "code bag descriptor" that defines the location and extent of a code bag. Second, the programmer stands face-to-face with concurrency. Sequential behavior is so familiar that abandoning it is hard. Third, because instructions in a code bag may execute in any sequence a literal can no longer be a piece of data separate from its use; a literal must be recorded as a value-destination pair. Fourth, because instructions are concurrent, FLEET can issue "standing instructions" that can assemble FLEET's parts together into a special purpose pipeline processor.

This has been an exciting time. Concurrency – how to make computers do many things at once – is the forefront of computer science today. FLEET's current design is right at the boundary between what we understand how to do and what we believe possible. So positioned, it offers fertile ground for interesting research.

MOTIVATION

When computers were new, logic and storage were expensive and wires were relatively cheap. Early computer designers avoided the expense of logic gates wherever possible but were not greatly concerned about the cost of communication. They made good design choices consistent with the costs of the day. My favorite example is the jump instruction. Early designers put jump instructions at the end of each block of code to avoid the expense of storing the address of the next block while executing the present block.

Today's chip fabrication methods invert the older cost balance. In today's integrated circuits, logic and memory are now almost free but communication costs dominate chip area, energy consumption and delay. In spite of these changes in the stuff from which we make computers, vestiges of the past remain in most of today's microprocessors. For example, jump instructions still appear at the end of each basic block of code in spite of the need to pre-fetch the next block while executing this one. A modern design might better begin each block by giving the length of current block and a pointer to the next block.

Instead of following the path of history, I seek to design a modern computer by listening carefully to what today's chip structures have to teach. I see three major lessons. First, simplicity and replication can reduce cost. Second, moving data will consume the majority of the time, energy and chip area. And third, the low cost of logic makes concurrency available if we can figure out how to use it.

BASIC STRUCTURE – Figure 1

SIMPLICITY – The FLEET architecture seeks simplicity by treating all processing devices alike. A collection of processing devices, known as SHIPs, together form a FLEET. An individual SHIP may have any number of inputs and any number of outputs. A SHIP acts after enough of its inputs receive data values. The SHIP may, but need not, consume those input values when it acts. At some later time, not specified by the architecture, the SHIP produces outputs which must persist until accepted by the switch fabric. The FLEET architecture is silent about the details of the processing done by SHIPs.

Although the FLEET architecture is silent about what individual SHIPs do, a particular FLEET design must specify the number and behavior of its SHIPs. Only by

examining the behavior of different FLEETs of SHIPs for a particular application can one learn which combinations of functions prove useful.

COMMUNICATION – The FLEET architecture seeks to control the cost of communication by putting it under direct programmer control. Thus FLEET avoids instructions like ADD or STORE that include concealed communication to and from a register file. Instead, the FLEET architecture provides only one instruction: MOVE. Where the data elements go determines what happens to them. For example, data elements delivered to an input of an adder combine with data elements at other inputs to form sums. Data elements entering a memory will be retained until recalled. Data elements delivered to an output unit will appear outside the FLEET system.

SHIPs communicate through a SWITCH FABRIC that moves data among them. The MOVE instructions of the program direct the switch fabric to transport data values from SHIP outputs to SHIP inputs.

Although the FLEET architecture requires the existence of a switch fabric, the architecture remains silent about how to implement the switch fabric. The switch fabric must transport data from any of several "sources" to any of several "destinations." The sources and destinations are, of course, exactly the outputs and inputs of the processing devices, or SHIPs, comprising the FLEET. The terms "source" and "destination" are relative to the switch fabric rather than to the SHIPs. The switch fabric's destinations are "inputs" of SHIPs, and the switch fabric's sources are "outputs" of SHIPs. Remember that output = source and destination = input. "Source" and "destination" refer to the switch fabric whereas "input" and "output" refer to the processing units or SHIPs.

The FLEET architecture is also silent about the speed, throughput and structure of the switch fabric. The architecture requires only that the switch fabric be able to wait as long as required for each SHIP to act. Thus a MOVE operation cannot begin until data are available at its source, the output some SHIP. Similarly, a MOVE operation cannot finish until space becomes available at its destination, the input of some SHIP. MOVE operations don't generally overwrite values; overfull inputs form queues.

CONCURRENCY – The FLEET architecture permits switch fabrics that execute several MOVE instructions concurrently. In fact, the FLEET architecture assumes concurrency nearly everywhere. This is in sharp contrast to our usual thinking about sequential machines and sometimes makes it hard to think about how a FLEET machine works. Make no mistake, concurrency is hard to contemplate, and the concurrent behavior of FLEET provides a delicious complexity.

Concurrency in FLEET appears in three ways. First, FLEET assumes concurrent fetch of instructions. FLEET groups its instructions into "code bags" and treats the instructions in each bag as entirely concurrent. This is unlike the instruction set architecture of a conventional sequential computer that uses the address sequence of instructions in memory to impose a completion sequence on them. FLEET avoids any correlation between the order of instructions in memory and their execution sequence. FLEET may fetch all of the instructions in a code bag without regard to their address ordering.

This form of concurrency appears in the behavior of a special SHIP, called a "fetch unit," that moves bags of instructions from main memory or the instruction cache, called I$ in the figure, into the "instruction pool." Because the instructions in each code bag are devoid of internal sequence, the fetch unit is free to pick a fetch sequence convenient to itself. For example, if the "last" part of a code bag happens to be in a faster instruction cache than the "first" part of that code bag, the "last" instructions may enter the instruction

pool first. Because the instructions of each code bag are concurrent, they may enter the instruction pool in any sequence, in groups, or all at once.

Second, FLEET assumes concurrent execution of instructions. Each Instruction remains in the instruction pool until it can execute by passing data to the switch fabric for delivery. An instruction must wait in the pool until its data are available at the output of its source SHIP. Because data for a particular instruction may appear only after some other actions are complete, an instruction may have to wait in the pool for other instructions to complete. New instructions entering the pool mingle freely with all the yet-to-be-executed instructions that remain in the pool from code bags fetched earlier.

Third, FLEET also assumes concurrent completion of instructions. The FLEET architecture must assume that the switch fabric is concurrent, because it might be. The switch fabric is free to complete instructions from the pool concurrently, in groups, or in any sequence, subject only to the constraints of data availability at the specified source and space availability at the specified destination.

In some cases, like MOVE A to B and MOVE X to Y concurrency in the switch fabric causes no problem because the destinations are distinct. However, two instructions with a common destination create uncertainty about the sequence of data arrival. For example, the concurrent pair [MOVE A to X and MOVE B to X] may deliver A and B to X in either sequence. The sequence may not matter if X is the input of an accumulator, but for other SHIPs it may matter a lot.

Only one sequencing rule applies to the switch fabric. Repetition of the very same instruction preserves the data source sequence at the destination.

Assuming concurrency in the switch fabric makes many implementations possible. One can imagine switch fabrics all the way from a single sequential bus to a complete cross bar. In between lie many interesting designs that optimize simplicity, throughput, space, or energy.

The switch fabric may even consist of several separate mechanisms specialized to particular data types. In addition to the widest fabric for transporting numeric values, for example, a narrow fabric specialized for transporting Booleans might not only be energy efficient but also offer lower latency than sending such a simple data type through a wide data path. Similarly, it may prove useful to transport characters in a separate fabric.

The switch fabric need not have uniform latency. Simplicity in the fabric for transporting Booleans might offer lower latency than found in the parts of the fabric for wider words. Frequently used paths might be built to have reduced latency at the expense of greater latency in less frequently used paths or in paths serving slow SHIPs.

ASYNCHRONY

For many years I've been interested in asynchronous design. The FLEETzero chip that we built at Sun Microsystems [see reference above] demonstrated an asynchronous form of switch fabric. Asked to move a data element, an asynchronous switch fabric will, before starting, wait as long as necessary for data to appear at the source. Before finishing it will also wait as long as is necessary for space to appear at the destination. In between start and finish the switch fabric must preserve the data value.

An asynchronous design for the switch fabric provides flexibility in the timing of the processing elements or SHIPs. Freedom from timing constraints offers those who implement SHIPs an opportunity to improve performance or reduce cost over time.

The FLEET architecture requires asynchrony only at the sources and destinations of the switch fabric. Synchronous implementations both of SHIPs and of the switch fabric are possible provided they use validity or occupancy signals to achieve the arbitrary delays required at sources and destinations. Individual SHIPs may also be synchronous or asynchronous because the FLEET architecture remains silent about their performance.

INSTRUCTION FETCH

FLEET's instruction fetch mechanism depends on two things. First, a special data type called a "code bag descriptor." Each code bag descriptor gives the location and extent in memory of a bag of instructions. Second, a special type of SHIP called a FETCH UNIT that interprets code bag descriptors, fetches instructions from memory and delivers them to the instruction pool. A FLEET must include one or more fetch units.

The programmer must use explicit MOVE instructions to send each code bag descriptor to a fetch unit. The fetch unit delivers the instructions of the new bag to the instruction pool, and is free to deliver them all at once, or in any convenient sequence. Multiple fetch units may deliver separate bags of instructions to the pool concurrently.

There is no jump instruction, nor any program counter to keep track of where the machine is in the sequence of instructions; indeed there's no sequence. Each code bag descriptor initially enters the system as a literal, but it may have been delayed, sorted, or selected by one or more SHIPs before reaching the fetch unit. If a program is to continue beyond one code bag, each bag must include instructions that deliver descriptors of one or more successor code bags to the fetch unit.

A bag of instructions may send many distinct code bag descriptors to the fetch unit, thus initiating concurrent threads of code. I prefer to call such threads "fibers" because they lack independent state. Fibers must share the state stored in all the SHIPs in the FLEET. Fibers must cooperate.

EXTERNAL LOCATION AND EXTENT – Figure 2

Notice that a bag of instructions is free of any internal representation of its own beginning, end or length. The hidden function of the jump instruction, namely to terminate the present block of code, is gone. Instead, the location and extent of the code bag is specified entirely outside the bag in the code bag descriptor. If code is stored in consecutive locations in memory, it may be possible for several code bag descriptors to use overlapping parts of a long sequence of code. As Figure 2 illustrates, compact representations for the start, body, and end of program loops may be possible.

DATA TYPES

A big lesson from modern programming languages is that data types are important. I want a type mechanism in FLEET. Each data type will have an escape form that I shall call "Out-Of-Band" (OOB). Out-of-Band is similar to but not identical to the "Not A Number" (NAN) form specified for some floating point operations. NAN can be a valid floating point representation for infinity, or other values that lie outside the usual range.

In addition to NAN, OOB values can represent termination and error conditions. One OOB value is the string terminator, LAST. Another OOB value can represent an error such as a memory parity error. Other OOB values can represent NAN.

Interesting FLEET implementations will have data words of at least 64 bits augmented by the type information. FLEET requires words longer than its numbers to accommodate data type information and the Out-of-Band values. Error checking requires additional bits. I think of 2**n + 10% as the right word length for FLEET. The 10% increment is arbitrary, but seems a good compromise between need and cost. A FLEET with 72 bits in memory seems to be about right.

FLEET should include at least the following types:
token: a dataless event useful only for sequencing, or OOB
boolean: TRUE, FALSE, or OOB
character: unsigned 16 bits or OOB
integer: 32 bits with sign or OOB
long: 64 bit signed value or OOB
code bag descriptor: a reference, possibly by name, to a bag of code or OOB
memory pointer: enough bits to address lots of memory or OOB.

Each of these types includes an Out-of-Band type called LAST. In addition to TRUE and FALSE, the boolean type will include at least a representation for LAST and perhaps other OOB values. Thus representing a boolean will require at least two bits. In addition to the 2**16 characters representable in 16 bits, the character type will include a representation for a LAST or terminator character. Thus the character representation will require at least 17 bits. Similarly, representations for the integer and long forms must each include at least one extra bit to provide for OOB values.

SHIPs may exhibit useful behavior when given OOB inputs. For example, a memory read SHIP may produce OOB data whenever given an OOB address. Similarly, a SHIP intended to check address bounds may produce OOB memory pointers when given inputs outside its defined range. Again, a stride SHIP intended to produce array indices may produce the OOB memory pointer, LAST, after the end of each sequence of addresses.

The behavior of many SHIPs can usefully depend on data types. For example, the address input of a memory read controller expects a memory pointer. If it gets any other type it may declare an error, ignore that input, or offer some other exceptional behavior. If given a LAST memory pointer it may deliver a LAST output value without ever accessing memory, perhaps choosing a data type like that of its most recent valid output.

SEQUENCE

The FLEET architecture provides two separate ways to control sequence. First, the flow of data through SHIPs can determine the sequence of events. For example, consider an addition. Concurrent MOVE instructions can deliver data values to the adder's two inputs and deliver the adder's output as needed. However, the adder will produce an output only after receiving a pair of inputs. Thus, the MOVE that transports the sum will occur only after both MOVEs that delivered the data inputs. Although the three MOVE instructions entered the pool concurrently, they execute in a good sequence.

The data form called "token" helps control the sequence of operations without conveying an extraneous value. One use for tokens is for an end-to-end acknowledge in

pipeline configurations. A receiving SHIP can acknowledge receipt of a data element by returning a token to the sending SHIP. By waiting for such an acknowledgement before sending the next data element, the sender can avoid congestion in the pipeline. In more complex situations SHIPs can exchange tokens to force actions into a suitable sequence.

The second form of sequencing in the FLEET architecture involves bags of instructions. FLEET avoids any attempt to know when the instructions in a particular bag are all done because that would impose a heavy communication cost inherent in the spatial distribution of SHIPs. The SHIPs that provide and receive data know when data have arrived and thus know when instructions act, but we prefer to avoid sending completion reports from those SHIPs to a central location.

Instead, FLEET offers communication of code bag descriptors as a means of controlling sequence. Because code bag descriptors are first class data objects, the switch fabric can move them from one SHIP to another and deliver them to the fetch unit. Any SHIP can include inputs and outputs for the code bag descriptor type. Such a SHIP may retain multiple code bag descriptors, selecting for output a particular code bag descriptor appropriate to its circumstance.

For example, a comparator SHIP may receive as input not only two data values to compare, but also a pair of code bag descriptors. It compares the two values and delivers as output not only a Boolean representation of their relative value, but also the matching code bag descriptor. If sent to a fetch unit, that descriptor will invoke a bag of code appropriate to the result of the comparison. That same SHIP might also store a code bag descriptor for code appropriate for unusual circumstances such as an OOB numeric input. Treating code bag descriptors as data provides FLEET with program branches.

Sending code bag descriptors to a fetch unit provides for sequencing. Because instructions cannot enter the instruction pool before the fetch unit is told to fetch them, a SHIP can invoke bags of instructions in proper sequence. In the sorting example above, suitable code to handle the choice enters the pool only after the choice is made.

THE MOVE INSTRUCTION – Figure 3

The basic MOVE instruction in FLEET has a source field and several destination fields that specify the source and destinations of the transfer. Providing multiple destinations in the MOVE instruction is a convenient way to provide for data fan-out. We tried the alternative of providing only a single destination in each MOVE and thus only point-to-point communication in the switch fabric. Although possible, that choice requires using a special REPLICATOR SHIP when fan-out is necessary, incurring more trips though the switch fabric that cause both extra latency and increased traffic.

We also considered the alternative of having variable length MOVE instructions, each containing only as many destinations as needed. Although this might reduce code size, interpreting variable length instructions requires a way to identify where each instruction begins and ends. Fixed-length instructions are more in keeping with the concurrent nature of code bags because they make it equally easy to access any part of a code bag.

For reasons I shall discuss later, I believe that three is the right initial choice for the fan-out to destinations available in each move instruction. Thus for a FLEET with 64 sources and 64 destinations, a MOVE instruction requires 24 bits of address information, 6 for the source address and 6 for each of the destination addresses as shown in Figure 3a.

The FLEET architecture contemplates three embellishments to the MOVE instruction. The first embellishment sets the number of times the MOVE instruction can act. Different MOVE instructions act once, repeat a specified small number of times, or, as "standing MOVEs" act repeatedly until killed. Some additional bits, four appear in Figure 3b, in each MOVE instruction distinguish these cases. MOVE instructions with counts can save both the time and the energy required to fetch a repeated action again. Standing instructions can "wire up" the SHIPs of a FLEET into pipelines that can process vectors, character streams, or numbers representing signals. A special value, perhaps 0000, in the count field also identifies short literal instructions described in the next section of this memo.

The second embellishment to the MOVE instruction tells the source which data to send and the destination unit how to treat arriving data. For example, two extra source bits, marked "s s" in Figure 3c, and two extra destination bits, marked "d d," could provide flexibility in many SHIPs. One can think of these bits as combined with the source and destination address to provide a larger address space. Because the meaning of these bits depends on the design of the source and destination SHIP, some SHIPs may use these bits as a form of OP code. For example, these bits might cause an adder SHIP to negate an input, thus subtracting instead of adding.

The third embellishment to the MOVE instruction provides one source bit and one destination bit to control destruction of data. The source bit, marked x in Figure 3d, controls whether data are "copied" or "drained" when sent into the switch fabric. This was originally intended to provide for fan-out of data to multiple destinations by making all but the last of several MOVE instruction copy the data. The concurrency of instructions in a code bag, however, makes it difficult to identify the "last" instruction, and led to our putting fan-out in each MOVE instruction. Copy or drain may be useful for a stack SHIP to distinguish between read and pop, although there remains a difficulty associated with FLEET's lack of sequence.

The corresponding bit in the destination fields, marked y in Figure 3d, has a meaning that differs slightly in different SHIPs. In some destinations it controls whether data may overwrite existing data or must wait for vacant space. In other destinations it controls whether data persist when entering the SHIP or are for one-time-use. For example, making one input to an adder persistent makes it easy to add the same constant, perhaps a base address, to a series of other values.

WHY THREE DESTINATIONS?

Given their fixed length, how many destinations should each instruction have? The minimum choice is two, and I expect that ultimately FLEET may settle for that minimum. However, I want initially to choose three destinations for two reasons. First, I want to err with too many rather than too few destinations. The gain is programming convenience and the cost is code density. I want to see if programmers find three destinations useful or if the third destination remains largely idle.

My second reason for wanting three destinations is instructional; we will learn more from that choice. The replication of multiple destinations will probably occur in the switch fabric. Were we to include only two destinations there would be only one place in the switch fabric where data would split towards different destinations. It's probably easy to identify that place. With three destinations, the switch fabric must include two splits, no doubt a harder design challenge that I want us to consider. If we can design a switch fabric able to deal with three destinations we can probably extend that design to an arbitrary number of destinations. Surely we can shrink such a design to two destinations.

LITERALS – Figure 4

We need a mechanism to introduce compiled values into FLEET. Because the MOVE instructions in each code bag are concurrent, we found it difficult to associate a raw literal value in memory with a particular MOVE instruction to deliver it. Therefore, FLEET associates a destination with each literal value. In some sense, each literal is just a MOVE instruction whose source address is replaced by a compiled value.

Thus it is easy to contemplate two types of literals. The first type, for short values, as shown in Figure 4a, is identified by a special value in the count field; 0000 appears in the figure. This "short literal MOVE" has exactly the same format as an ordinary MOVE but pre-empts the source address to hold the literal bits. Such literals will serve for ZERO, ONE and other small integers. Each such value goes to the three destinations. Note that special SHIPs will provide sources for tokens and Booleans.

The second type of literal, for long values, as shown in Figure 4b, requires many more bits of storage. Because its format is so different from other MOVE instructions, FLEET will sequester such literals in their own area of memory. Each such literal will require two words of storage, one for its value and the other, mostly empty, for its destinations. Each long literal remains a MOVE instruction with a fixed source value rather than a source address.

Because the long literals are so different in form, the code bag descriptor must include an indication of the location and extent of the area in which they appear. Thus each code bag descriptor actually describes two bags of code, one containing ordinary MOVE instructions and another, possibly empty, containing literal MOVE instructions.

TOKENS AND PIPELINES – Figure 5

Acknowledging receipt of data from an asynchronous communication proves essential. The FLEET architecture provides the token data type for exactly this purpose. As shown in Figure 5a, a SHIP designed for pipeline service should produce a token at its input interface to acknowledge receipt of input data and indicate readiness to receive more data. Similarly, as shown in Figure 5b, a SHIP designed for pipeline service should include a token input in its output interface to accept such acknowledgements. The sending SHIP should refrain from releasing its next data value until if gets an acknowledgement from the receiving SHIP. Notice that the input interface includes a token source and the output interface includes a token destination.

SHIPs with such token inputs and outputs will readily connect together in pipelines with first in first out (FIFO) behavior as shown in Figure 5c. The arrows in the figure represent standing MOVE instructions that transport data from one SHIP's output to another SHIP's input. The figure shows an address generator or "stride" SHIP that accepts and retains a base address, a stride and a count. Thereafter for each token it receives on its output interface it produces the next successive memory address. The figure connects such an address generator to a memory read SHIP whose input interface produces a token in response to receiving a memory read address. The token produced at the input interface of the memory read ship indicates not only that the previous address has been accepted, but also that the memory read SHIP is ready to accept another address. When connected with a bilateral pair of standing MOVE instructions as shown in the figure, these two SHIPs cause the memory to read successive values as fast as it is able.

We are just beginning to learn how best to use such token-passing to synchronize operations. It appears that SHIPs with multiple inputs should produce a token

at their input interface only after receiving enough input values. Such a token may go to the multiple sources of such data using a multiple-destination MOVE instruction. It is less clear whether SHIPs that produce multiple outputs need only one token destination at their output interface or require as many token destinations at their output interface as they have data outputs.

It is important to note that two SHIPs connected in a pipeline require two trips through the switch fabric per data element passed. The simple connection shown in Figure 5c passes a data value and a token alternately through the switch fabric. Thus it passes one data item for every two trips through the switch fabric. If instead, one inserts two tokens in such a loop, the forward and reverse messages can be concurrent, for higher throughput.

Our present standard for pipeline interfaces makes both input and output interfaces “passive.” This means that the output interface produces a data value only after it receives a token on its token input. To get a loop like that of Figure 5c started, the programmer must “prime the pump” by inserting a token into the loop. Although other interface standards are possible we have chosen the passive form of output interface because it can be made active by inserting an initial token. Languages, such as TANGRAM [reference needed] provide a model for using tokens from which I hope to draw.

ESCAPE FROM STANDING INSTRUCTIONS

The existence of the Out-Of-Band (OOB) data form provides an escape from standing pipelines. Many SHIPs intended to participate in standing pipelines may include a place for a code bag descriptor that identifies some code to execute in the event that OOB data passes along the pipe. Thus, for example, the memory read unit of Figure 5c might include a code bag descriptor input and a code bag descriptor output not illustrated in the Figure. The program should give the memory read unit the descriptor for a code bag to fetch when the pipeline is no longer useful, as indicated by invalid data entering the read unit. A one time instruction attached to the code bag descriptor output could pass this descriptor to the fetch unit when appropriate.

After the address generator sends its final address, it can generate one more output, the OOB value called LAST. The LAST value serves as a separator between sequences of data values.

The LAST value causes two actions. First, LAST kills standing instructions. After sending the LAST signal to the read address unit, the standing instruction at the output of the address generator dies. The second action of a LAST value is taken by a SHIP receiving it. For example, when the LAST address arrives at its input, a memory read SHIP produces a LAST data output to pass the termination condition on down the pipeline. The memory read SHIP will also acknowledge receipt of the LAST input by sending a LAST token back to the stride SHIP. The standing instruction sent tokens back to the stride SHIP likewise sends the last token and then dies. The stride SHIP accepts but ignores the LAST token that arrives at its output interface.

To progress to the next operation, the memory read SHIP will also deliver a code block descriptor for transport to the memory fetch unit. Thus the LAST address from the address generator not only terminates the string, but also disassembles the pipeline and invokes appropriate code for the next process.

RESET

The existence of persistent values and standing instructions requires some way to reset SHIPs. Of course there will be a master clear process for the entire FLEET. But in addition, each SHIP probably needs a separate reset function that returns the SHIP to a pristine empty state. The clearing process will override data operations underway, but because it is outside the normal flow of operations, must be designed and used carefully.

My current idea is to endow each SHIP with a Boolean output that indicates its status. If the address of this Boolean is used as a source but copied, it will merely report the status of the SHIP, e.g. running or idle. However, if this Boolean is used as a source and drained, the SHIP will reset; standing instructions at its output will vanish as will any internal state, and the SHIP will soon assume a known initial state. How this happens and how long it takes must be a function of the individual SHIP design.

I fear that we do not yet understand the implications of clearing an entire FLEET of SHIPs. Do we need to control the sequence in which they clear? What happens to data elements that are in flight in the switch fabric during the clearing process? We've work to do here.

THE DESIGN OF SHIPS

There are several special types of SHIPs worth mentioning. The first type can supplement the short literal form previously described. These are source-only SHIPs that produce common values not available from the short literal MOVE, such as the constants TRUE, FALSE, and TOKEN. FLEET may also need fixed sources for LAST elements of each data type. A real-time clock and a random number generator are examples of similar output-only SHIPs that produce values that change with time or upon use.

A second type of SHIP is a pure destination. I know of only one that I shall call the "bit bucket." The bit bucket need not be a real destination. Instead, the switch fabric could recognize this special destination address and save power by simply dropping values destined for it. This destination can also fill destination slots in MOVE instructions that require fewer than the available number of destinations.

A third type of ship, the "register" has one input and one output. A register stores a single value that may be overwritten. Because FLEET can execute more than one MOVE instruction concurrently, it is possible that the value in a register might be read and written concurrently. In order to avoid mixing old and new data, a register must therefore contain some kind of sequencing device to ensure that a complete read or write is finished before the second action starts. The special arbitration hardware required to provide such safety may render register implementations complex.

A fourth type of SHIP exhibits the First In First Out (FIFO) behavior of a pipeline. The simplest FIFO merely accepts values, stores them and presents them at its output in FIFO sequence. FIFOs can decouple the timing of an input stream from the timing of an output stream.

Some FIFO SHIPs can also process data elements as they stream through. For example, an adder might form sums of pairs of elements from two input streams. SHIPs of this type can have many inputs and a single output or many outputs and a single input or multiple inputs and multiple outputs. For example, a divider ship might have inputs for numerator and denominator and outputs for quotient and remainder.

In addition to having spatially separate inputs and outputs, SHIPs can accept multiple inputs or produce multiple outputs over time. For example, an accumulator SHIP accepts and accumulates the sum of values received at a single input over time producing one output after many inputs. On the other hand, an address generator may produce a string of output addresses at a single output in response to each receipt of a set of input data including base address, stride and count. SHIPs that combine processing and FIFO action may need to store code bag descriptors that can invoke special code to handle unusual cases such as the end of a stream of data.

A fifth type of SHIP compares values. A SHIP to choose the larger of two numbers might be useful, but it will be more useful if it not only selects the larger but also indicates which input it has chosen. It might deliver that indication in the form of a Boolean output value that another SHIP could use to select between two code bag descriptors. It may prove simpler for the very same SHIP not only to select the preferred value but also to select the code bag descriptor appropriate to its data choice. The chosen code bag descriptor should identify the code appropriate to the choice. SHIPs that choose between two code bag descriptors provide branching. A SHIP that chooses between many code bag descriptors is FLEET's equivalent of a case statement.

A sixth type of SHIP controls the flow of data to help ensure proper sequencing. The simplest such SHIP "joins" two streams of tokens, producing a token at its output only after receiving a token at each of its two inputs. A matching "fork" SHIP produces separate tokens on each of two outputs for each receipt of a token. Other SHIPs may count tokens, producing a single output token only after receiving N input tokens. Join and fork SHIPs that deal with data values may also be useful. For example, a SHIP might accept two input values in two separate destination addresses in its input interface and, only after both have arrived, deliver them to a single output in a known sequence. Another simple SHIP might collect two such inputs in sequence and deliver them as two separate outputs only after both have arrived. Because such SHIPs serve only to impose sequence in an otherwise concurrent machine, they have no analogy in sequential machines or programming.

WHAT TO DO

Because the FLEET architecture can accept a wide variety of processing units, called SHIPs, the design of SHIPs will be a continuing process. An earlier memo, UCIES# 2005-is03, offers a preliminary definition of some interfaces for SHIPs and combines them into some possible SHIP designs. Over time we will continue to explore these and other designs to learn what forms of SHIP are useful.

The process of managing FLEET's development is likely to be painful because concurrency is hard, and we must face it directly. We have a software simulator on which to try out our ideas, but it has yet to include complex SHIPs. We hope to implement a first version of fleet in an FPGA system at UC Berkeley during the Spring semester.

ACKNOWLEDGEMENTS

I want to thank the great students and visitors who have been working with me on FLEET. Their discussions and enthusiasm have led us to a place where building a FLEET is almost possible. There's much more to do, but together we will find it interesting and stimulating. Graduate education is the process whereby youth discovers new knowledge and teaches it to the more mature. So the Fall term has been. So, I hope, will be the future. There is much to learn and I remain eager.

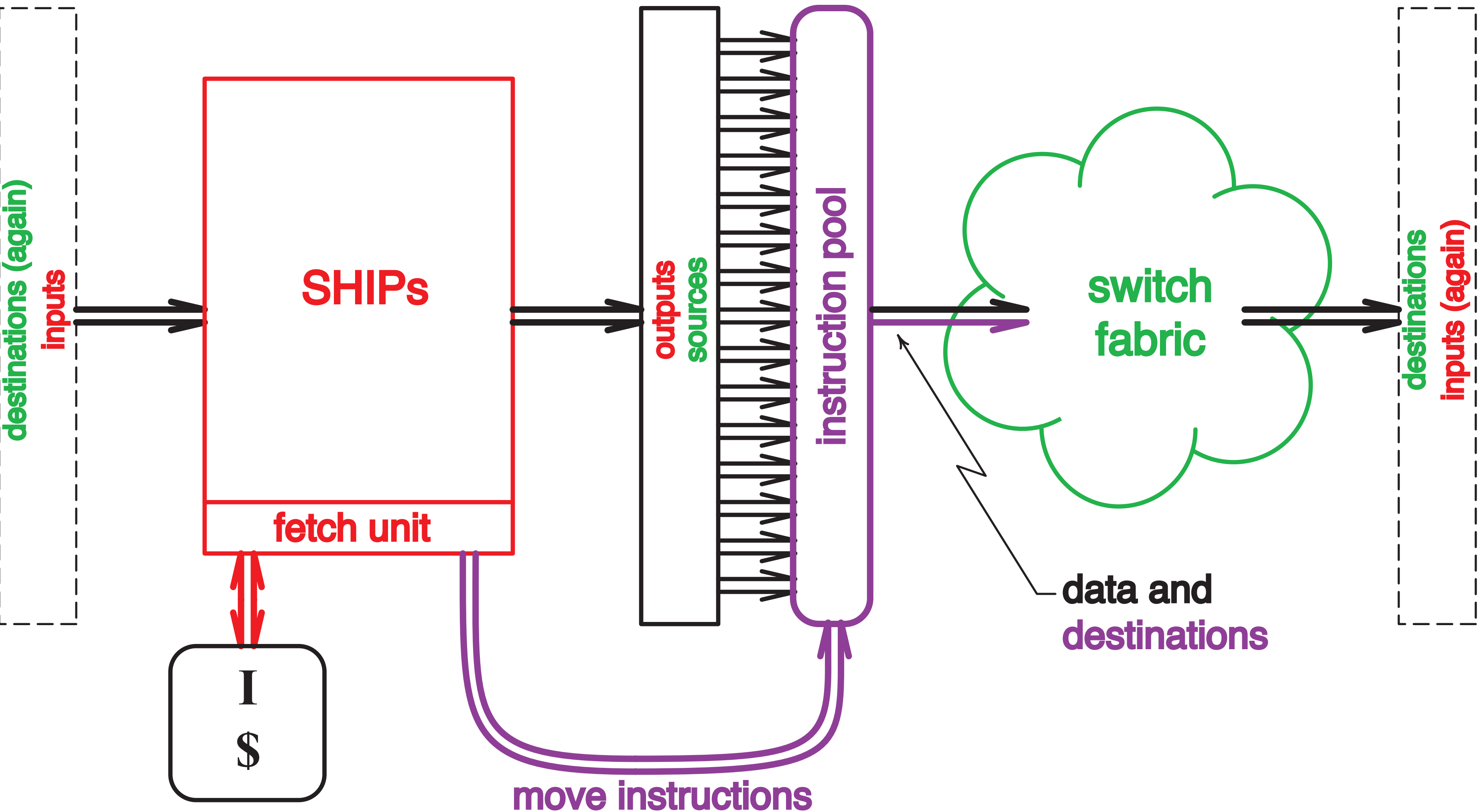
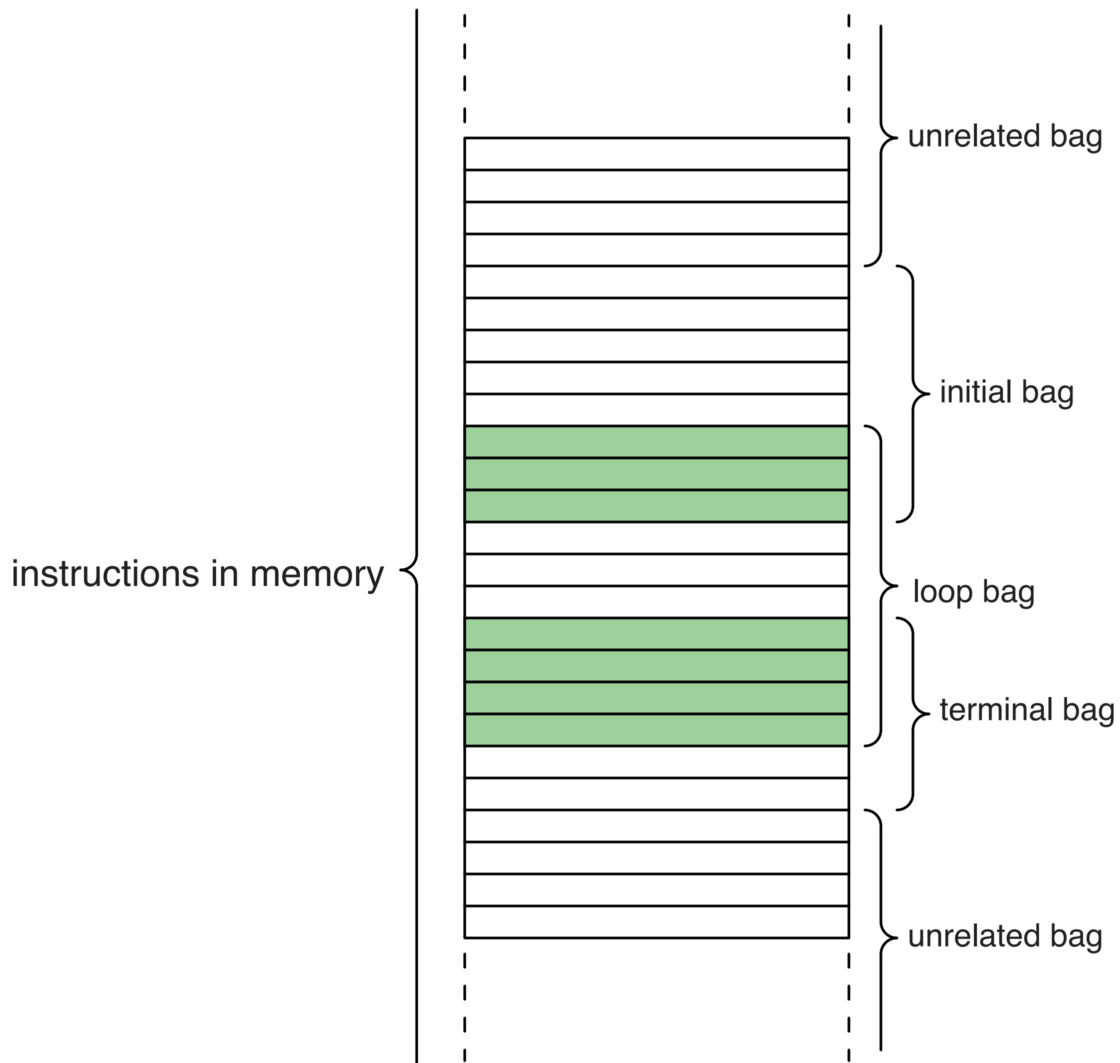


Figure 1: An Abstract View of FLEET



The shaded instructions appear in more than one code bag.

Figure 2: Code bags may overlap in memory

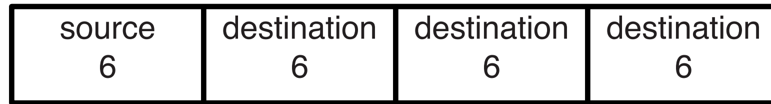


Figure 3a: Basic format

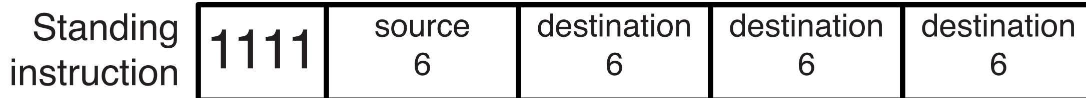


Figure 3b: With count field

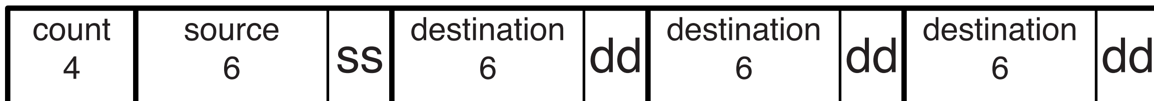


Figure 3c: With extra bits for source and destination

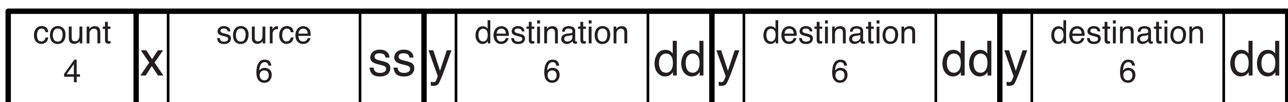


Figure 3d: Complete instruction format

Figure 3: Instruction formats

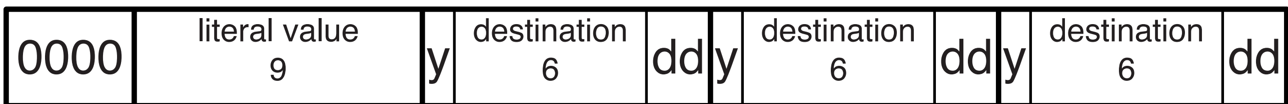


Figure 4a: Short literal

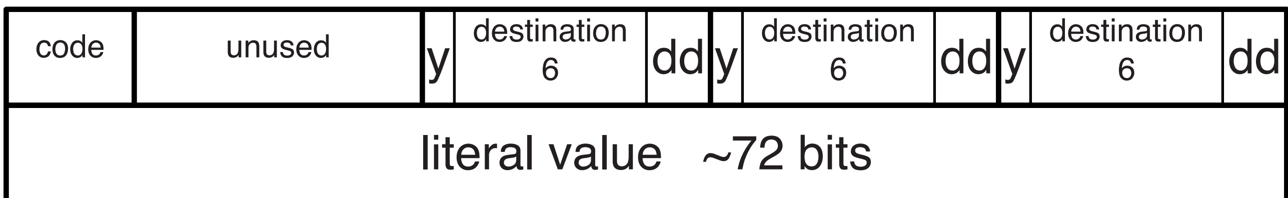


Figure 4b: Long literal

Figure 4: Literal instruction formats

A token appears here after each data input to say that the interface is ready for further input. An invalid data value produces a LAST token that kills a standing instruction after departure.

Data arriving here may be persistant but may never be overwritten. Each value fills the interface, blocking further input until the interface is ready.

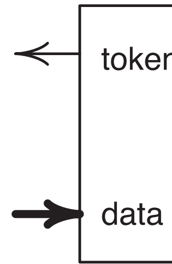
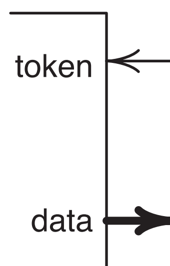


Figure 5a: Pipeline input interface



This destination must receive a fresh token for each new output data value.

A new data value appears here, if available, only after the previous value is drained and a fresh token has arrived. Move instructions may copy or drain data values. After departing, an invalid output kills a standing instruction.

Figure 5b: Pipeline output interface

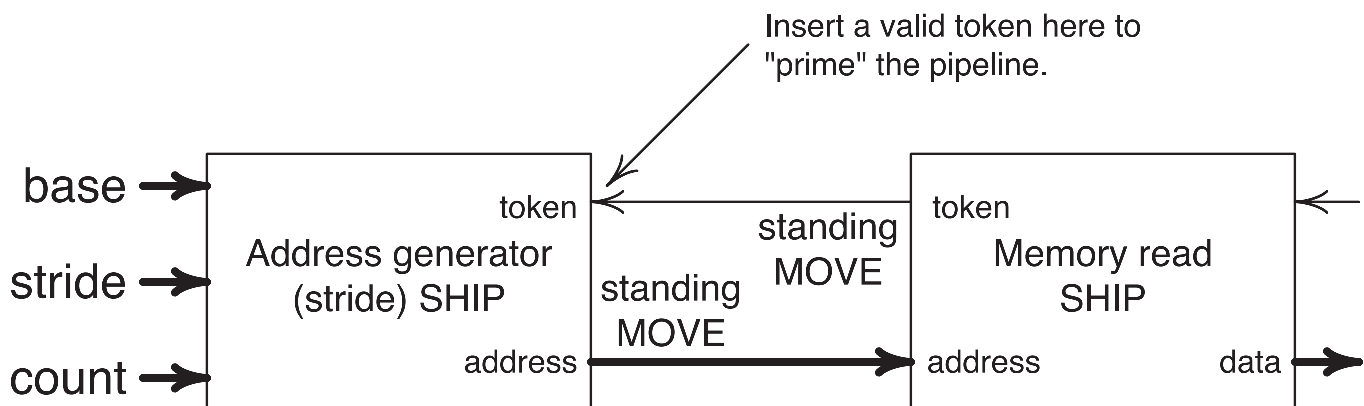


Figure 5c: A pipeline connection