

# UC Berkeley Computer Science

**Subject:** FLEET – A One-Instruction Computer  
**Date:** August 25, 2005  
**From:** Ivan Sutherland  
**Number:** UCIES #2005-is02

## References:

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

This memo provides a summary and overview of the FLEET computer as I now understand it. I offer this memo as a starting point for readers new to FLEET. I apologize in advance for the incompleteness of the memo. This is research, so I don't really know the answers yet. I've attempted to set forth as much as now seems useful.

## 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 more 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, or OM for short. At Sun Microsystems in the late 90s we built an asynchronous test chip, called FLEETzero, 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 now at hand is whether such a switch fabric can form the basis of an interesting computer architecture.

## MOTIVATION

When computers were new, logic and storage were expensive and wires were relatively cheap. Early computer designers avoided using logic wherever possible but were not greatly concerned with communication. They made 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 one.

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 many of today's common microprocessor designs. 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.

---

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.

It seems better today to store a pointer to the next block and the length of the current block early in each block of code.

Instead of following the path of history, I'd like to listen carefully to what modern chip structures have to teach about how one might build a modern computer. I see three major lessons. First, simplicity can reduce cost. Second, moving data will consume most of the time, energy and chip area of a modern computer. 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 when enough of its inputs have 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 on the details of the processing performed by SHIPs.

Although the FLEET style of architecture is silent about what individual SHIPs do, a particular FLEET design must specify the number and behavior of its SHIPs. By examining the behavior of different FLEETs of SHIPs I hope we can learn which combinations of functions prove to be useful.

The SHIPs communicate through a switch fabric that moves data among them. Move instructions direct the switch fabric to transport data values from SHIP outputs to SHIP inputs.

**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, a FLEET architecture provides only one instruction: MOVE. Where the data elements go determines what happens to them. For example, data elements entering an input of an adder will combine with data elements at other inputs to form sums. Data elements entering a memory will be retained until recalled. Data elements sent to an output unit will appear outside the FLEET system. The MOVE instructions of the program control the actions of the “switch fabric” that moves data elements within FLEET.

The FLEET architecture requires the existence of a switch fabric but remains silent on its implementation. There are many possible implementations of the switch fabric that vary in throughput and latency. Proper function of a FLEET machine must be independent of the particular implementation chosen for its switch fabric.

The switch fabric can 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 used with respect to the switch fabric rather than the SHIPs. The destinations to which the switch fabric delivers data are “inputs” of SHIPs. The sources accessed by the switch fabric are “outputs” of SHIPs. Remember that source = output 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 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 the SHIPs to function. Thus a MOVE operation cannot begin until data are available at the source, which is the output of some SHIP. Similarly, a MOVE operation cannot finish until space becomes available at the destination, which is the input of some SHIP. MOVE operations generally don't overwrite values; overfull inputs form queues. The architecture permits switch fabrics that can execute several MOVE instructions concurrently as well as switch fabrics that perform MOVE instructions sequentially.

**CONCURRENCY** – 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 what a FLEET machine will do. Make no mistake, concurrency is hard to contemplate, and the concurrent behavior of FLEET provides a delicious complexity. By making FLEET largely concurrent I hope to learn better ways to think about concurrency. At the present time this aspect of FLEET lies at the limits of my ability to understand what features will be effective.

Concurrency in FLEET appears in two ways. First, the FLEET architecture must assume that the switch fabric is concurrent, because it might be. Given a concurrent switch fabric, it makes sense to issue concurrent bags of MOVE instructions; the MOVE instructions in each concurrent code bag may complete in any order. The instructions in a code bag enter the "instruction pool" in any sequence convenient to the fetch unit, or in groups, or all at once. For example, if the left half of the code bag happens to be in a faster instruction cache than the right half of the code bag, its instructions will enter the instruction pool first. Moreover, the switch fabric is free to complete instructions in the pool concurrently, in groups, or in any sequence, subject only to the constraints of the availability of data at the specified source and the availability of space at the specified destination.

In some cases, like MOVE A to B and MOVE X to Y concurrency causes no problem. 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 either A or B to X first. 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 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, energy, or some other parameter. Indeed, the switch fabric may consist of several separate mechanisms specialized to particular functions. For example, it may be energy efficient to have not only a narrow fabric specialized for transporting tokens, but also a separate medium width fabric specialized for transporting characters, as well as a separate wide fabric specialized for transporting numbers.

Switch fabrics need not have uniform latency. Simplicity in the fabric for transporting tokens might offer lower latency than the parts of the fabric for wider words. Frequently used paths might have lower latency at the expense of less frequently used paths or 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 will store the data value.

One simplicity of an asynchronous design is that it offers flexibility in the timing of the processing elements. This provides enormous flexibility for implementers to improve the performance or reduce the cost of the processing devices in a FLEET system.

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 bits to achieve the arbitrary delays required at sources and destinations. Individual SHIPs may be synchronous, of course, because the FLEET architecture makes no demands on their performance.

## INSTRUCTION FETCH

The FLEET architecture omits the program counter. Including a program counter would imply that instructions have a natural sequence, an idea from which I shrink. Instead, the instruction fetch mechanism of the FLEET architecture is a special SHIP connected to the switch fabric. The programmer must use explicit MOVE instructions to send code bag descriptors to the fetch unit. Each code bag descriptor tells the fetch unit how to find the specified bag of instructions. The fetch unit adds the new bag of instructions to the instruction pool. The fetch unit is free to deliver the instructions of the code bag concurrently or in a sequence of its own choice.

There is no jump instruction. Instead, a MOVE instruction sends a code bag descriptor to the fetch unit. The source of the code bag descriptor may be the output of some SHIP or come from the literal mechanism. We have not yet defined FLEET's literal mechanism.

A bag of instructions may send as many separate code bag descriptors to the fetch unit as desired, 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.

Notice that a bag of instructions has no 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. This may provide a compact representation for the code in the start, body, and end of program loops.

The FLEET architecture specifies that if several code bag descriptors are sent to the fetch mechanism concurrently, the instructions they specify may enter the instruction pool concurrently. Instructions in the pool may complete in any sequence. Complex implementations of FLEET might have several concurrent fetch units. Such fetch units are just special SHIPs, able to fetch and issue the specified instructions. The instructions in each

code bag are concurrent which means that they may issue in any sequence convenient to the implementation, including simultaneous issue.

## DATA TYPES

A big lesson from modern programming 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 perfectly valid floating point representation for infinity, or other values that lie outside the usual forms.

Interesting FLEET implementations will have data words of at least 64 bits augmented by the type information. FLEET requires words longer than a power of two to accommodate the type information and the Out Of Band values, and additional bits are required for error checking. 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.

In contrast 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.

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 integer 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. Thus in addition to TRUE and FALSE, the boolean type will include a LAST form. In addition to the  $2^{*16}$  characters representable in 16 bits there will be a representation for a LAST or terminator character. Similarly the integer and long forms each include a representation for the last element in a string.

The choice to include a specific representation for OOB values implies an increase in word length. It takes at least 33 bits to store an integer, at least two bits to store a Boolean, and so forth.

SHIPs may exhibit useful behavior when given OOB inputs. For example, 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 also produce the OOB memory pointer, LAST, when it reaches the end of its action.

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 promptly deliver a LAST output, perhaps choosing the same type as the last valid output it delivered.

Many SHIPs will include inputs and outputs for the code bag descriptor type. Such ships may hold multiple code bag descriptors, selecting one for output depending on

conditions. For example, a comparator SHIP compares two values and provides as output not only a Boolean representation of their relative value, but also a code bag descriptor that identifies some code appropriate to that result. That same SHIP might also store a code bag descriptor for code appropriate to the exception of getting an OOB numeric input. Treating code bag descriptors as data provides FLEET with program branches.

## SEQUENCE

The FLEET architecture provides two separate ways to control sequence. First, the flow of data through the SHIPs can determine the sequence of events. For example, consider matrix addition. Suitable MOVE instructions can connect two address generators to the address input of the memory read SHIP and deliver successive values to the inputs of an adder ship. Because the adder accepts a pair of inputs for each output, the flow of data to and from the adder controls the master sequencing of the otherwise concurrent MOVE instructions.

The data form called “token” can help control the sequence operations without having to convey an unneeded value. For example, a receiving SHIP can acknowledge receipt of a data element by returning a token to the sending SHIP, whereupon the sender can send the next data element. In more complex situations tokens can control the sequence in which SHIPs act on their data.

The second form of sequencing in the FLEET architecture involves bags of instructions. We might make FLEET’s fetch unit able to control sequencing, but that has proven difficult in the absence of some way to tell when instructions are finished. Because they are distributed over space, only the SHIPs themselves can know when data have arrived without having to send a message somewhere. I prefer, therefore, to place responsibility for sequencing the code on code bag descriptors processed by the SHIPs. Because code bag descriptors are first class data objects, SHIPs can process them and pass them to the fetch unit. Obviously, instructions cannot enter the instruction pool before the fetch unit is told to fetch them and thus the SHIPs can ensure proper sequence of bag of instructions. However, because the instructions in the instruction pool are concurrent, unexecuted instructions from one code bag may remain when instructions from the next code bag enter the pool.

Because FLEET’s only sequencing guarantee lies in its SHIPs, it is important to include SHIPs with explicit paths for code bag descriptors. For example, to sort lists of data it is not enough to have a SHIP that compares pairs of numbers and selects the larger. A SHIP for sorting must also indicate which input it chose for consumption so that that input can be replenished. The SHIP might provide a Boolean output that could later select one of two code bag descriptors. I prefer to make the same SHIP both select the larger datum also select a code bag descriptor appropriate to its choice. The code bag descriptor it selects will describe code appropriate for fetching the next element of the chosen list. Sequencing is possible because the new code bag descriptor appears only when the comparison completes.

## THE MOVE INSTRUCTION – Figure 2

The basic MOVE instruction in FLEET has a source field and a destination field that specify the source and destination of the transfer. In a FLEET with 64 sources and 64 destinations, the basic MOVE instruction requires 12 bits of address information, 6 for the source address and 6 for the destination address, as shown in Figure 2.

The FLEET architecture contemplates three embellishments to the MOVE instruction. The first embellishment sets the number of times the MOVE instruction can act. Any move instruction can act once, can repeat a specified small number of times, or can be a “standing instruction” that repeats an unspecified number of times. Some additional bits, four appear in Figure 2, in each MOVE instruction provide for such a count. MOVE instructions with counts can save both the time and the energy required for instruction fetch. Standing instructions can “wire up” the SHIPs of a FLEET into configurations that can process vectors, character streams, or numbers representing signals.

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 the figure, and two extra destination bits, marked “d d” in the figure, could provide flexibility in many SHIPs. One can think of these 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 performing subtraction rather than addition.

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 the figure, controls whether data are “copied” or “drained” when sent into the switch fabric. This is useful when data from one source must travel to many destinations. All but the last MOVE instruction that takes such data may be marked to leave the data in place. A stack ship provides one example of how such marking may be useful because one may wish either to read the top of the stack or to read and pop the stack.

**Warning:** how do we know which is the “last” of several MOVE instructions issued concurrently to a source? It may be better for each MOVE instruction to specify multiple destinations. If so, how many destinations: one, two, many, or an arbitrary list?

The corresponding bit in the destination field, marked y in the figure, 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.

## TOKENS AND PIPELINES

In asynchronous communication we often see the value of acknowledging receipt of data. The FLEET architecture provides the token data type for exactly this purpose. A SHIP designed for pipeline service should produce a token at its input interface to acknowledge receipt of the input data it requires. Similarly, a SHIP designed for pipeline service should have a token input destination in its output interface to accept such an acknowledgement from a receiver. The sending SHIP should refrain from releasing its next data value until it 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. Consider, for example, an address generator SHIP that accepts and retains a base address, a stride and a count. Thereafter for each token it receives on its output interface it will produce the next successive memory address. Such an address generator might be connected to a memory read SHIP whose input interface produces a token in response to receiving a memory read address. The

token thus produced would indicate not only that the previous address was accepted, but also that the memory read SHIP was ready to accept another address input. If two such SHIPs communicate with a bilateral pair of standing MOVE instructions, the memory will fetch successive values from memory 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 connecting two SHIPs in a pipeline requires two trips through the switch fabric per data element passed. The simple connection that exchanges a data value and a token sequentially will thus operate at half of the potential speed of the system. Instead, one may insert two tokens in such a loop. With two tokens the bilateral communications may be concurrent, resulting in more rapid data transport.

At present we plan to ask the programmer to “prime the pump” by inserting the required tokens. The notion of “active” and “passive” interfaces seen in some languages, such as TANGRAM, is missing here. Instead the programmer must activate interfaces as needed by inserting an appropriate initial token.

## **ESCAPE FROM STANDING INSTRUCTIONS**

The existence of the Out Of Bound (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 described above might also include a code bag descriptor input and a code bag descriptor output. The program should give the memory read unit a descriptor of the 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 would pass this descriptor to the fetch unit should it be needed.

When the address generator reaches the end of its set of addresses, it might generate a single LAST address output, an OOB value. This LAST output might serve two functions. First, after sending it to the read address unit, the standing instruction at the output of the address generator would die. Second, when the LAST address arrives, the memory read unit might produce a LAST data output to pass the termination condition on down the pipeline. The memory read SHIP might likewise cancel any standing instruction on its input interface. Further, the memory read SHIP might produce a value at its block descriptor output for transport to the memory fetch unit. Thus the LAST address from the address generator would not only terminate the string, but also disassemble the pipeline.

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 clear function that returns the SHIP to a pristine empty state. The clearing process will override data operations underway and must be designed and used carefully outside the normal flow of operations.

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. If this Boolean is used as a source and consumed, 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.

There are several special SHIPs worth mentioning. There are fixed sources such as those providing the constants 0, 1, TRUE and FALSE. Also, FLEET must have a fixed source of valid tokens for initiating token-controlled actions, and may need a source for the Out of Band token called LAST. A real time clock and a random number generator are also pure sources, but they provide values that change over time or with use. I know of only one pure sink; I call it the "bit bucket."

## THE DESIGN OF SHIPS

It seems clear at the outset that a few classes of SHIP will prove to be generally useful. One class of SHIP, that I have called the "register," stores a single value that may be overwritten. Overwriting data in a concurrent system can cause problems if the new value arrives just as the former value is used. The special hardware required to make overwriting safe may render simple registers too complex for general use.

A second class of SHIP has the behavior of a pipeline processor or FIFO. The simplest of these stores a sequence of values so as to decouple the timing of an input stream from the timing of an output stream. Such a SHIP could also process the data elements as they stream through. For example, it might form sums of pairs of elements from two input streams. Such SHIPs may need to process code bag descriptors that can invoke special code to handle unusual cases or the end of a stream of data. SHIPs of this class can also have a one-to-many or many-to-one correspondence between input values and output values. For example, an address generator may produce a string of output addresses in response to receiving an initial address and stride. Such an address generator can provide input to a pipeline. Again, an accumulator ship may compute a single sum from multiple inputs to terminate a pipeline.

A third class 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 more useful for the very same SHIP both to select the preferred value and to select the code bag descriptor appropriate to its data choice. The chosen code bag would contain the code required to handle the result of the choice. SHIPs with a choice between two code bag descriptors provide branching. One can also imagine SHIPs that implement a case statement.

A fourth class 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 one at each of its inputs. A matching "fork" SHIP produces separate tokens on each of two outputs for each receipt of a token. Join and fork SHIPs that deal with data values also seem 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, delivers them to a single source address in a known sequence. Another simple SHIP collects two such inputs in sequence and delivers them as two separate outputs only after both have arrived. Such SHIPs serve primarily to ensure proper sequencing and therefore are entirely unfamiliar to programmers of conventional machines.

## 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. A companion memo offers a preliminary definition of some interfaces for SHIPs and combines them into some possible SHIP designs. Over time I hope we can try out these and other designs to learn what forms of SHIP are useful.

The process is likely to be painful because concurrency is hard, and we will face it head on. We have a software simulator at hand on which to try out our ideas, but it has yet to include any complex ship types. We may need special software to check that our programs sequence code as we expect.

It is my hope that Berkeley's students and faculty will find problems in this design space that will interest them. I know that graduate education is the process whereby youth discovers new knowledge and teaches it to the more mature. I believe that there is much here to learn. I am now a "visiting scholar" at Berkeley, a position in which I hope to learn much more than I teach.

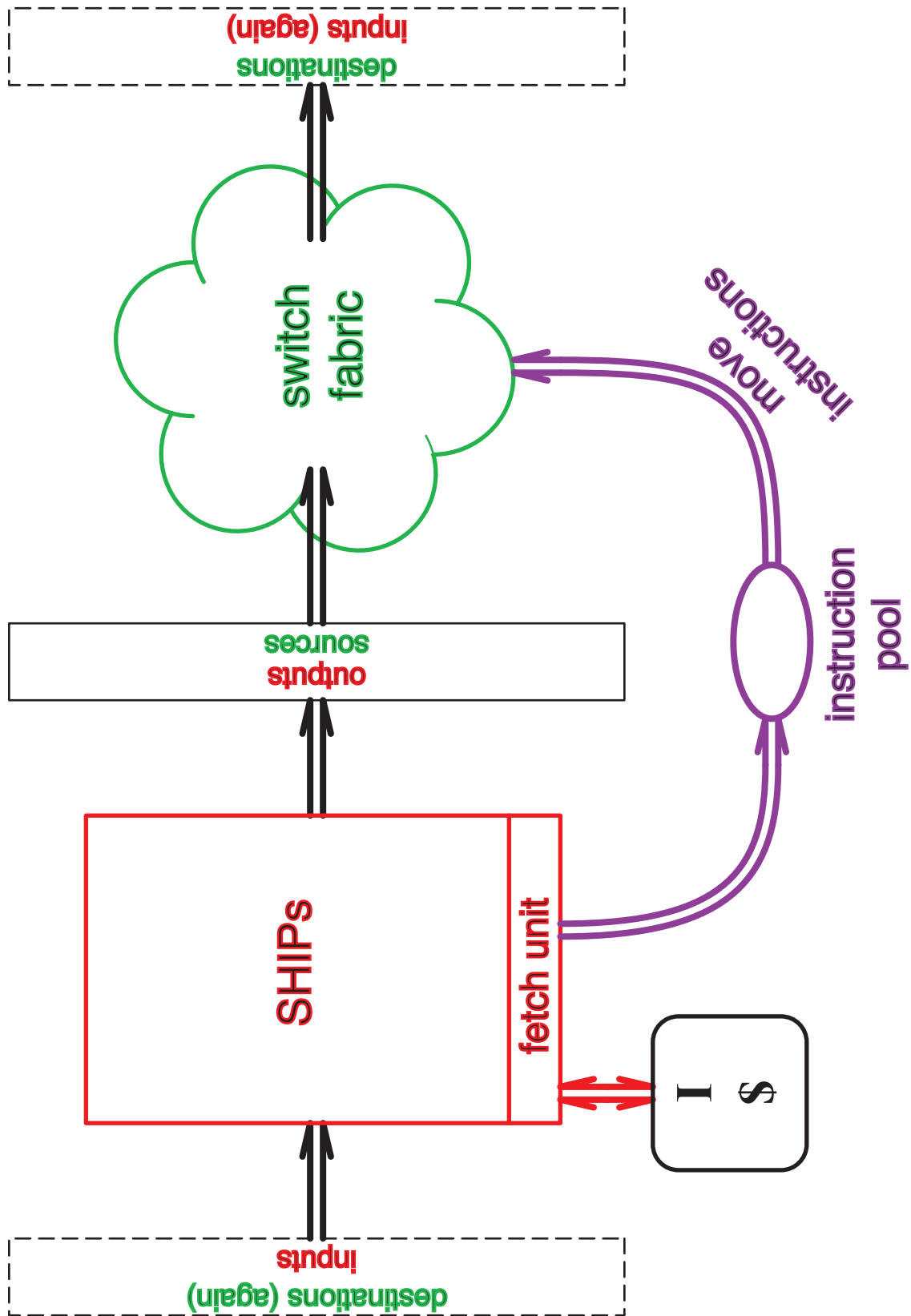


Figure 1: An Abstract View of FLEET

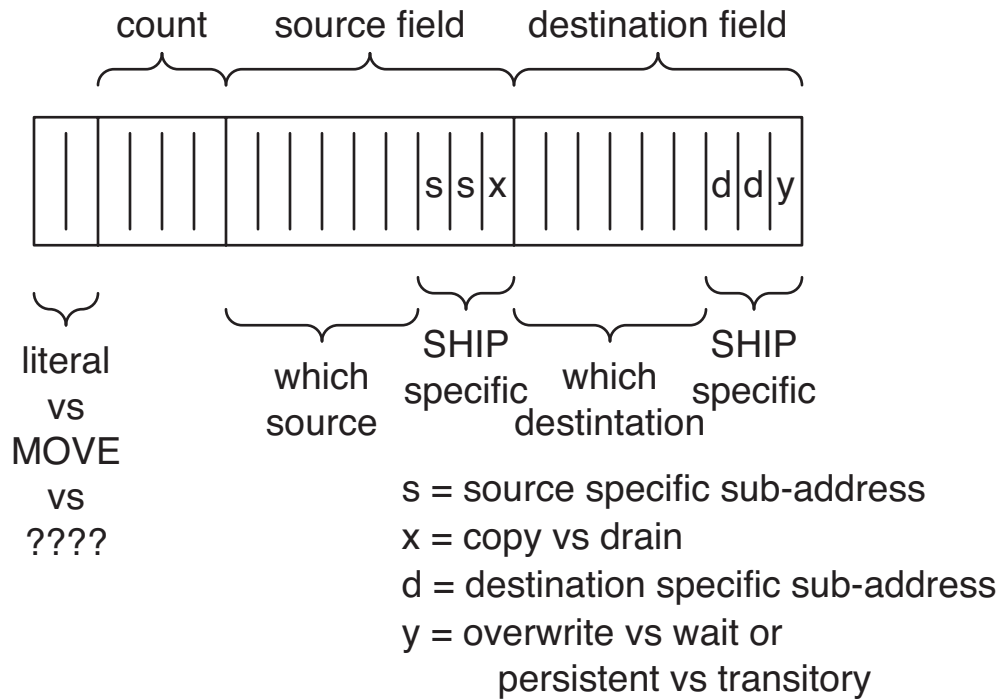


Figure 2: MOVE instruction format