

UC Berkeley Computer Science

Subject: FLEET – A One-Instruction Computer
Date: January 17, 2006
From: Ivan Sutherland, Adam Megacz and Igor Benko
UCIES #2007-is44

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
UCIES# 2006-is30: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2006
UCIES# 2006-is31: Some SHIPs, Adam Megacz and Ivan Sutherland, 24 August 2006
UCAM# 2006-am13: Unified Boxes, Adam Megacz, 2 January 2007

SUMMARY

This memo describes the major features of FLEET as we see it at the start of 2007. The general plan for FLEET remains unchanged. Its instructions deal with communication rather than data operations. Where data goes determines which calculations get done. For example, the program forms sums by sending data to an adder SHIP, stores data values in memory by sending them to a memory write SHIP, and so on.

FLEET remains entirely asynchronous and highly concurrent at the instruction level. If sequencing is necessary, the program must provide for it. To help with sequencing FLEET offers a data-less token type as well as the more usual data types.

During the last four months we have factored the communication actions out of FLEET's SHIPs into elements we call an "InBox" and an "OutBox," or a "BenkoBox" to mean either. An InBox on each SHIP input couples a switch fabric destination to that input. An OutBox on each SHIP output couples that output to a switch fabric source. InBoxes and OutBoxes interpret instructions that control how data pass from one SHIP through the switch fabric to another SHIP. The standardized design of InBoxes and OutBoxes not only simplifies the design of SHIPs, but also standardizes the programmer's view of communication.

Each OutBox interprets instructions that control when data enter the switch fabric and where the switch fabric carries the data. Each InBox interprets instructions that control when and how to accept data from the switch fabric and whether and where to send a token to acknowledge data arrival. The InBox fills a long-standing gap in the FLEET architecture: the ability to do instructions when data emerge from the switch fabric.

Each FLEET instruction carries a small integer that specifies how many times to repeat the instruction. FLEET also permits "standing instructions" that repeat an unspecified number of times, continuing until replaced.. An arbiter in each BenkoBox provides for clean termination of standing instructions.

Students in Ivan's research class at UC Berkeley have written a variety of programs for FLEET. The programs are written in an assembly language designed by Adam Megacz and compiled on his compiler. Students have used a simple FLEET

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.

emulator to test their code. Igor Benko made an event-driven simulator that can provide performance information for FLEET programs.

BASICS – Figure 1

Figure 1, an abstract view of FLEET, has remained unchanged for a long time. Figure 1 is cylindrical; the dotted boxes at the left and right edges of the figure represent the same objects. A FLEET system consists of a number of simple asynchronous logical units called “SHIPs,” shown red in the figure, that communicate through an asynchronous “switch fabric,” shown green in the figure. Each SHIP, such as an adder, may have as many data inputs and outputs as its designer wishes. Each part of FLEET will wait until it has enough information to begin its task and will wait again until its output data, if any, move on.

FLEET is entirely concurrent. Instructions enter FLEET’s “instruction pool,” shown purple in the figure, from an “instruction cache,” shown in the figure as a black box labeled I\$. Instructions are said to “issue” when they enter the pool. One or more special SHIPs, called “fetch units,” issue instructions to the instruction pool. Once in the instruction pool an instruction remains active until it gets data on which to act. Instructions are said to “execute” each time they act on data, and instructions are said to “complete” when there is nothing further for them to do. An instruction may execute several times before it completes, or even, in the case of standing instructions, remain active indefinitely.

Each instruction carries a “count,” a small integer indicating how many times it will repeat. One particular value of the count field designates an instruction as a “standing instruction” that repeats indefinitely until replaced. Repeated and standing instructions let FLEET move large amounts of data with relatively few instructions.

Instructions control only the flow of data. A SHIP can act only in accord with the data it gets. In contrast to conventional computers, FLEET’s instructions avoid telling SHIPs how to treat data. Thus, a SHIP that can do more than one task, the arithmetic SHIP for example, must have a “command” input from the switch fabric to tell it how to treat its data.

InBox and OutBox – Figure 2

BenkoBoxes interpret instructions that come from the instruction horn. As shown in Figure 2, each OutBox and each InBox is both a source and a destination on the switch fabric as indicated by the arrows. An OutBox can send a data value into the switch fabric with routing information that tells the switch fabric where to deliver the value. An OutBox can also wait to receive a token before taking action. An InBox can receive a data value from the switch fabric. An InBox can also send a token into the switch fabric with routing information that tells the switch fabric where to deliver the token.

An “OutBox” connects each SHIP output to the switch fabric, see Figure 2. An instruction delivered to an OutBox causes the OutBox to send data from the SHIP into the switch fabric and thence to the destination specified in the instruction. The OutBox will do nothing with an instruction until it gets data from the SHIP as well. If data from the SHIP reach an OutBox before it gets an instruction, the data will wait for an instruction to arrive.

An “InBox” connects each data destination of the switch fabric to an input of a SHIP, see Figure 2. An InBox also interprets instructions that tell it what to do when data arrive. The InBox will do nothing with an instruction until it gets data from the switch fabric. If the data from the switch fabric reach the InBox before it gets an instruction, the data will wait for an instruction to arrive.

FLEET's operation is asynchronous, and so instructions and data will wait as long as necessary before acting. Because the instruction horn is asynchronous, issued instructions can queue there until their InBox or OutBox is ready for them. The instruction horn and the instruction registers in the BenkoBoxes together form the "instruction pool" seen in Figure 1. Likewise the switch fabric is asynchronous. Data or tokens dispatched into the switch fabric can queue in the switch fabric until accepted at their destination. The programmer must take care to avoid deadlock from too early issue of instructions into the instruction horn or by premature dispatch of data or tokens into the switch fabric.

BenkoBoxes can use tokens to control the flow of data through the switch fabric. Each and every OutBox is not only a data source on the switch fabric, but also a token destination. A suitable MOVE instruction may wait in the OutBox not only for data from the SHIP, but also for a token from the switch fabric. Such a token can release data to flow through the switch fabric.

Similarly, each and every InBox is not only a data destination on the switch fabric, but also a token source that can send a token to a destination specified in its instruction. A suitable ACCEPT instruction can send a token into the switch fabric in response to each data value the InBox gets from the switch fabric. The program can use such tokens to indicate that a SHIP has received a previous input value and is ready for another.

FLOW-CONTROLLED CONNECTIONS – Figure 3

Both the data and token ports of the InBox and OutBox appear in Figure 3a and 3b. A "flow-controlled" channel between two SHIPs appears in Figure 3c. In such a flow-controlled connection, data flows from sender to receiver, but tokens flow from receiver to sender. Programs use the flow of tokens from an InBox to an OutBox to avoid congestion in the switch fabric. For each data value received, the InBox sends a token back to the OutBox to indicate that the OutBox may safely send another data value. This kind of flow-controlled connection permits a program to "wire up" the outputs of some SHIPs to the inputs of other SHIPs. A flow-controlled connection may also involve three SHIPs in a triangle or many SHIPs in a ring. Data and tokens flow through such flow-controlled connections at whatever speed the SHIPs can handle.

Factoring the flow-control mechanism out of the SHIPs themselves and into the InBoxes and OutBoxes provides three advantages. First, a standard InBox and OutBox design can be used everywhere. Second, SHIPs are free of the details of flow-control. Third, the programmer is free to use flow-controlled connections anywhere.

A program can use counting instructions to form a flow-controlled connection that lasts for a limited number of data elements. After the specified number of actions, the InBox and OutBox await their next tasks.

A flow-controlled connection of standing instructions is also useful. Two standing instructions, one at an InBox and one at an OutBox, can control orderly flow of many thousands of data items between SHIPs. Standing instructions in flow-controlled connections permit a program to "wire" outputs to inputs. Very high throughput is possible.

We've spent a lot of time learning how to "kill" standing instructions. Suppose, for example, that we want to send 10,000 values from A to B. A pair of standing instructions will move the data, but after 10,000 moves we must replace those standing instructions with other instructions suitable to the next task.

Any BenkoBox will replace a standing instruction with the next instruction it gets. An arbiter in each BenkoBox replaces the instruction at an opportune moment at the end of a unit action by the existing instruction. Thus to replace a standing instruction, the program just issues its replacement.

Programmers have found special SHIPs useful to help control how much data flows through a channel composed from standing instructions. One kind of helper SHIP counts passing tokens and diverts the Nth one to indicate that the channel has done as many operations as desired. The diverted token tells the program when to issue the fresh instructions that will replace the standing ones. This kind of helper SHIP is useful for moving a fixed but large number of data values. A different kind of helper SHIP examines passing data elements and diverts their flow when it detects a termination value. This kind of helper ship is essential when a special data value indicates the end of a string of data.

INSTRUCTIONS

A FLEET instruction consists of four parts. First, there is the address of the BenkoBox to which the instruction applies. This address is the instruction port address of the designated BenkoBox on the instruction horn. We used to call it the “source address” before we had InBoxes, but “source” has now become ambiguous. The instruction horn shown in Figure 2 decodes the instruction port address and delivers the instruction to the proper BenkoBox. Notice that a SHIP output has an instruction port address only by virtue of its connection to an OutBox. Notice also that every SHIP input also has an instruction port address by virtue of its connection to an InBox. Every BenkoBox must have an instruction port address because it can execute instructions. Moreover, each BenkoBox can both send and receive either tokens or data values from the switch fabric.

The second part of a FLEET instruction is a few bits that tell the InBox or OutBox how to behave. These bits specify which input signals must arrive before the instruction can execute and which inputs the instruction can ignore. These bits also specify which outputs the instruction must produce when it executes. For example, an instruction for an OutBox might send data values from its SHIP into the switch fabric without waiting for a token. On the other hand, an instruction for an OutBox might send such data only after getting a token. Yet another OutBox instruction might take the data from the SHIP only when a token arrives and discard the data instead of sending it on. The assembler will convert instructions that would send data to the “bit bucket” into such “accept and destroy” instructions to save needless traffic in the switch fabric. A complete set of the possible BenkoBox actions is described in a memo from Adam Megacz. [AM13].

The third part of a FLEET instruction is a destination port address. An OutBox will send data to the indicated destination, generally the data input port of an InBox, unless instructed simply to discard the data. An InBox will send its token, if appropriate, to the indicated destination, generally the token input port of an OutBox. A data element that arrives at a token destination such as an OutBox serves as a token; its data value is ignored. A token arriving at a data destination such as an InBox will enable the InBox to act with a synthetic data value, possibly zero. The details of how tokens convert into data values are not yet complete.

The fourth part of a FLEET instruction is a small count. An instruction with count N takes the very same action as N identical instructions in sequence. Counting instructions increase the amount of data moved per instruction. If a programmer desires a count larger than provided by the count field, the action can continue in the next instruction. One value of the count field designates the instruction as “standing.” Only standing instructions can be replaced; other instructions always run to completion.

INSTRUCTION FETCH

A special ship called a “fetch SHIP” moves instructions into the instruction pool. Figure 1 shows a fetch SHIP at the bottom of the rectangle representing the SHIPs. A part of such a fetch SHIP also appears at the bottom of Figure 2. To issue instructions more rapidly, a fetch SHIP may issue several instructions concurrently, or more than one fetch SHIP may issue instructions concurrently.

A collection of instructions in FLEET is called a “code bag.” We use the term “bag” rather than “block” to emphasize that the instructions in a code bag are concurrent. FLEET provides only one guarantee of sequence called the “source sequence guarantee.” Any instructions in a bag that go to the same BenkoBox will remain in sequence. In other words the BenkoBox that will interpret such instructions gets them in the sequence established by the programmer. Instructions for different BenkoBoxes are concurrent and may execute in any order, because each waits only as long as necessary for data on which to act. The program must establish any necessary sequencing.

An instruction fetch SHIP must get a “code bag descriptor” to tell it which instructions to fetch. The fetch SHIP has an input, complete with InBox, to receive code bag descriptors. Each code bag descriptor specifies where to find a bag of instructions and the size of the code bag. Given a code bag descriptor, the fetch SHIP is free to issue the instructions in the bag in any sequence it pleases provided it maintains the source sequence guarantee. In particular, a fetch SHIP may concurrently fetch several, or even all, of the instructions in a code bag.

A fetch SHIP often gets involved in data conditional actions. One attractive fetch SHIP design has two token inputs in addition to an input for a code bag descriptor. It requires both a code bag descriptor and one token before it will act. Where it receives the token determines whether it issues the designated bag of instructions or discards the code bag descriptor. Another attractive fetch SHIP design has separate places for two or more code bag descriptors. It selects which of the indicated code bags to issue on the basis of a data value it gets on a data input.

There is no program counter and no implicit instruction fetch and issue. Every code bag must contain instructions that fetch one or more successor code bags lest that thread of code bags terminate. Of course, a code bag may also instruct the fetch unit to issue more than one successor code bag for greater concurrency. It is the responsibility of the programmer to avoid conflict in code bags issued concurrently.

MEMORY READ AND WRITE – Figures 4 and 5

The class chose a memory read SHIP with address, stride and count, as shown in Figure 4. The memory read SHIP has separate inputs for a starting memory address, a stride to add to it after each memory access, and a number of words to read. The memory read SHIP reads the block of memory thus specified and delivers the values in sequence at its data output. The OutBox at the data output treats individual output words and may require a token input for each word it delivers. The memory read SHIP stops reading when the count runs out. It is then ready to take the next address, stride and count.

Likewise the class chose a memory write SHIP with similar inputs for address, stride and count, as shown in Figure 5. The memory write SHIP accepts data values at a data input and writes them in successive locations in the specified block of memory. The InBox for data can issue a token for each data element written. The memory write SHIP

stops writing when the count runs out. It then acknowledges the address, stride and count, causing each of those InBoxes to issue a token if it was instructed to do so.

One can imagine other types of memory SHIPs. Imagine, for example, a frame buffer memory SHIP connected to a display external to FLEET. The interface to such a frame buffer SHIP might make use of the two-dimensional nature of the field of pixels. Such a SHIP might have two address inputs, one for an X coordinate and one for a Y coordinate. It might include a two-dimensional stride to facilitate reading or writing rectangular blocks of pixels. It should also be able to read or write pixels by coordinate address.

An initial FLEET chip may have a small on-board cache memory. If so, it needs a way for programs to transfer data to and from the cache, namely a SHIP capable of transferring information between such a memory and main off-chip memory. The form of such a SHIP has yet to be defined.

ARITHMETIC AND LOGICAL SHIPS – Figure 6

The role of the ALU in an ordinary computer might best be divided in FLEET into several separate SHIPs. An arithmetic SHIP could provide for arithmetic but omit logical operations. Such a SHIP might add, subtract, take absolute magnitude, and so on. A shift SHIP could shift and rotate, and a logical SHIP could manipulate bits.

Because there is no place in FLEET's instructions for an arithmetic "OP CODE" to control the arithmetic SHIP's behavior, an arithmetic SHIP requires a "command" input. The arithmetic SHIP shown in Figure 6 has three data inputs: A, B and C. The command input tells it what to do with the values it gets at inputs A and B. Figure 6 shows a link input and output to handle carry for multiple precision, the output of comparisons, and selection of either the A or the B input as may be required to implement conditional expressions. The adder must get all four inputs before it can act and always produces two outputs. Recall that each InBox can repeat an input value to save sending it again, and each OutBox can discard values that are not needed. A good implementation will offer low latency for sending data from link out to link in.

The separate command input to the arithmetic SHIP offers many more control bits than are available in the OP codes of conventional computers. Thus FLEET's SHIPs can provide a very rich variety of data processing. For example, the arithmetic SHIP might select the larger of two inputs, or produce an output value only if the two inputs are equal. It might also serialize its two inputs by copying them in sequence to its output. What commands are available depends, of course, on the implementation.

A rotate SHIP should contain data paths to shift and rotate bits and bytes within the word. It might be useful for it to offer double-wide input and output for packing and unpacking bit fields that extend over word boundaries. A command input would establish which data paths in the rotate SHIP are active, selecting whether to rotate, shift, sign extend, and how far to shift or rotate.

A logic SHIP would perform bit-by-bit logical operations on three data inputs. Three rather than two data inputs could provide for concurrent masking and logic. Each bit position could select one of eight possible outputs depending on the eight combinations of three inputs in that bit position. Eight bits of the command word would suffice to specify a value for each combination of input bits.

We may want a bit reversal SHIP or possibly a SHIP that does a perfect shuffle of the bits. Such SHIPs need be little more than an InBox and OutBox connected internally by a pattern of data wires.

SPECIAL SHIPS

A set of first in first out (FIFO) registers in FLEET will be useful. Such a "FIFO file" can replace the register file common in conventional computers. Each FIFO register will store a few values, say up to eight, in sequence. There should be enough FIFOs, say 16, to accommodate the needs of inner program loops. Given the standard InBox and OutBox, a FIFO register SHIP is easy to design. Because a FIFO may store much data, a FIFO file may occupy a lot of chip area.

One use of the FIFO file is to store loop variables. Loops often increment some variables and decrement others by different constants. Such variables and appropriate constants could be placed in FIFOs, and the commands that say how to combine them can also be put into a FIFO. The program could use successive elements in three such FIFOs as input to the arithmetic SHIP. It could use a flow-controlled connection to update the loop variables en masse to mask the switch fabric latency.

We need a stack SHIP. The interface to a stack SHIP requires careful design to avoid confusion about concurrent push and pop. Moreover, should our stack be limited in depth, and if so what should happen when it is full? The stack SHIP cries out for careful design.

OTHER SHIPS

The "other" category catches SHIP types that are unknown in conventional computers. A first FLEET chip should include a direct input and a direct output SHIP. A direct input SHIP gives the program access to the values of several of the chip's input pins. A program can use such inputs to select between alternative actions. A direct output SHIP delivers the value of some internal bits to output pins on the chip. A direct output SHIP can provide a very simple interface for performance testing. Every time the program reaches a certain point it loads a new value into the direct output SHIP. That offers a simple way to observe and time program operations.

A random number SHIP might be useful for making test patterns. Generating a real random number on chip based on shot noise is possible but seems overkill. A shift register sequence, on the other hand, is nearly as useful and is repeatable. A real-time counter or an event counter may also prove useful.

Token counter SHIPs are useful. A counter SHIP has an input for a count value, N . Each time a token arrives on its token input it reduces by one the value of N . As long as $N > 0$ it steers arriving tokens to one of its two token outputs. If $N = 0$ or less, it steers subsequent tokens to a second output.

Token merge SHIPs are also useful. A token merge SHIP emits a token only when it receives inputs on both of two input ports. Duplicator SHIPs are likewise useful. A duplicator SHIP copies one input onto two outputs to let pipelines branch.

Several forms of SHIP to do merge sort have been proposed and are discussed elsewhere. A merge sort SHIP accepts two input values on different inputs and produces two outputs for each pair of inputs. One output is the smaller value and the other output is an associated code bag descriptor previously entered into the SHIP as a

constant. Pipeline connections to a memory write SHIP and two memory read SHIPs can merge streams of data values as fast as memory can deliver them. A quick sort SHIP separates a single input stream into two output streams in a similar way.

BIT CONFIGURATIONS – Figures 7, 8 & 9

For the first time we have enough good ideas about a FLEET architecture to contemplate building a demonstration chip. The UC Berkeley class has done many example programs to refine the InBox and OutBox designs. The flow-controlled communication methods seem sound. We have some simulators and emulators on which to continue programming experiments.

It will be good to start 2007 with a specific idea of word size and instruction format. Defining a set of choices for actual instruction formats will enable us to emulate something that could actually be built. It will also enable us to refine our programming work to learn whether the proposed instruction formats serve. We have made some choices that are illustrated in Figures 7, 8 and 9.

Figure 7 shows the 37 bit data word and one use of its bits as a code bag descriptor. We chose a 37 bit word as the smallest prime number larger than 32 bits. Making the internal word larger than 32 bits permits the design of SHIPs that deal not only with 32 bit values but also with markers to define data type if desired.

Only the Fetch SHIP interprets a code bag descriptor. Thus the format for the code bag descriptor depends on the particular implementation of the Fetch SHIP. The format shown is typical of what a fetch SHIP might use.

Other types of SHIPs may define other data formats. For example, a 32 bit arithmetic SHIP might define which of the 37 bits it will interpret as arithmetic values. A character string unpacking SHIP may define how it will allocate the 37 bits to characters.

Figure 8 offers four types of instructions distinguished by their first two bits shown in red. If the first two bits are both zero, the remaining 35 bits pass through the instruction horn to guide the actions of a BenkoBox.

If the first two bits are 01, the use of the remaining bits is undefined.

If the first two bits are 10, the low order 24 bits pass directly to the destination named in the eleven-bit destination name field. The remaining $37 - 24 = 13$ bits are sign extended. This format makes it possible to send a literal to any chosen destination. The program must compose literals longer than 24 bits, from multiple 24 bit parts.

If the first two bits are 11, the low order 24 bits, sign extended, are treated as an offset from some base value. The most likely candidate for the base value is the code bag address of the code bag in which the relative literal resides. The details of relative literals remain to be defined.

Figure 9 offers a possible view of how the data and instruction bits might appear in a VLSI layout of FLEET. The figure shows a total width of $48 + 35 = 83$ bits plus several control units. This layout remains tentative.

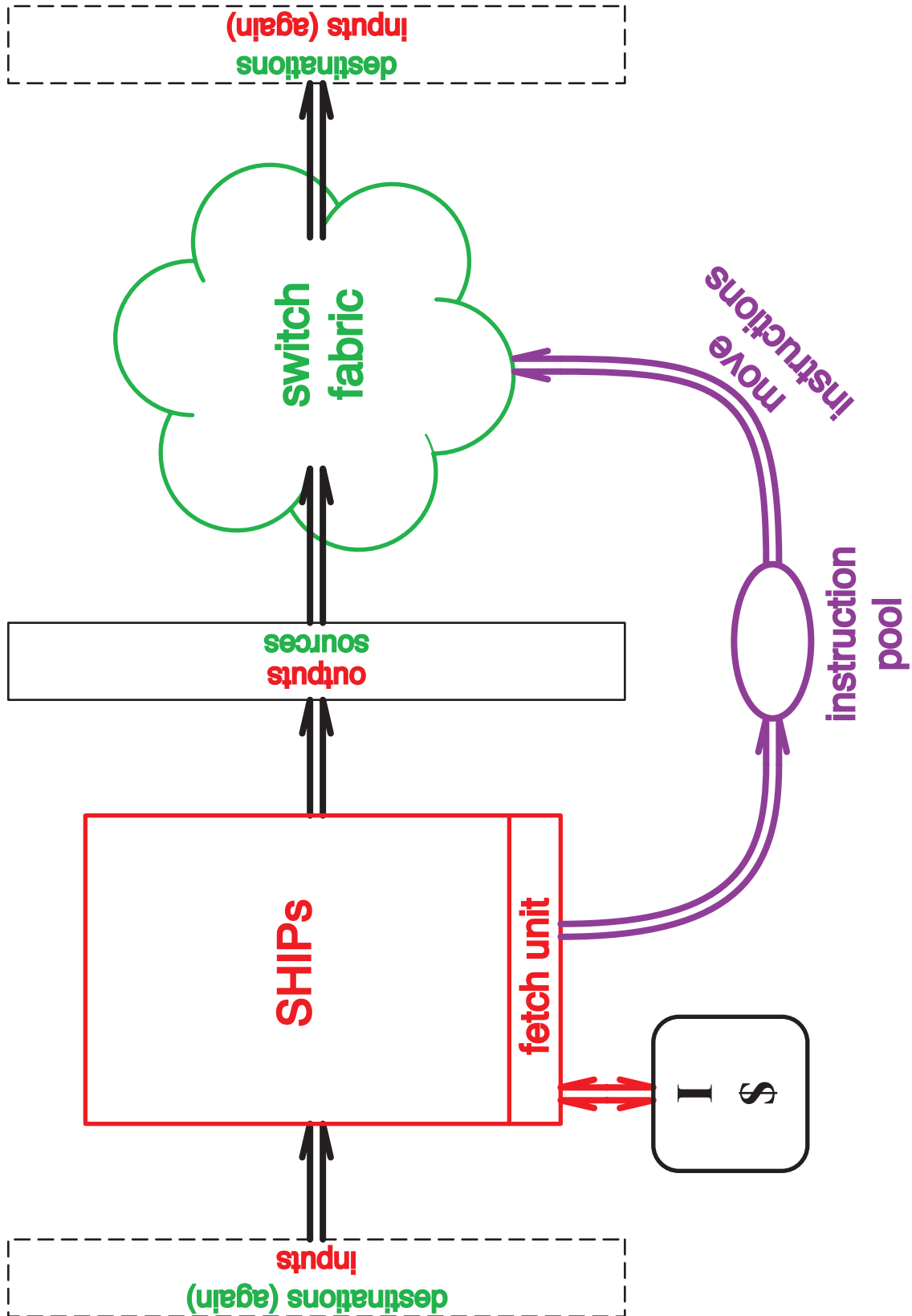


Figure 1: An Abstract View of FLEET

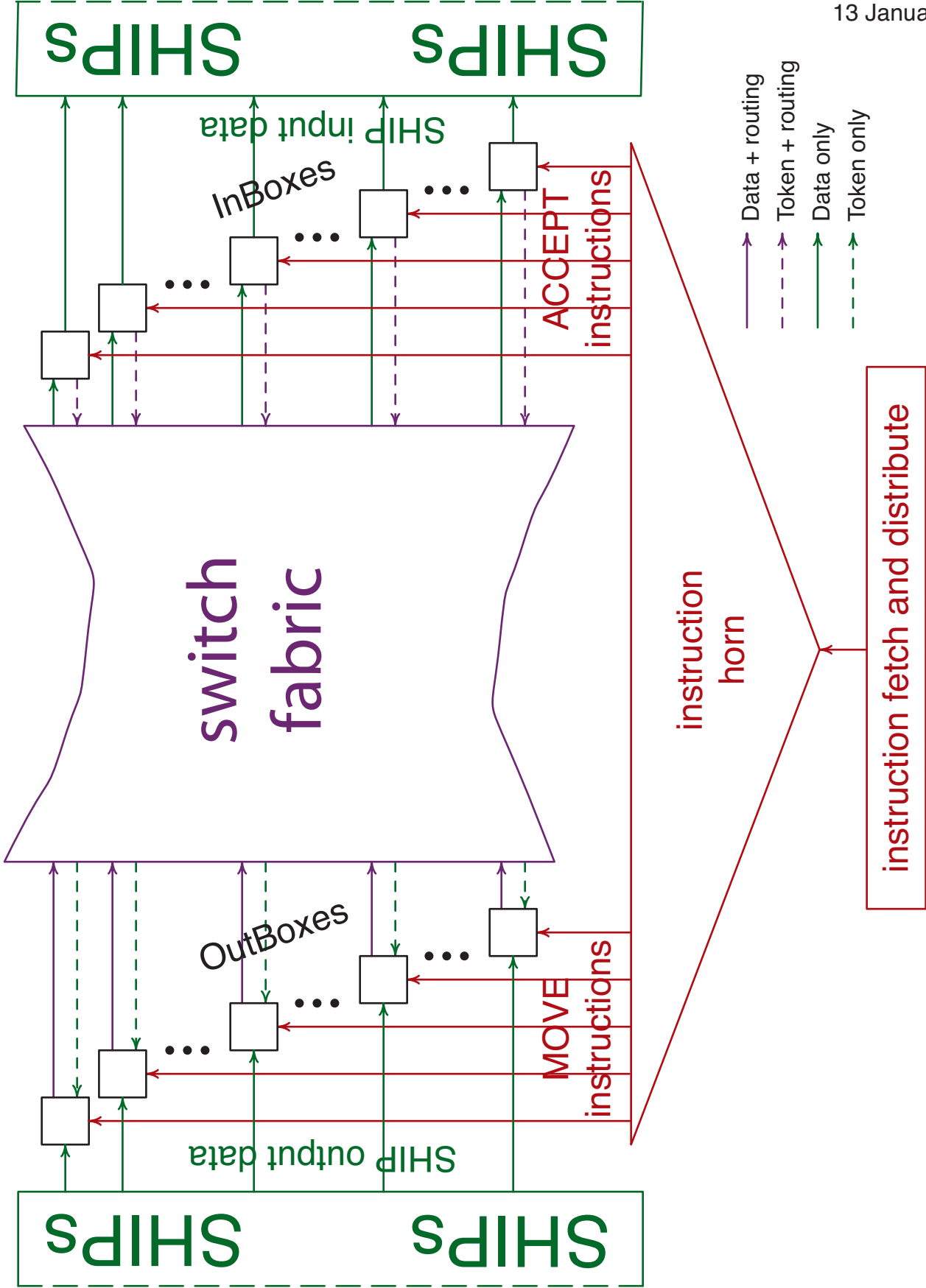
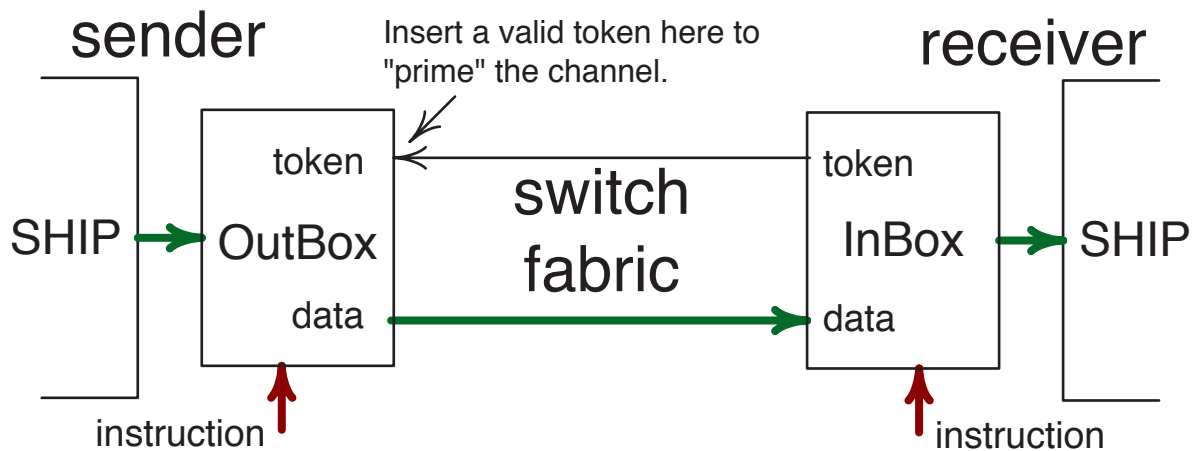
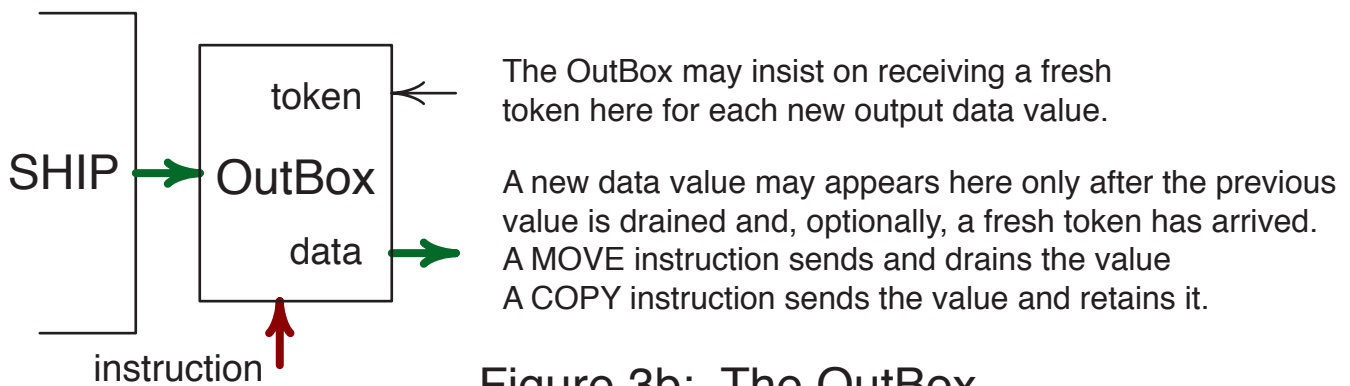
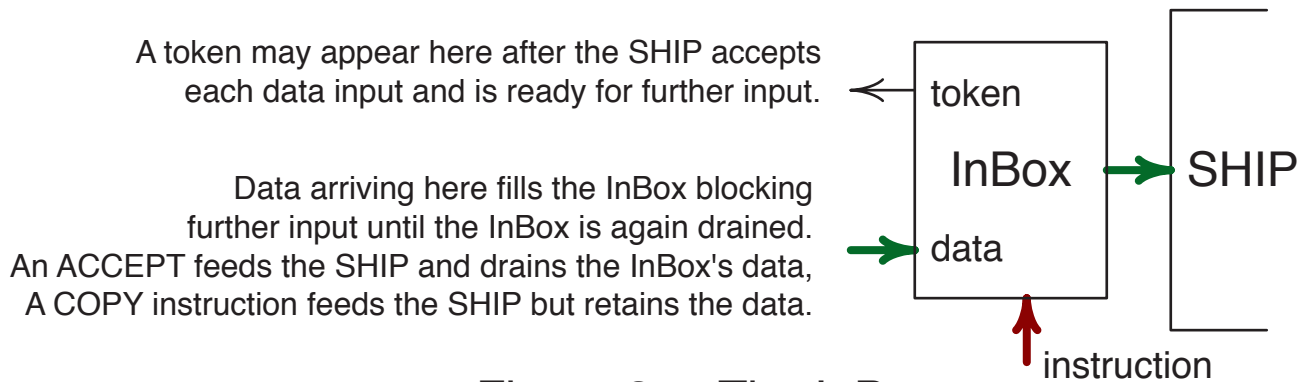
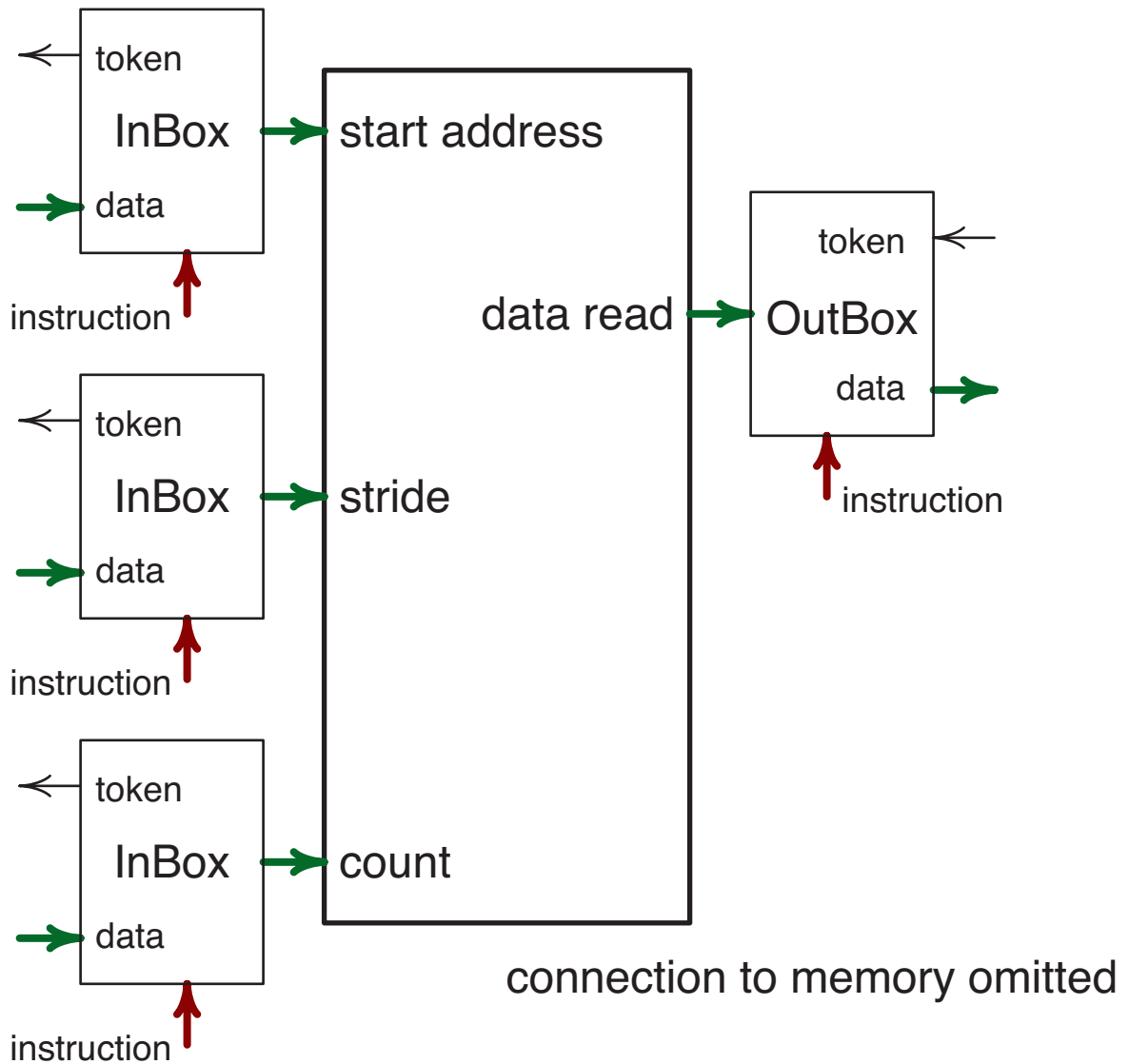


Figure 2: InBoxes and OutBoxes





This SHIP delivers successive data words to its OutBox and reads the next word when the OutBox takes this one.

OR

When the block read is done, tokens emerge from the InBoxes for address, stride and count.

Figure 4: Memory read SHIP

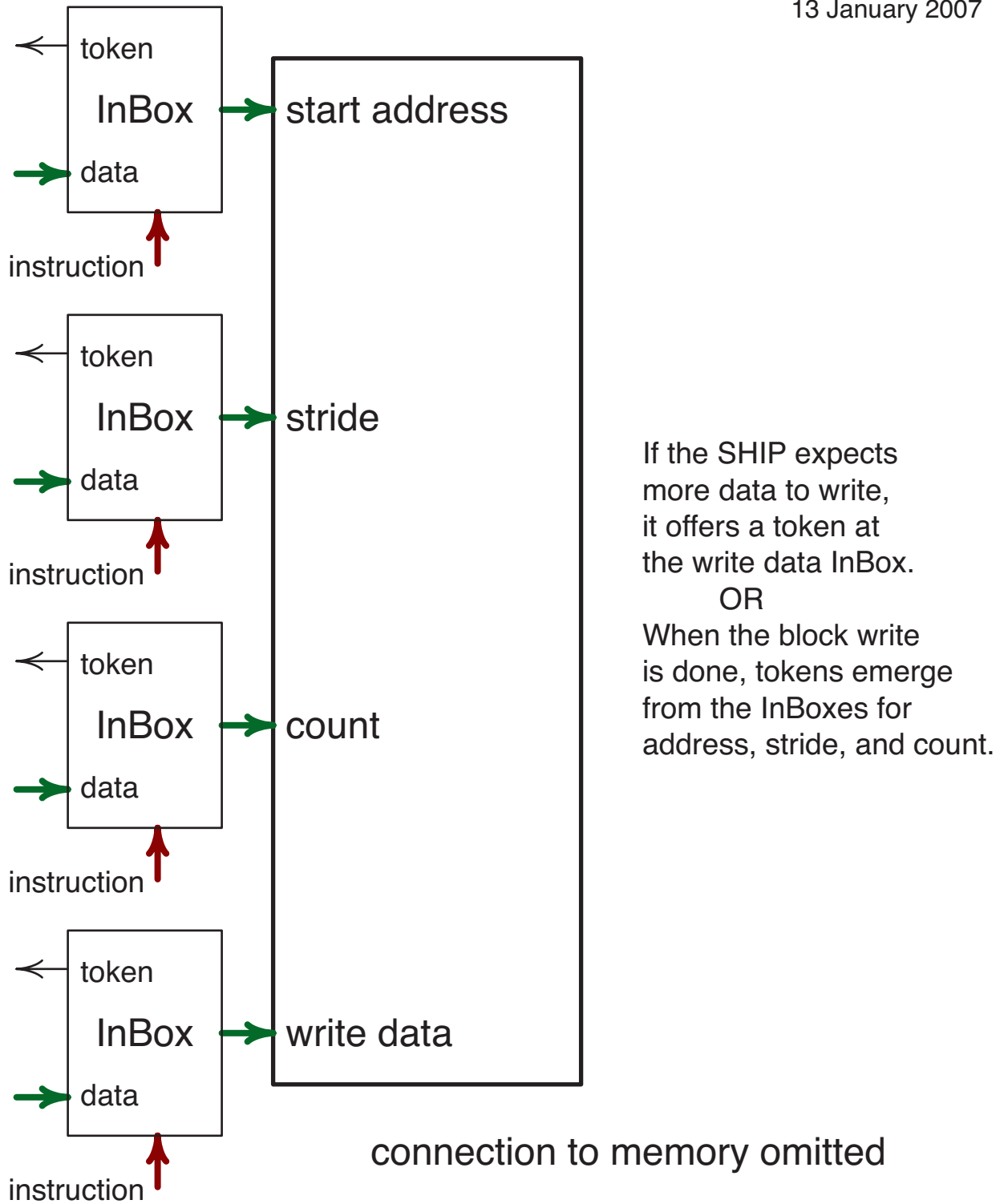


Figure 5: Memory write SHIP

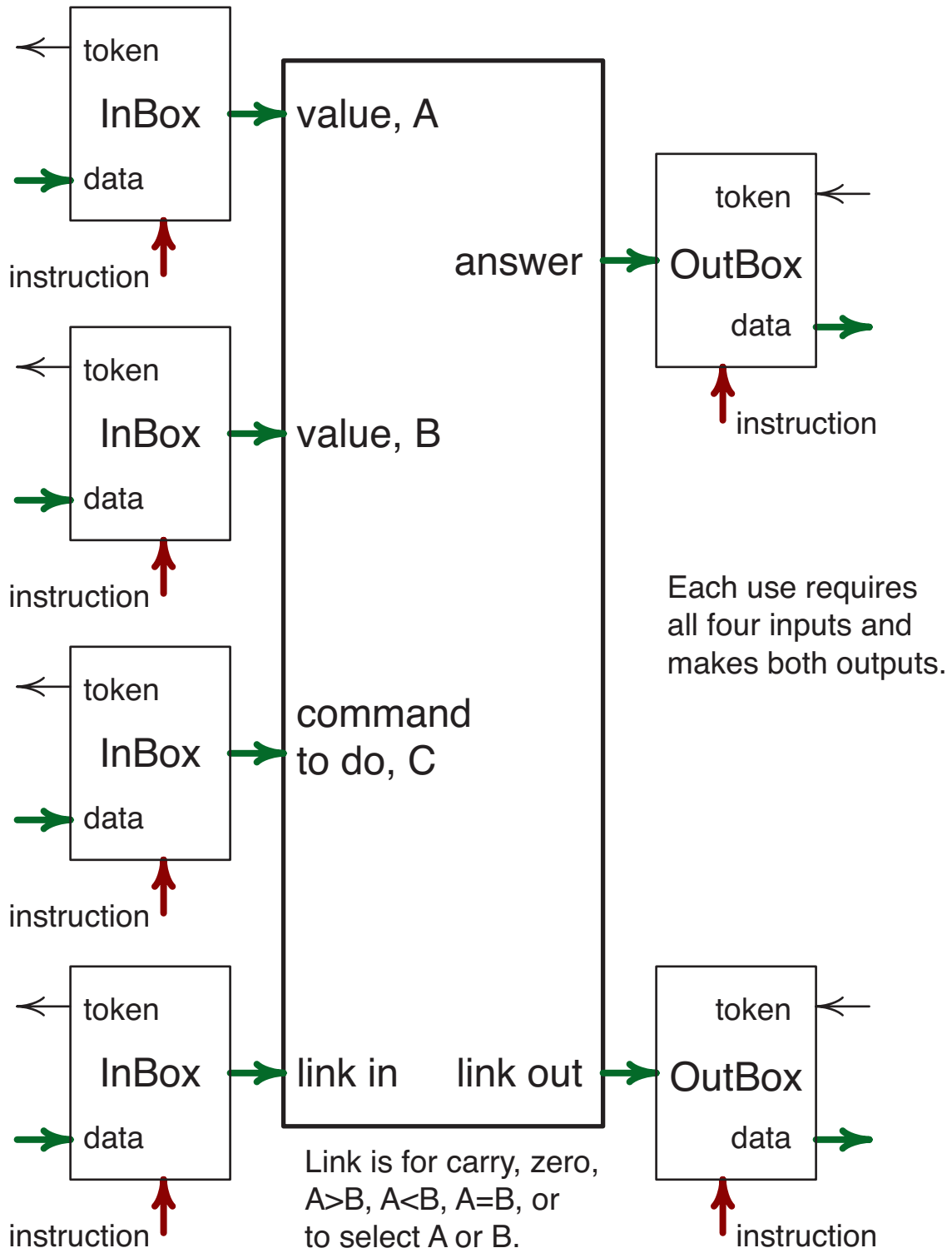


Figure 6: Arithmetic SHIP

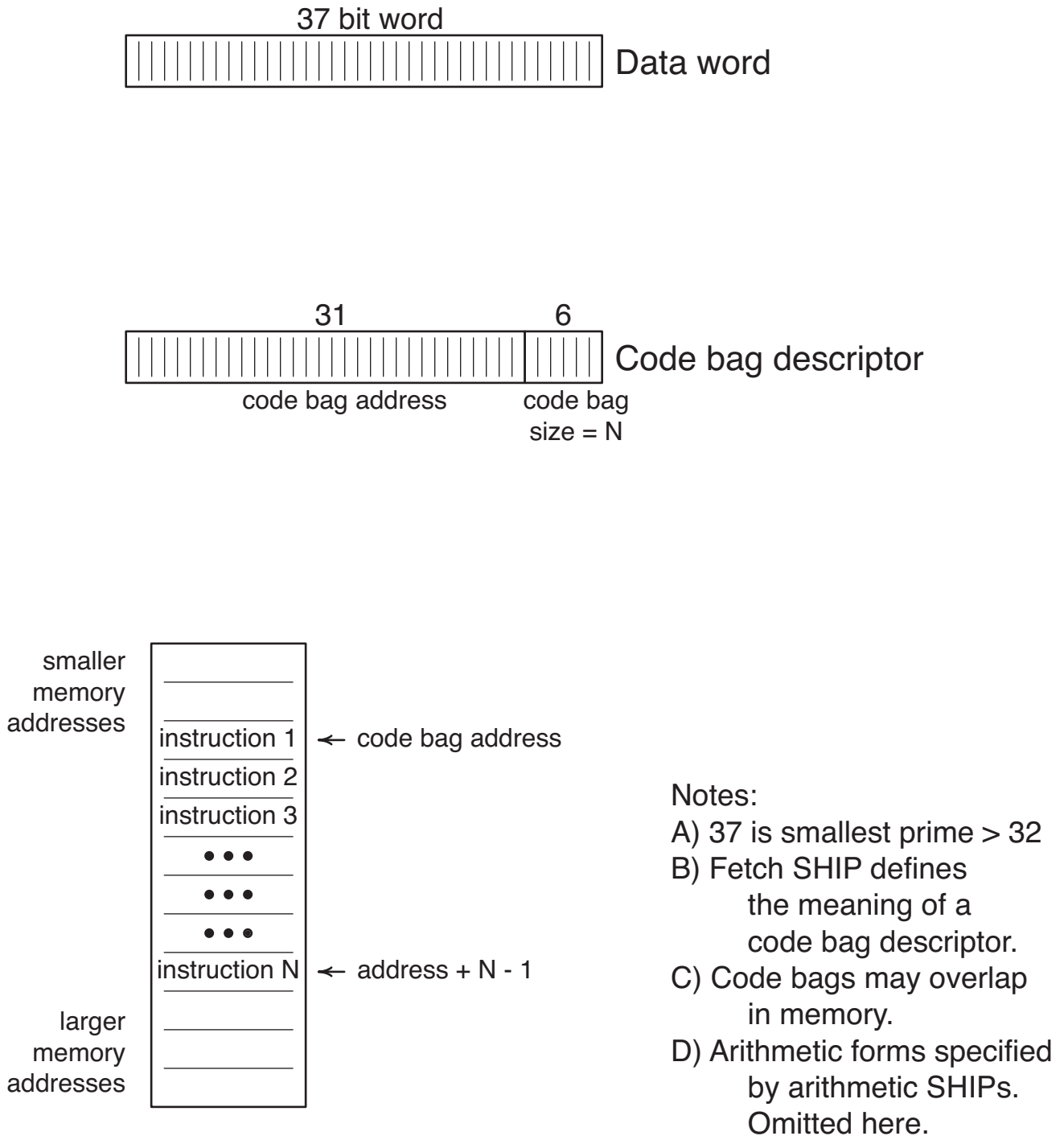
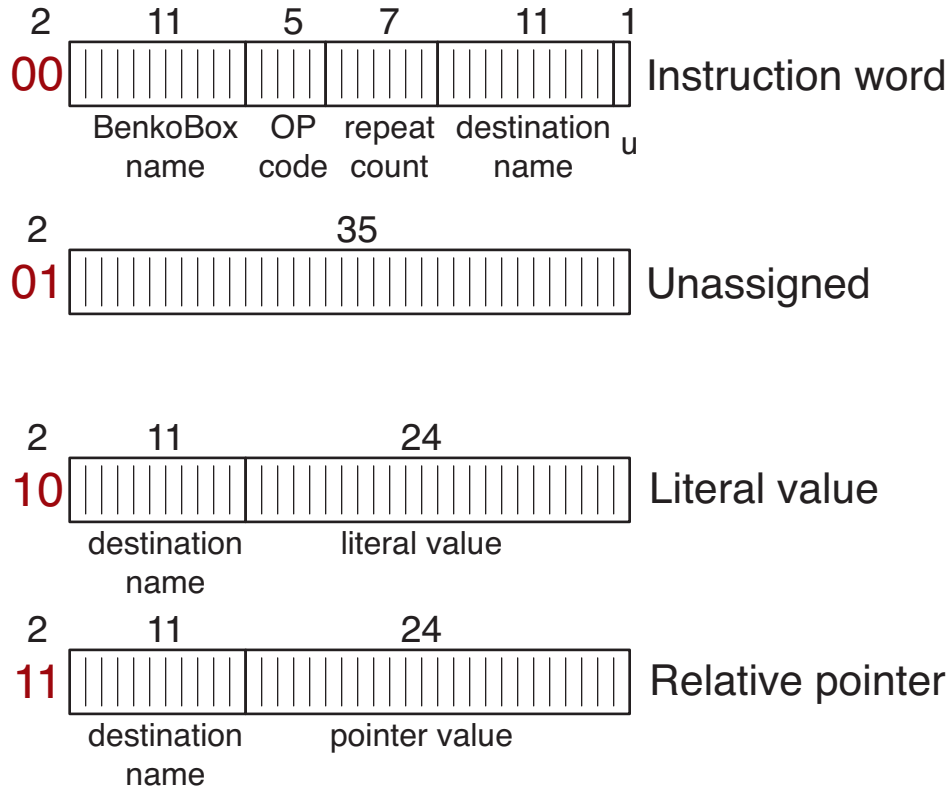


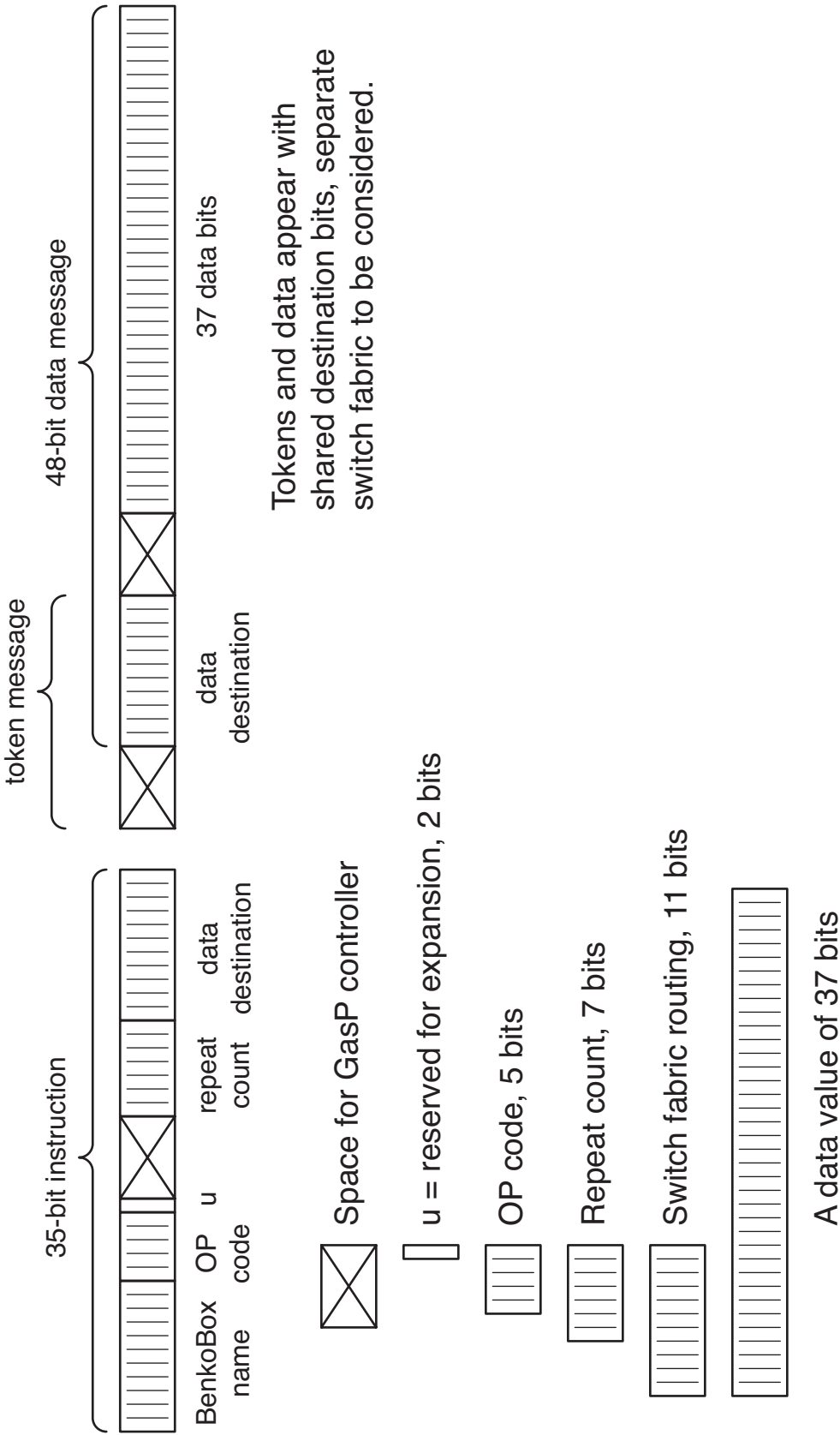
Figure 7: Data formats



Notes:

- A) First two bits (in red) define instruction type.
- B) 11 bit switch fabric destination name allows for routing as well as the fabric size.
- C) Instructions, literals and pointers intermingle.
- D) Literal value is in the least significant bits.
- E) Assemble full-word literals from two 24-bit literals.
- F) Relative pointers are relative to what?
(to be decided)

Figure 8: Instruction formats



Instructions are a total of 35 bits.
 Instructions flow through the "Instruction horn" to their BenkoBox.
 Data messages are a total of 48 bits.
 Data flow through the switch fabric to their destination.
 Token messages are a total of 11 bits.
 Tokens flow through the switch fabric to their destination.

Figure 9: A possible bit layout