

UC Berkeley Computer Science

Subject: Defining Some SHIPs
Date: August 24, 2005
From: Ivan Sutherland
UCIES #2005-is03

References:

UCIES# 2005-is02: FLEET – A One-Instruction Computer, Ivan Sutherland, 24 August 2005

PURPOSE

This memo offers initial designs for some SHIPs. I hope to make each design clear enough to allow us to use them in trial programs. After writing a first draft of this memo without interface definitions I decided to define some interface types from which to define the behavior of SHIPs. A SHIP may have as many of these interfaces as its designer thinks appropriate.

INTERFACES – Figures 1- 7

REGISTER – Figures 1 and 2 summarize the register interfaces. The register may be either full or empty. Delivering a data value or an Out of Band (OOB) value to the register's input interface fills the register with the delivered value and may or may not overwrite a previous value the register may have contained. The "overwrite" bit in the MOVE instructions destination field controls whether the arriving data destroys a previous value or whether the MOVE instruction waits until the register drains before delivering the new value.

A MOVE instruction whose source is the register's output must wait until the register is full before it can act. If the register is full or when it becomes full, such a MOVE instruction will transport the content of the register elsewhere, copying or draining the register according to the "drain" bit in the instruction's source field. The register becomes empty drained and remains full if copied. Draining the register does not necessarily clear the stored value, but renders that value inaccessible by declaring the register to be "empty." Some SHIPs may automatically replace a drained value with a fresh value.

Notice that concurrent overwrite and read of a register might produce erroneous data values. The reason is that some of the bits may be in the process of changing when the read occurs. To make a register safe from this kind of error requires arbitration to decide whether the register's next act is to deliver its existing value or overwrite it with a new value.

PIPELINE – Figures 3 and 4 summarize the pipeline interfaces. The pipeline input and output interfaces serve SHIPs that connect together to form pipelines. Each such interface has both a source and a destination aspect. Only the direction of data flow distinguishes the input interface from the output interface. Each of them responds to receiving something by delivering something. The input interface responds to receiving data by delivering a token. The output interface responds to receiving a token by delivering data. Because each responds only to an external stimulus they are called "passive" interfaces.

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.

The pipeline interfaces are embellished by their treatment of the Out of Band (OOB) values, particularly the one called LAST. The pipeline input interface responds to OOB data input in three ways. First, it delivers a LAST token in response, because LAST is the only OOB token form. Second, it converts any standing or counting MOVE instruction assigned to its token source by converting that MOVE to one time use; moving the LAST token is the last act of the standing or counting instruction. Third, it passes the OOB value into its SHIP. The SHIP may ignore the OOB data or use it for some purpose unique to that SHIP.

The pipeline output interface may deliver an OOB output data element, especially the one called LAST. If so, it converts any standing or counting MOVE instruction assigned to its data source to one time use. Moving the OOB data element is the last act of the standing or counting instruction. The pipeline output interface ignores receipt of an OOB token, returning to its passive state.

Figure 5 shows the pipeline output interface of a sender connected to the pipeline input interface of a receiver. Such a connection requires two standing MOVE instructions. One MOVE instruction sends the data from the source at the pipeline output interface of the sender to the destination at the pipeline input interface of the receiver. The other MOVE instruction sends the token in the other direction. Such a loop is self-timed because a new data element will move only after the token returns from the receiver.

Because both the pipeline input and the pipeline output interfaces are passive, one must initialize at least one of them to make data flow. This may be done by inserting a single token at the token destination of the sender's output interface. After receiving such an initializing token, the sender's output interface will deliver data elements from the sender as they appear. The sender's output interface will wait for a fresh token between outputs.

Given a pipeline connection between an output and an input interface, data elements will flow from interface to interface at a rate set by the responses of the SHIPs involved and the latency of the MOVE instructions. If one uses standing MOVE instructions, the two interfaces are, in effect, wired together. Such a "wired" connection will dissolve automatically after it passes an OOB data element. One-time or counting MOVE instructions can connect pipeline interfaces, but they must be replenished periodically.

MULTIPLE DATA INTERFACES – Figures 6 and 7 summarize the pipeline interfaces that handle multiple data elements. Each of them has several data inputs or outputs but only a single token input or output. Like the simple pipeline interfaces, actions on the inputs and outputs of these interfaces alternate. The input interface produces an output token only after it receives a fresh data element on each and every one of its data inputs, and then only if the SHIP is ready for a new batch of input data.

The multiple output interface produces a new set of data outputs only after all of its previous outputs have been drained, and it has received a fresh token at its token input, and suitable values are available to it from its SHIP. New outputs appear in concurrent batches, one on each data output. The program must discard unused data outputs.

These interfaces treat arrival of Out of Band (OOB) data as a termination signal for the entire interface. For the input interface, an OOB input value on any data input converts any standing or counting MOVE instruction assigned to the input interface token source to one time use. The standing instruction will move the resulting OOB token as its last act. For the output interface, an OOB value from the SHIP to any output will convert all standing or counting MOVE instructions attached to any output of the interface to one time use. Moving the value there will be its last act.

BUILT-IN CONSTANTS

A FLEET probably needs sources for the Boolean constants TRUE and FALSE and the small integers ZERO and ONE. It also needs a source of valid and the only form of OOB token, called LAST. The token source is intended for initializing pipelines. These fixed sources provide a very simple example for us to start with.

The SHIP for built in constants needs as many register output interfaces as there are constants, and no input interfaces. That ship appears to the switch fabric as a set of sources that always have data available. We can think of these constants as multiple outputs of a single SHIP or as separate outputs of separate SHIPs.

Unlike other source locations which distinguish MOVE instructions that “drain” the data from those that “copy” it, these sources ignore the “drain” bit in the source field of the MOVE instruction and always copy the data. Because they copy data that is always available, it is very dangerous to send counting or standing MOVE instructions to these sources. Standing or counting MOVE instructions are likely to saturate the switch fabric.

The source-field bits that might otherwise tell whether or not to “drain” the output value might here be used instead to select between TRUE and FALSE or to select between ZERO and ONE. At the location that delivers tokens that same bit might select the valid or the LAST token.

Do we need sources of OOB constants? I can’t think of a use for any but the OOB token,.

THE BIT BUCKET

A FLEET needs a place to deposit unwanted values. This SHIP has only a single register input and no outputs. All MOVE instructions targeted for the bit bucket overwrite any previous value sent there whether or not the instruction is explicitly marked for overwrite. The bit bucket destination is always empty and will always accept any data type sent to it. The address bits that might otherwise say whether to treat this value as persistent are meaningless.

Might it be useful to count the uses of the bit bucket? Such a count might be useful for debugging programs or for gathering statistics.

Note that MOVE instructions with the bit bucket as destination need not be executed. The switch fabric itself might recognize the bit bucket destination and destroy the data in flight to save switch fabric resources and energy.

LIGHTS

Here’s a though example. Consider connecting the bits of a simple register to output lights. There might be one light for each bit in the register and two to indicate whether the register is full or empty. It might be wise, though unnecessary, to extinguish all lights except the one marked “empty” when the register is empty. If the register was full of the ZERO data value all lights would be extinguished except the “full” light.

Like all registers, the master clear signal should render the register empty. The master clear signal need not, however, clear the data bits in the register because the actual value in an empty register is immaterial.

REAL TIME CLOCK

The real time clock SHIP is included here as another simple example. It has a register output interface with an ever-changing value of known counting rate. A counting rate in excess of 10 MHz will prove useful. Note that 32 bits can represent numbers up to about 4 billion. A 32 bit counter will repeat every 400 seconds or so if counting at 10 MHz. A 10 MHz real time clock will require more than 32 bits.

In an asynchronous system it is not easy to guarantee that the value read from such a clock will be meaningful. Suppose, for example, that we try to read the clock while it updates. Some bits may exhibit the new value and some the old value for an erroneous reading. A proper design must include arbitration to decide whether the clock's next action will be to increment or to deliver an output value.

SIMPLE REGISTER

A simple register SHIP has storage for one word of data and a way to remember whether its word is full or empty. A register can, for example, store a code bag descriptor for a SHIP to use if it finds an exceptional condition.

A register presents a register input and a register output interface. Some registers may be agnostic to data type; others may accept only specified data types, ignoring all other types. The data type of a register output will be whatever data type is stored there. The register differs from a FIFO of length one by omitting acknowledgement tokens at its input and output interfaces.

The "overwrite" bit of the destination address indicates whether or not new data will overwrite the existing value. The "drain" bit in the source address indicates whether or not the MOVE instruction copies the data value, leaving the register with its previous data or drains the value in the register, leaving it empty. In either case, a MOVE instruction that attempts to get data from the output of an empty register must wait for the register to fill.

A standing instruction at the output of a register must drain the register contents. The reason is that a standing instruction that copies the register contents would saturate the switch fabric by flooding it with repetitions of the stored value. A counting instruction at the output of a register should likewise drain the contents of the register, because such an instruction would otherwise produce many copies of the register content and might saturate the switch fabric.

RANDOM NUMBER GENERATOR

The random number generator deserves a pipeline output interface. Its output interface should include both a value source and a token destination. Before receiving a valid token at its output interface, the random number generator presents no output. After receipt of each valid token the random number generator will present the next random value at its data output source. Subsequent MOVE instructions may copy that data value, leaving it in place. Alternatively, a MOVE instruction that consumes the value will allow the random number generator to create a new value, but the new value will be available at the output only after receipt of a valid token at the output interface destination.

Notice that one may think of the random number generator in two ways. One can think of it as having a standard output interface. Alternatively, one may think of it as a register with input and output interfaces that converts a token into a random number. The pipeline connection is merely a convenient way to generate a stream of random numbers.

A standing MOVE instruction at the output interface will deliver the value as instructed, emptying the output interface. Each time the random number generator receives a fresh token it will produce a new random number.

It is not clear to me what the random number generator should do if it receives a LAST token. In keeping with the passive nature of its pipeline output interface, it should ignore the LAST token so as to return to its passive state with no available output. I am concerned, however, that this leaves no way for the random number generator to kill standing MOVE instructions at its output interface.

Perhaps it is better to think of the random number generator as having a pipeline output interface and a pipeline input interface that takes token inputs. That would give the random number generator three token connections, two destinations and one source, and one data source connection. In that view, receiving an OOB token at the input interface would cause the random number generator to produce a LAST data output that would, in turn, kill any standing or counting MOVE instructions assigned to its data output source.

How to initialize the random number generator remains an issue. A register with exactly the register input and output interfaces could hold the seed value.

FIFO – Figure 8

A FLEET should contain at least one, and probably many First In First Out registers often known as FIFOs such as shown in Figure 8. Each FIFO will have a capacity determined by its design and expressed as a number of words. Several FIFOs each of less than a dozen words will no doubt prove valuable. Many SHIPs that compute may also have FIFO semantics. For example, a tally SHIP that counts the number of bits in each word might be just like a FIFO except for its ability to transform data. Similarly, a memory read SHIP may accept several address inputs before delivering their memory values at its output interface.

A FIFO uses the standard pipeline input and output interfaces. Each data value received moves forward through the FIFO to appear at its output interface. The FIFO preserves the sequence of input values, presenting output values in the same sequence as they were received.

MOVE instructions that transport data away from the output interface may either copy or drain the output data values. Any such MOVE that drains the output data will reduce by one the number of data values stored in the FIFO, freeing the FIFO to advance all of its data. After a MOVE instruction that drains its output data value, the FIFO will deliver its next data value only following arrival of another valid token at its output interface token destination.

Some FIFOs may be data type agnostic. Such FIFOs must preserve the type of each input. Thus such FIFOs must not only be wide enough to hold the widest data type but also carry extra bits for the data type. Other FIFOs may accept and store only certain data types such a Boolean, character, etc. Such specialized FIFOs may require fewer transistors and consume less energy than more general FIFOs.

If a FIFO receives an OOB data input, its input interface will respond by emitting a LAST token and killing any standing or counting instructions. The FIFO must also pass the OOB data value through its internal registers so that the OOB value reaches the data output interface in its proper sequence. When the OOB data element reaches the output interface,

the output interface will respond accordingly, emitting the OOB data value and killing any standing or counting instructions.

Some FIFOs may also include a register for a code bag descriptor. Such a code bag descriptor provides a way for the FIFO to invoke some special code when it gets an OOB data input. The output of the register will appear only when the FIFO gets OOB data input. Recall that the input to a register may either overwrite its current content or wait for the register to drain before taking on the new value. The code bag register output for a FIFO would probably be given a standing MOVE instruction to deliver its output to the FETCH unit. Such a MOVE would probably drain the code bag descriptor register to leave the register empty.

VECTOR FIFO

A vector FIFO is similar to an ordinary FIFO except that it concatenates its several data input values before accepting them into its internal FIFO. After they pass through the FIFO, the concatenated groups appear at the FIFO's multiple outputs only after all of the previous set have been drained and a new token has arrived at the token destination of the FIFO's output interface.

Inputs that are marked as persistent data need not receive fresh data. They remain full with their preset value. Thus persistent inputs for some components can make an N component vector FIFO behave as a narrower vector FIFO. Of course the corresponding unused outputs must be drained, perhaps by standing MOVE instructions to the bit bucket.

A vector FIFO probably carries the code bag descriptor register previously described for the ordinary FIFO.

THREE-INPUT ADDER – Figure 9

The three-input adder combines an input interface like that of a three-component vector FIFO with a simple pipeline output interface. Indeed, such an adder might be embedded in a FIFO of any length, able to accept several sets of new inputs before delivering their sums. I believe that a useful FLEET will include several such adders.

I propose a three-input adder rather than a two-input adder because I recall a comment by Bill Wulf, a compiler writer, about how often compilers generate code to add triples rather than pairs of numbers. The three-input adder offers flexibility.

The “persistent” bit in the destination address for the adder can insert constant input values such as base addresses for arrays. A persistent ZERO on one input will change a three-input adder into a two-input adder. The special bits in the destination address might also negate an input, to make the adder subtract.

If the adder contains a code bag descriptor register, it will work as described for the FIFO. Of course, adding anything to an OOB input creates an OOB output. Overflow may also create OOB outputs which might be of the NAN type. Whether overflow produces a LAST output or merely one that is not a number remains a choice for the designer of the adder. I believe the not a number form is preferable.

ACCUMULATOR

An accumulator combines a pipeline input interface with a pipeline output interface. Values delivered to the input accumulate, to be delivered on demand at the output. Like other pipeline interfaces, the output value will appear only upon receipt of a valid token at the destination of the output interface. Notice, however, that the accumulator is interesting because it delivers one output value only after receiving many input values. A good design will carry extra precision internally to avoid unnecessary overflow in intermediate sums.

We must face the beginning and end questions, i.e. when to present the accumulated value and when to initialize the accumulator to zero. One choice for when to present output contemplates adding streams of data terminated by LAST markers. This kind of accumulator would present an output at its pipeline output interface only after receiving a LAST data element as input. Another choice of when to present output treats the adder more like the real time clock, making the most recent value available as output on demand whenever a valid token arrives at the token destination of the output interface.

Another choice involves when to reset the accumulator to zero. One choice is to reset the adder to zero whenever its output is drained. Another choice is to reset it just after it receives a LAST value at its input. This choice might permit the accumulator to start on a new string of inputs prior to delivering the accumulated value for the previous string. A third choice would use the "overwrite" bit in the destination field of the MOVE instruction to load a value into the accumulator rather than adding it to the existing value.

There is an issue about simultaneous input and output producing ambiguous results. Consider, for example, what would happen if a MOVE instruction attempts to read the output value just as a new value begins to increase the existing value. The accumulator SHIP must include arbitration to decide whether its next act is to deliver the existing output value or to do the next addition.

ADDRESS GENERATOR

The address generator is sometimes known as a "stride" SHIP. It provides an interesting contrast to the accumulator because it produces many output values for each set of inputs. It can use FIFO interfaces both for input and for output.

The input interface accept a base address, an initial offset, a stride, a count, and a limit address. Once it has accepted a valid set of input data it responds with a valid acknowledge token. If any of the input data are invalid or unusable, it responds with an invalid or LAST token.

Once it has a valid data set, the stride SHIP will generate a new address output every time it gets a valid token at its output interface. Its fist such output is the sum of the base address and the initial offset. Thereafter the outputs are separated by the stride until termination.

Termination occurs upon the first of three conditions: A) when the count is exhausted, or B) when the address about to be issued exceeds the limit value, or C) when a LAST token arrives at the token destination of the output interface. For cases A and B, the stride SHIP issues an OOB address pointer, probably LAST, as its output, and cancels any standing MOVE instructions at its output. For termination condition C it issues no output.

Any termination condition resets the stride SHIP, consuming its input values. The stride unit is then idle until it gets a new set of input data. Of course, some input values may have been made persistent, in which case they carry over from one data set to the next.

MEMORY READ

A modern memory system can support several concurrent outstanding memory accesses. Because of this it is important for the memory system to remember what to do with the data that returns after each read operation. In many existing computers the memory system remembers the name of the destination register in the register file. Thus as data returns, perhaps even out of order, it appears in the appropriate place in the register file.

A similar approach in FLEET would require the memory system to remember the destination SHIP for each memory read. Although this might have some advantages, it is the first SHIP, other than the FETCH ship, than needs to deal with separate MOVE instructions. It is the first place where the destination part of an instruction must be treated as data. Heretofore there was no form of "latent MOVE" instruction. Moreover, it prevents using the memory read process as a stage in a pipeline. How would acknowledge tokens return to the output interface of such a memory system?

Instead, let us treat the memory read SHIP as part of a pipeline with an input and an output pipeline interface. Because there may be several users of the memory system, let us provide several such pairs of interfaces, say four of them, with different addresses on the switch fabric. We can call each pair a "channel" and assign different source and destination addresses to each channel. The memory system will keep track of which interface requested a particular access and return the resulting data to the corresponding output interface. This approach allows us to insist that the memory system preserve sequence in each of its channels, but allow it complete freedom between channels. Thus the memory system appears to be four separate channels, each with FIFO semantics. Each channel is, in effect, a FIFO that converts memory pointers to data values.

The memory system may generate exceptions. One common exception happens when the memory system is given an OOB address, probably LAST. Its behavior should be just like that of the FIFO described above. That is, it immediately returns the OOB token, LAST, at its input interface and it kills any standing or counting MOVE instruction at its input interface. Because it is a FIFO, it must remember the sequence in which it received the OOB address and return an OOB output at the corresponding position in the sequence of its outputs. The OOB output will, in turn, kill any standing or counting move instructions at its output interface.

The memory system may also generate exceptions on its own accord. For example it may strike a parity error, or be asked to find a value outside its address range. In these cases it should send a code bag descriptor to the fetch SHIP from its exception descriptor register. It may be that each channel requires a separate exception descriptor register to preserve the appearance of four separate memory read SHIPs.

MEMORY WRITE

In most computers memory write is treated as "fire and forget." Most memory systems fail to return a response saying that they have, in fact, completed the desired memory write function, or at least accepted responsibility for its safe completion.

I want FLEET to do better. I want to treat the memory write function as a pipeline just like the memory read process. For memory write, however, the input interface is a vector input interface with two components, a memory pointer and a piece of data. The memory write output interface is also a pipeline interface, but requires only a token output instead of a data value.

The memory write input interface will deliver its acknowledgement token whenever it receives an input pair, but only after it has room for more input. This choice makes it possible to construct a pipeline input to a memory write interface that will run at a rate set by the memory write SHIP's ability to accept data. If the memory write system has an internal pipeline, it may be able to accept several inputs in rapid succession before delaying the return of a token at its input interface.

The memory write unit will deliver a token at its output interface only when it has accepted responsibility for a write operation and has received a valid token at the token destination of its output interface. Because the output interface is "passive" one must prime it with an initial token before the stream of output tokens that indicate successful memory writes will appear.

Like the memory read interface, we probably need several, say four, memory write channels. Each of these channels must have an input and an output interface as described. Moreover, while these channels may compete for use of a single memory, the memory write system must keep track of its channels separately in order to deliver the "success" tokens to the proper channel.

If a memory write channel gets an OOB address or data input, it will respond in two ways. First, it will deliver an OOB token at its input interface instead of a valid token, and then kill any standing or counting MOVE instruction at its input interface. Second it will, in turn, deliver an OOB token at its output interface instead of the success token and then kill any standing or counting MOVE instruction at its output interface.

The memory write unit may generate exceptions to otherwise valid data inputs. For example it might be asked to write at a location outside its address range, or it might discover some internal fault. In these cases it should deliver an invalid token at its output interface and deliver a code bag descriptor to the fetch SHIP.

A SHIP FOR SORTING – Figures 10 and 11

For the final example SHIP in this paper I will describe a SHIP that can merge sorted strings of characters. The sorting SHIP chooses the "earlier," in sorting sequence, of two input characters for delivery to its output. It is not enough merely to deliver the proper character; the SHIP must also indicate which choice it made so that the chosen character can be replaced with the next element from its string.

Although the ship could indicate its choice with a boolean output, I prefer to let it automatically invoke some code appropriate to replenishing the chosen character. In order to invoke code, the sorting SHIP will store two code bag descriptors, one for each of its input strings. We can think of each the two inputs to the sorting SHIP as a two-component vector, one component being the character and the other the code bag descriptor appropriate for replenishing it. The sorting SHIP likewise has a two component output vector, one component being the chosen character and the other being the appropriate code bag descriptor.

Selection of a particular input character keys off several actions. First, whichever interface was chosen emits an acknowledgement token. Second, the output pipeline interface emits the chosen character if it has received a token at its output interface token destination. Third, the code bag descriptor corresponding to the chosen character emerges from the code bag token output.

Receipt of a LAST input character on an input channel will result in a LAST acknowledgement token on that channel, and will kill any standing or counting MOVE instruction at that input interface. The other input interface will not be affected.

Comparison of a valid and a LAST character will always choose the valid character. Comparison of two LAST characters will produce a LAST character as output. The LAST output value will, as usual, kill any standing or counting MOVE instructions at the output interface. These choices provide for automatic termination of data streams terminated by LAST characters.

We can connect such a SHIP in a pipeline with other pipeline elements as shown in Figure 11. Before the sorting SHIP we put two address generators and a pair of memory read channels to fetch the two lists. After the sorting SHIP we put an address generator and a memory write unit. The combined pipeline may be "wired" by suitable use of standing instructions. It will then sort lists and automatically dissolve its own wiring as the lists terminate.

If one prefers, one can close the input loops via the instruction fetch unit. Because an appropriate code bag descriptor emerges with each character, a standing instruction can send it to the fetch unit. When it arrives it can dump a code bag into the instruction pool appropriate to replacing the input character that was consumed as output.

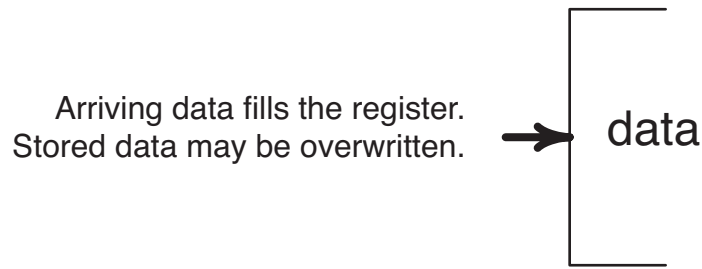


Figure 1: Register input interface

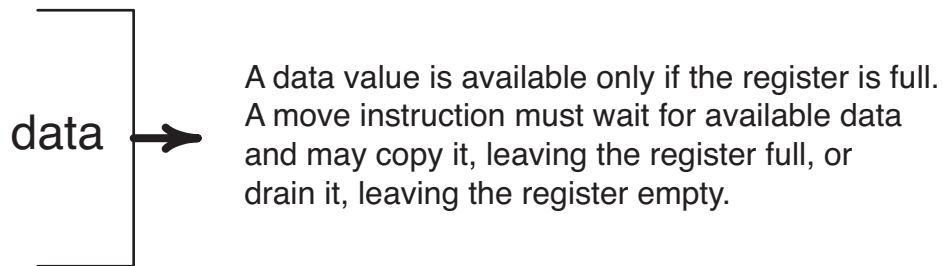


Figure 2: Register output interface

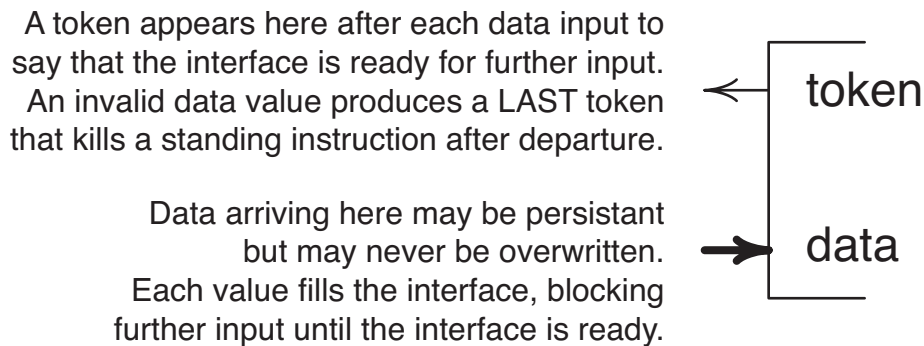


Figure 3: Pipeline input interface

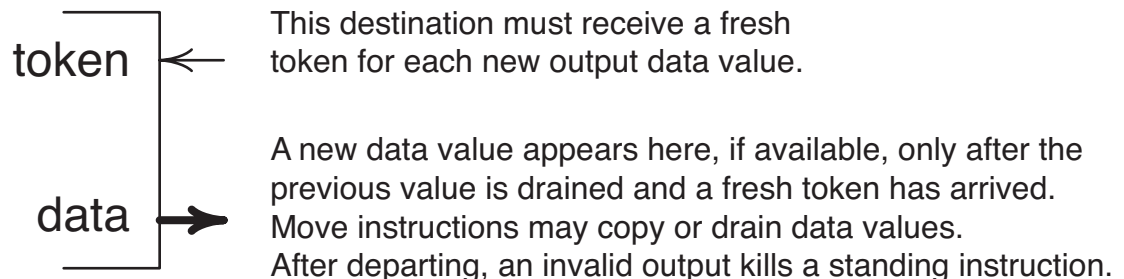


Figure 4: Pipeline output interface

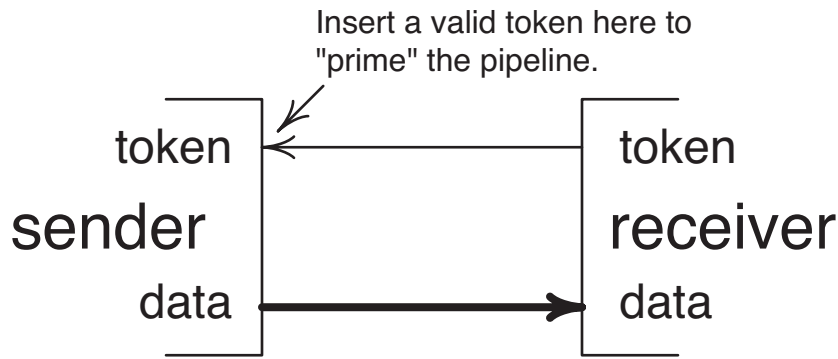


Figure 5: A pipeline connection

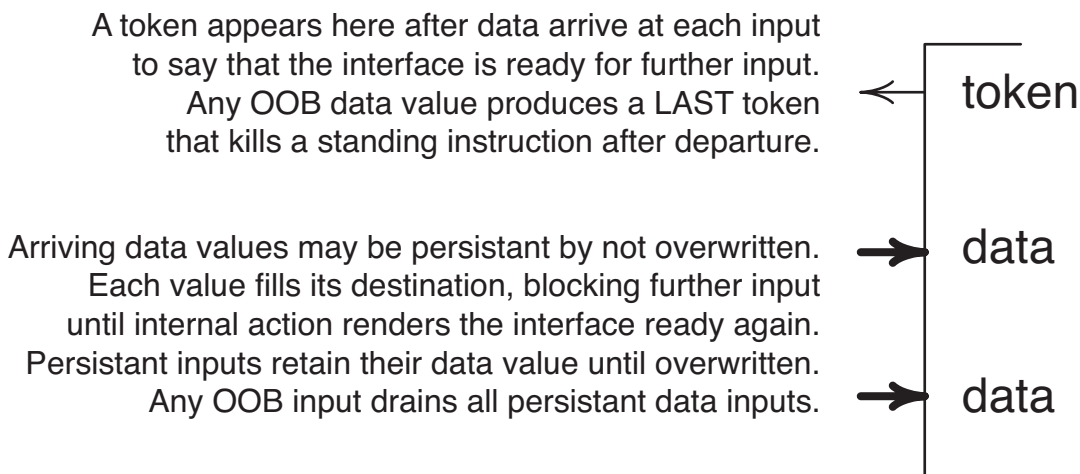


Figure 6: Multiple input pipeline interface

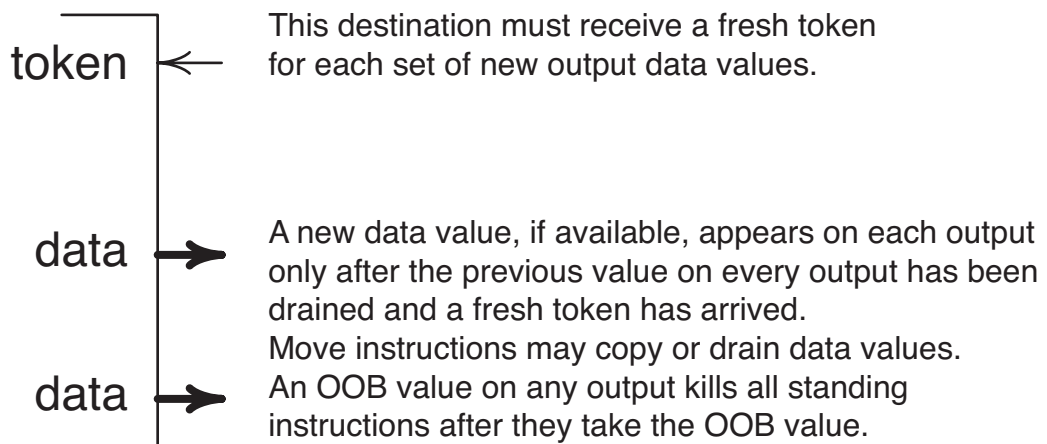


Figure 7: Multiple output pipeline interface

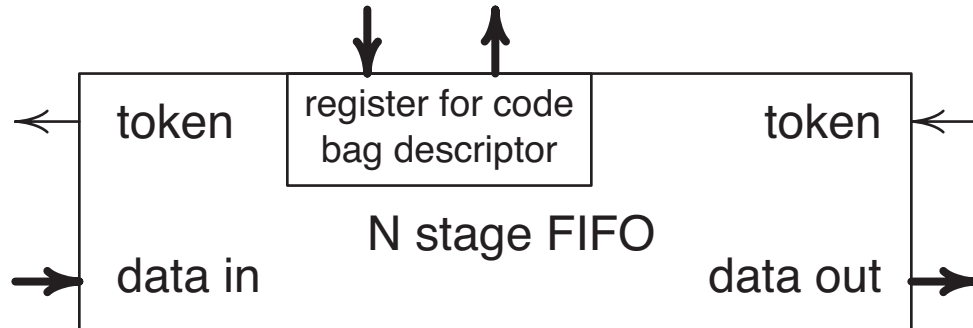


Figure 8: A FIFO SHIP

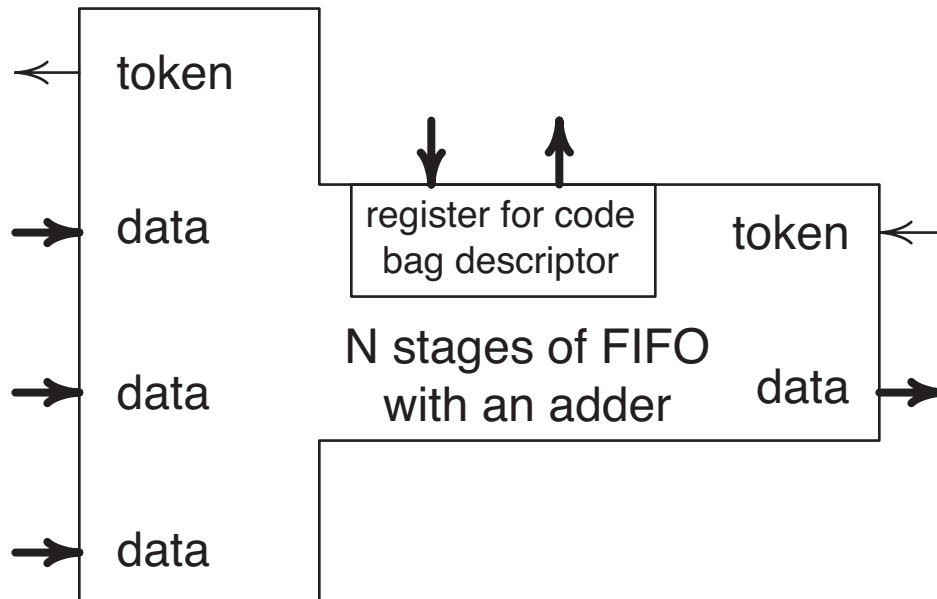


Figure 9: A Three-input Adder SHIP

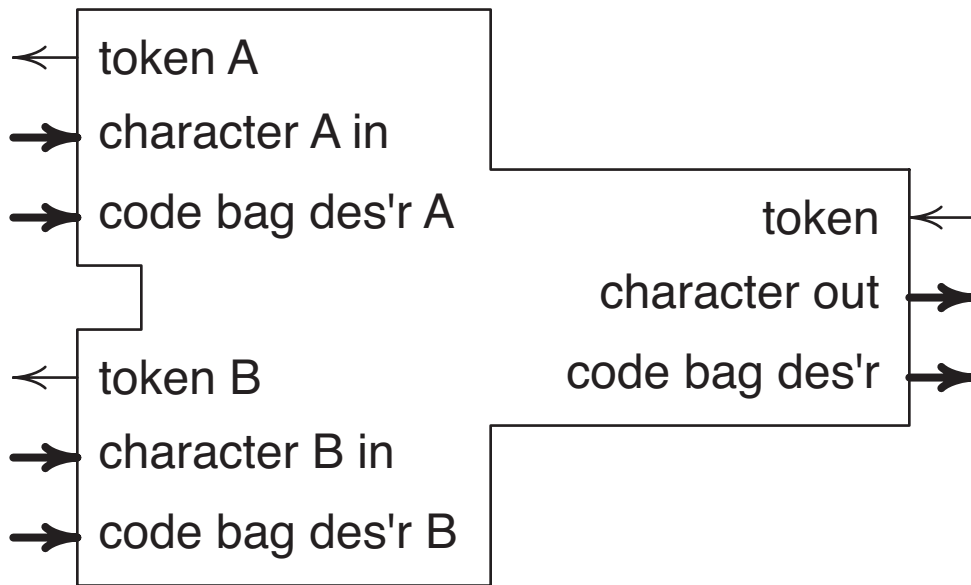


Figure 10: A sorting SHIP

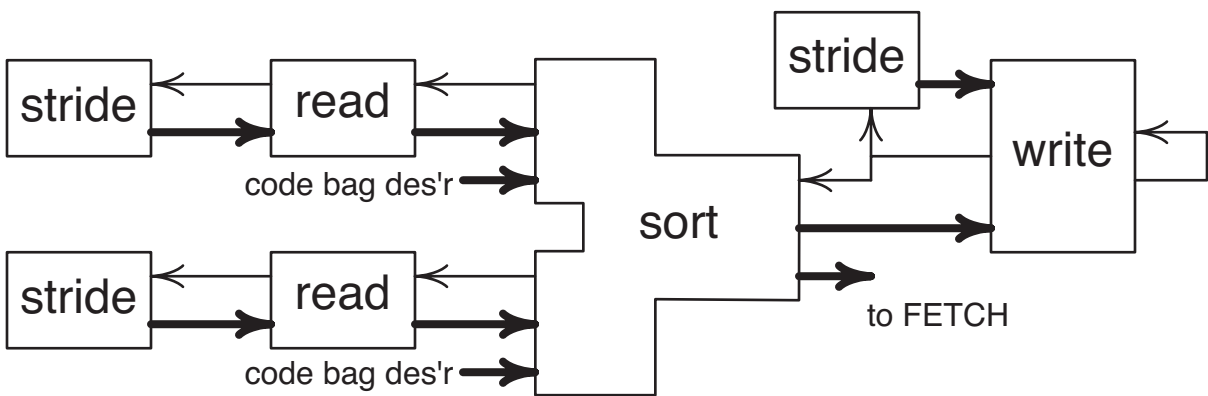


Figure 11: A sorting pipeline