

# UC Berkeley Computer Science

**Subject:** A Record Store Implementation  
**Date:** September 6, 2006  
**From:** Ivan Sutherland  
**UCIES** #2006-is32

## 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-is23: FIFO Register File, Ivan Sutherland, 30 June 2006  
UCIES# 2006-is24: Parallel Switch Fabrics, Ivan Sutherland, 10 July 2006  
UCIES# 2006-is25: Instruction Sequence in FLEET, Ivan Sutherland & Igor Benko, 19 July 2006  
UCIES# 2006-is26: Record SHIPs, Ivan Sutherland, Igor Benko & Mark Greenstreet, 29 July 2006  
UCIES# 2006-is27: Scatter and Gather, Ivan Sutherland, 2 August 2006  
UCIES# 2006-is28: How Long is a Record, Ivan Sutherland, 10 August 2006  
UCIES# 2006-is29: The Size of a Record, Ivan Sutherland, 7 August 2006  
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

## INTRODUCTION

There was some discussion in class today about how a record store might interact with the switch fabric. This memo records the ideas that came to mind during that discussion and immediately afterwards. The discussion centered on how a record store knows how many records it contains and remembers where they begin and end. It would be nice to have a simple implementation for the record store.

A record is an ordered set of data values with between one and eight elements. The upper bound of eight is entirely arbitrary. I chose eight by an efficiency argument. First, I made an arbitrary choice to use a power of two as the maximum. Second, I assume that the “cost” of each MOVE instruction in overhead and instruction fetch is about the same as the cost of moving one data element. With that cost structure, a 4 element record has 80% payload and 20% overhead. An 8 element record has only 11% overhead and 89% payload. A 16 element record has only 5.8% overhead and 94.2% payload. I estimate, with no facts to back up my estimate, that the cost in storage of allowing records longer than 8 exceeds the value of the overhead reduction of 6.2%. When we know more about real costs, I shall be happy to entertain proposals for longer or shorter maximum length records, including the proposal put forward today for a prime number as the maximum length.

## THE OUTPUT INTERFACE

Instructions can choose to MOVE or COPY an entire record or just the first element of the record. An instruction that MOVES only the first element need not wait for an entire record to be available, only for a single available element. A MOVE instruction for an entire record, on the other hand, must wait until there is an entire record available to move. Thus the output interface of a record store must reveal to the instruction process both the presence of entire records and the presence of a individual data elements

---

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.

A MOVE of the first element will peel off the first element of the record and send it into the switch fabric as a record of length one. It will eliminate the first element from the record, allowing the rest of the record, if any, to advance towards the output of the record store. A MOVE of the entire record will move all of the elements of the record sequentially, possibly taking many cycles at the output interface to move the entire record. A marker in the record store must tell the output interface when it has reached the end of the record.

## **INPUT INTERFACE**

The input interface gets data from the switch fabric. It will get elements of a record in succession, as many elements as are in the record that is passing through the switch fabric. The last such element must be marked as the last so that the input interface of the record store knows that no further data elements will arrive and that the record is complete.

## **FIRST-IN-FIRST-OUT REGISTERS (FIFOs) – A very short tutorial**

A FIFO is a series of latches, called stages, with an input end and an output end. Each stage of such a FIFO can be full or empty. A full stage holds meaningful data; the latches of an empty stage may contain data, but because the stage is known to be empty the state of those latches is irrelevant. The control circuits of the FIFO have a full/empty bit for each stage that records whether that stage is full or empty. Figures in this memo omit the full/empty control bits and illustrate only the data latches.

Whenever a full stage has an empty successor stage, a three-part atomic operation takes place. First, the empty stage copies the data from its full predecessor, placing the same data in both stages. Second, the successor stage is declared to be full by setting its full/empty bit to the full state. Third, the predecessor is declared to be empty by setting its full/empty bit to the empty state. The data abandoned in the predecessor stage will henceforth be ignored.

In an asynchronous FIFO each stage performs such a three-part atomic operation whenever local conditions permit. In a FIFO of synchronous design, the atomic operations happen only when not only conditions permit but also a clock pulse happens. I find design of asynchronous FIFOs easy enough because the local rule of behavior is so simple. In effect, the local control circuits generate a local clock pulse that activates some latches and changes the state of two full/empty bits.

The action of the FIFO control forces data towards the output end of the FIFO. If there is no new data entering and nothing leaving, the occupied part of the FIFO will be the contiguous stages at the output end of the FIFO. The empty part of the FIFO will be the contiguous stages at the input end of the FIFO. If elements are removed from the output, all data values eventually advance one position towards the output and the number of empty stages increases by one. The control acts so as to keep the final output stage full if that is possible. Only when all stages of the FIFO are empty will the output stage remain empty.

You've now had a very brief explanation of how a FIFO works. You may want more, and you'll be able to get more in class. We'll often do a Kinetic Learning Activity (KLA) about FIFOs to get a feel for how they behave.

## **THE EQUIPMENT – Figure 1**

Now let's turn our attention to Figure 1 which shows how two FIFOs can implement a record store. Each of the rectangles in the figure represents a FIFO eight

stages long. Data in these FIFOs flows from left to right, from input to output. The figure omits the full/empty bits for these FIFOs. You have to imagine the action of the control; data flows as far to the right as possible in each of the rectangles.

The lowest box in the figure, the “elements FIFO”, represents a FIFO whose latches can store the N-bit data elements of a record. Each of its eight stages has enough latches, i.e. is “wide” enough to store a single data element of N bits. The N bits in each data element in this FIFO move forward together from one stage of N latches to the next.

The middle box in the figure, the “end FIFO”, represents eight latches that each hold a single bit. These bits serve as markers to identify the last element of a record. A one in such a latch means that the corresponding element of the data is the last element in the record. A zero means that the corresponding data element is not the last. Note that in addition to the latch recording the last/not last choice, there is also a full/empty bit, so that each stage of this FIFO has three states: 1 = end, 0, and empty.

I have drawn the top box in the figure, the “record queue”, shorter than the others, but it is also a FIFO eight element long. The data elements stored here are zero bits wide; the record queue has no data latches; its only storage is the full/empty latch of its control. Such a data-less FIFO is sometimes called a queue, and the elements in it are sometimes called tokens. You can think of these zero-bit data elements as markers. Each stage either has a marker or it is empty.

So here’s the general idea. The elements FIFO holds the record’s data; the end FIFO marks the last data element of each record, and the record queue holds a token for each complete record in the record store. In the paragraphs that follow I’ll try to explain in general terms how the input and output interface act to preserve the integrity of the information in these three FIFOs.

## INPUT INTERFACE REVISITED

The input interface signals with dark arrows are data values. The input interface signals with lighter arrows are control signals required to couple the first stage of the FIFOs of the record store to the circuits of the switch fabric. Roughly speaking the signal called req reports the state of the full/empty bit of the last stage of the switch fabric, and the signal called akg reports the state of the full/empty bit of the first stage of the elements FIFO.

Notice that the figure omits control signals for the end FIFO. In fact, the end FIFO and the elements FIFO can share a single control mechanism because they always operate in lock-step.

Remember that as a record flows through the switch fabric, element by element, its last element and only its last element, carries a mark, “the purple bit.” When a data element arrives, at the input interface copies its data element value into the elements FIFO. It also puts the end marker, or lack thereof, into the end FIFO. Thus if an element carries the purple bit indicating that it is the last element, a one goes into the end FIFO. If the element carries no purple bit, a zero goes into the end FIFO.

Finally, if, and only if, a purple bit arrives, a token goes into the record queue. Thus the number of tokens that get into the record queue and the number of ones that go into the end FIFO are identical. However, because only purple bits cause entries in the record queue, the tokens in this queue are adjacent, whereas the 1’s in the end FIFO may be spaced apart by zeros marking data elements that are not the final ones in their record.

In a later section I shall describe a slight modification of this procedure that may have important programming benefits, but I'll save that until after I describe the output interface action.

## **OUTPUT INTERFACE REVISITED**

How does the output interface know if there's a complete record ready to enter the switch fabric? If the record queue has a token at its output end, there is at least one record available to send into the switch fabric. That first record includes at least the first data value in the elements FIFO and additional elements up to and including the first one that matches with a 1 in the end FIFO. There may be more records following, but they don't immediately show at the output interface.

A single-word MOVE instruction removes an element from the elements FIFO and from the end FIFO. If the end FIFO bit that was removed happens to be a 1, it also removes a token from the record queue.

A record MOVE instruction does nothing until there is a token at the output of the record queue. Then it removes that token. It also removes data elements from the elements FIFO and corresponding end marks from the end FIFO up to and including the element with the 1 in the end FIFO.

A single word COPY instruction is the same as the single word MOVE except that it avoids making the last stage of the record store empty. It leaves the state of the record store entirely unchanged.

A full record COPY is harder, because the data in the FIFOs must circulate to expose each of the data elements so that they may flow into the switch fabric. I suggest that an "end around" flow path, not shown, be used to copy the elements of the record back into the input of the record store as they enter the switch fabric. If there is more than one record in the record store, however, this procedure will put the first record last. Although that seems a little peculiar, I believe it may be workable.

## **A SLIGHT MODIFICATON OF MOVE**

We have talked about using the single word MOVE instruction to "scatter" the elements of a record. Someone asked in class what is the corresponding "gather" operation. What one wants in a gather operation is to collect together a group of single words and declare them to be a record.

There is a bit in each instruction that distinguishes full record moves from single word moves. Let us think of it as the "source record/word" bit. The source record/word bit is strongly associated with the source address. Equipment at every source implements its action. Is there a corresponding bit associated with the destination address?

We can create a corresponding bit for destinations. What it will do is to declare that this MOVE carries data that completes the record, or only data for the first part of a longer record. One can think of this as the "more/done" bit. Done means the record is complete with this MOVE. More means that subsequent MOVE operations will bring additional values to append to these to make a complete record. The final data delivered to the record store must have the more/done bit in the done setting.

The data to append can be a record of one or more elements. Thus short records can be assembled into longer ones by moving all but the last record to be so assembled in the more setting and the final one in the done setting.

The symmetry of the word/record bit in the source field and the more/done bit in the destination field appeals enormously to me. It give us a way for any record store to assemble pieces into a complete record.

Moreover implementation is simple. As the astute reader will already have realized, the only change is at the input interface. Move instructions that are done act as described. MOVE instructions marked MORE simply ignore the purple bit and avoid inserting a one into the end FIFO and a token into the record queue because the record being assembled is not yet complete.

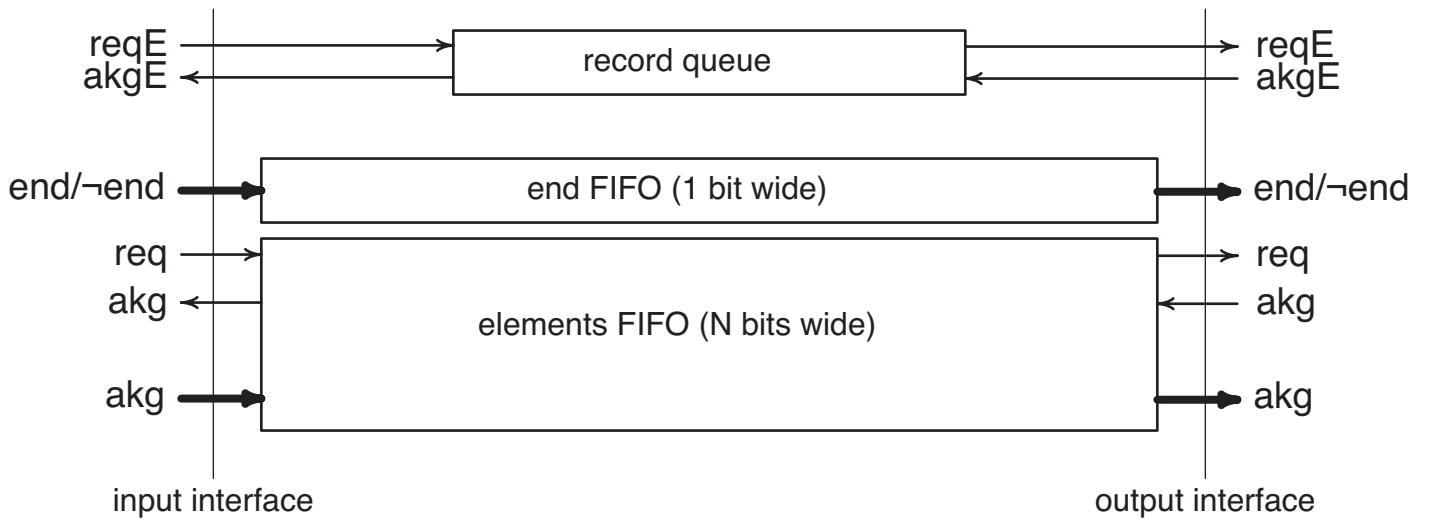


Figure 1: A Record Store

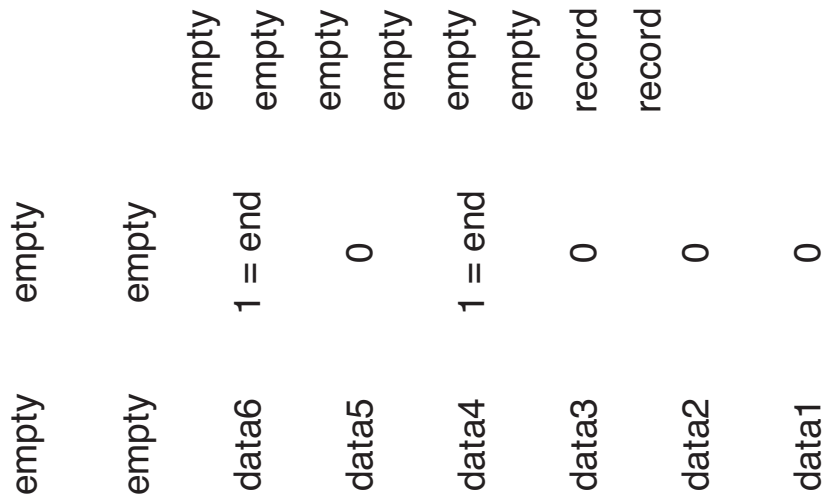


Figure 2: Content for a four-element record followed by a two-element record.