Low Latency Self-Timed Flow-Through FIFOs

Erik Brunvand
Department of Computer Science
University of Utah, SLC, Utah, 84112

Abstract

Self-timed flow-through FIFOs are constructed easily using only a single C-element as control for each stage of the FIFO. Throughput can be very high in this type of FIFO as the communication required to send new data to the FIFO is local to only the first element of the FIFO. Circuit density can also be high because the control overhead is very small. However, because data must travel through every cell in the FIFO when moving from input to output, latencies can be long.

This paper describes some alternative approaches to building self-timed flow-through FIFOs that reduce the latency while retaining the high throughput and relative simplicity of a flow-through design. Five designs are presented: a standard linear flow-through FIFO in which the data pass through every latch in the FIFO, a parallel FIFO in which data are delivered in turn to a set of parallel flow-through FIFOs, a tree FIFO in which data are fanned out into a tree of simple FIFOs, a square FIFO in which the tree is organized as a square array to achieve better layout packing, and a folded FIFO in which data will try to skip as many of the empty FIFO cells as possible to find the shortest path to the output.

1: Introduction

Self-timed flow-through FIFOs have a particularly simple two-phase implementation using a single C-element as the control for each FIFO stage [10]. Because the input process communicates locally only to the first cell in the FIFO, throughputs can be high for these designs. The throughput is determined only by the speed of the individual cells and is not dependent on the size of the FIFO. Latencies, however, can also be high because the data must move through every stage of the FIFO before becoming available at the output. As the depth of the buffer increases, so does the latency.

In this paper, I present some alternative approaches to organizing self-timed flow-through FIFO circuits. These circuits attempt to retain the desirable features of a flow-through FIFO, while reducing the latency. The key idea is to keep the data from having to travel through every cell in the FIFO on its journey from input to output. In addition to lower latency, these FIFO circuits should exhibit lower power dissipation than a linear flow-through design since a shorter path from input to output means fewer signal transitions are required to move data through the FIFO. In an empty linear FIFO, for example, every cell in the FIFO must pass the data before it is available at the output. In a full case, removing a word means that all the data currently in the FIFO must move up one slot and again every cell in the FIFO must pass data.

There are a number of possibilities for organizing self-timed flow-through FIFOs with reduced latency. In this paper five different circuits are presented:

Linear FIFO — This is the standard micropipeline FIFO [10]. This consists of a linear array of FIFO cells in which data must flow through every cell in the FIFO on the way from input to output.

Parallel FIFO — In this FIFO, data are delivered to a number of parallel FIFOs, and then collected into a single stream at the output. The input side of the FIFO distributes the data to the parallel FIFOs in a fixed order, and the output side delivers data from those parallel FIFOs in the same order.

Tree FIFO — This FIFO is an extension of the Parallel FIFO in which the data are fanned out in a binary tree of simple FIFOs, and then collected back in another binary tree to a single output stream.

0-8186-7047-9/95 $04.00

Square FIFO — This FIFO is organized as a square array of FIFO cells rather than as a tree. Data take an L-shaped path through the square array making the latency slightly higher than for a tree, but allowing for the possibility of more compact layout due to the square shape.

Folded FIFO — This FIFO attempts to move the incoming data ahead of as many empty FIFO cells as possible by looking at the state of the FIFO cells near the output. Arbitration is required to make sure that a cell's status is not changing as it is being checked.

The circuits described here were developed using the Viewlogic schematic capture and simulation tools, and Actel FPGAs and CMOS standard cells as target devices. The FPGAs allow the FIFO circuits to be tested quickly for functionality, but are not directly comparable to the CMOS versions. Performance results from the FPGAs in terms of absolute time are influenced by the speed and routing of the FPGA, and performance in terms of gates delays is influenced by the availability of particular gate types on the FPGA. In particular, the folded FIFO requires an arbiter circuit that can only be approximated using the FPGAs so timing based on that approximation is particularly suspect. The CMOS versions were created using the Lager silicon assembly system and the ITD CMOS standard cell library (v2_3) [6]. A comparison of all the FIFOs in terms of numbers of generic FPGA gates, and of the linear and folded FIFOs in terms of spice simulation of CMOS circuits is given in the conclusions.

2: Self-Timed Circuit Modules

Although self-timed circuits can be designed in a variety of ways, the FIFO circuits considered here use two-phase transition signaling for control and a bundled protocol for data paths. However, the particular signaling protocol used is not as important as the overall FIFO organization. The organizations presented in this paper could be used easily with other forms of self-timed signaling and control.

Two-phase transition signaling [9] uses transitions on signal wires to communicate. Only the transitions are meaningful; a transition from low to high is the same as a transition from high to low and the particular state, high or low, of each wire is not important. The communication protocol uses two wires called some variation of *Req* and *Ack* to request service, and acknowledge completion respectively.

A bundled data path uses a single set of control wires to indicate the validity of a *bundle* of data wires. This requires that the data bundle and the control wires be constructed such that the value on the data bundle is stable at the receiver before a signal appears on the control wire. This condition is similar to, but weaker than, the equipotential constraint [9]. Its use is a compromise to complete self-timing but one which allows the use of standard data path circuits in conjunction with the transition control circuits.

The FIFO cells described here are built using a library of self-timed two-phase circuit modules similar to those presented in Sutherland's paper [10] and implemented in a variety of technologies [3, 5]. They include the following circuits:

Merge The "OR" function for transitions, implemented by an XOR gate.

Join The "AND" function for transitions, implemented by a C-element.

Call A module that acts as a hardware subroutine call allowing multiple access to a shared resource. The Call module routes the *Req* signal from a client to the subroutine, and after the subroutine acknowledges, routes the *Ack* back to the appropriate client. The requests must be mutually exclusive.

Select A module that steers an input transition to one of two outputs based on the value of a Boolean *SEL* signal. The *SEL* signal is a bundled signal with respect to the input transition.

Q-Select A module like a Select, except the *SEL* signal is not bundled and may be changing even when the Q-Select is looking at it. Thus, it requires some way of sampling the *SEL* signal reliably. A Q-Select uses a Q-flop [8] or some other form of arbiter to perform this sampling.

Toggle A module that produces transitions alternately on two or more outputs in response to input transitions.

Latch A module that latches bundled data signals upon receipt of transition control signals.

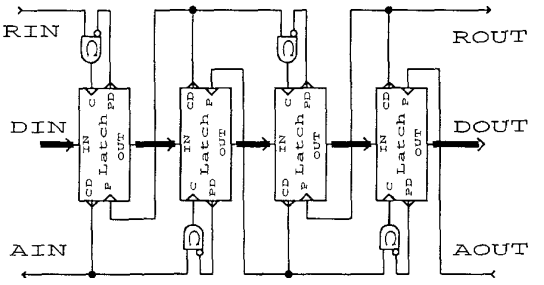

Figure 1. Linear Self-Timed Flow-Through FIFO

3: Linear FIFO

Linear self-timed flow-through FIFO using two-phase transition control is described in detail in Sutherland's paper [10]. The key idea is to use a C-element as the local control for each stage of FIFO, and then to stitch together identical FIFO stages as required to make a FIFO of the desired depth. The throughput of the FIFO is determined by the response time of a single FIFO cell and is not related to the overall depth of the FIFO. The latency, however, clearly is a function of the depth of the FIFO as each data item moves through every cell of the FIFO on its way from input to output.

Four stages of linear FIFO are shown in Figure 1. Each latch is a transition latch that captures bundled data in response to a transition on the *C* (capture) control input, and becomes transparent from input to output in response to a transition on the *P* (pass) control input. In the micropipeline latch, a *C* transition is followed by a *Cd* transition to indicate that the capture is done. Likewise, a *P* transition is followed by a *Pd* transition to indicate that the pass function is done. This type of latch can be built directly using a pair of gated latches gated on opposite edges of the clock and a mux on the output for each bit, or by using standard gated latches with an XOR generating the gate signals from the *C* and *P* inputs and a Toggle module generating the *Cd* and *Pd* signals [10]. The direct approach is taken in the FPGA latches because this maps well to the Actel architecture. Each latch bit uses only one Actel basic module. For the CMOS circuits, standard gated latches are used from the ITD standard cell library.

The control for standard micropipelines is built from C-elements. These gates, drawn as an AND gate with a C inside, will drive their output low when both inputs are low, and high when both inputs are high. When the inputs are at different states, the output is held at its previous level. Note that one input of the C-element used in each stage of Figure 1 is inverted. Thus, assuming that all the control signals start low, the C-element will produce a transition to the capture input of the latch when the input request *RIN* line first goes from low to high. The acknowledge from this latch, *Cd*, will produce a similar request through the next C-element to the right. The C-element at each stage will not produce another request to its latch until there are transitions both on *RIN* (signaling that there are more data to be accepted) and on *AOUT*, the Ack from the next latch to the right (signaling that the next stage has finished with the current data). Each FIFO stage acts as a concurrent process that will accept new data when the previous stage has data to give, and the next stage is finished with the data currently held. Notice that the *AOUT* from the following FIFO stage which signals that the current data have been taken first makes the current latch transparent using the *P* signal before enabling the C-element to latch new data into the current latch.

A nice feature of a flow-through FIFO of this sort is that when the FIFO is empty, all the latches in the FIFO are transparent. This allows the FIFO, and any processing logic that may be inserted between the FIFO stages, to be tested for functionality before the storage capacity of the FIFO is tested.

All of the FIFOs shown in this paper use an 8-bit wide data path. This is the FIFO circuit that is used as the basis for comparing each of the other FIFO designs. In particular, an 8-bit FIFO with a capacity of sixteen words is the circuit which is implemented using each of the FIFO organizations.

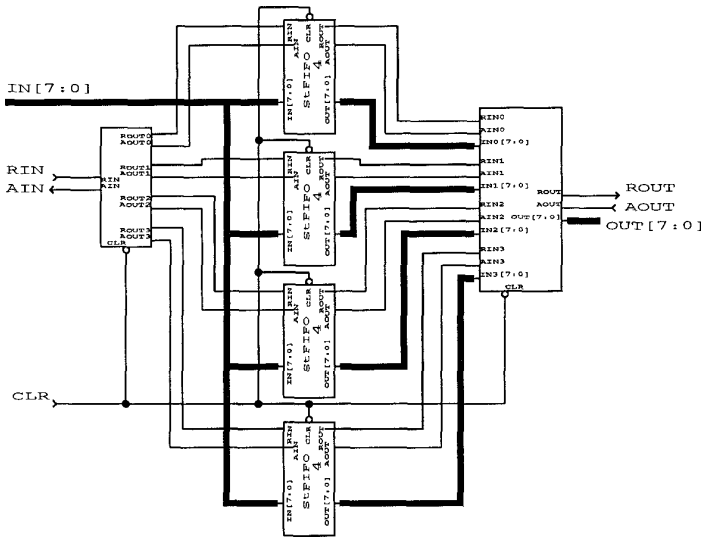

Figure 2. A 16-Deep Four-way Parallel Flow-Through FIFO

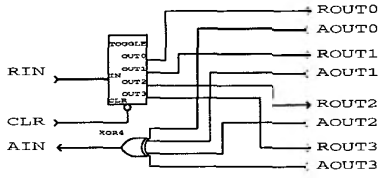

Figure 3. A Four-way Toggle-Distribute Circuit

4: Parallel FIFO

In a parallel flow-through FIFO, data are delivered to a number of shorter FIFOs that operate in parallel. Thus, depending on the number of parallel FIFOs used, the data travel through far fewer cells on their path from input to output. For example, in the 16-deep case and splitting the data into four parallel FIFOs each data word travels through only four FIFO cells on the way from input to output reducing latency by approximately a factor of 4. There is some overhead in the circuits that distribute and collect the data from the parallel FIFOs, but the latency reduction is still significant.

The distribution of data into the four parallel FIFOs is accomplished using a four-way transition Toggle on the input side to send data alternately to each of the four FIFOs, and a four-way Toggle on the output side to collect the data from the four FIFOs in the same order that they were distributed. The circuit for the 16-deep four-way Parallel FIFO is shown in Figure 2 where each of the parallel FIFOs are of depth four. The circuit used to distribute the data at the input end of the FIFO is shown in Figure 3, and the circuit used to merge the four streams back into one is shown in Figure 4.

Figure 3 shows the four-way toggle-distribute circuit. This circuit takes the *RIN* (request-in) and sends it

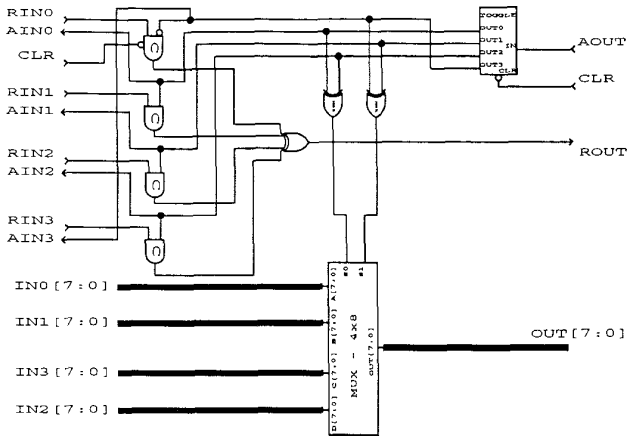

Figure 4. A Four-way Toggle-Merge Circuit

alternately to each of the four output requests using a four-way Toggle. Each time an output request is sent, the corresponding acknowledgment is sent back to the *AIN* input acknowledge through the XOR (merge). The Toggle module is responsible for sending output requests in the proper alternating order. Note that another way of describing the behavior of an n-way transition Toggle circuit is as an n-bit transition-sensitive Johnson counter.

The four-way toggle merge shown in Figure 4 is slightly more complicated. Each of the parallel FIFOs may send a request to the merging circuit. This is because multiple data words may be sent to the FIFO before the first is removed. Requests from each of the four parallel FIFOs can be pending before data are removed from the first. However, the circuit must take the requests in the same order in which data were originally put into the parallel FIFOs. That is, it must take the request from the topmost FIFO first, followed by the next lower FIFO, and so on. So, each of the incoming requests from the parallel FIFO are passed through a C-element. This allows the merging circuit to let only the signal it is interested in pass to the output. Notice that the topmost C-element is the only one with an inverted input. Thus the first output request from the top FIFO will make it through to the XOR, and to the *ROUT* signal. When the *AOUT* signal announces that the consumer has finished with the data, the four-way Toggle will enable the next of the parallel FIFO requests by sending a transition to the next C-element in the ordering. After the last FIFO has delivered data to the output, the first FIFO is again enabled to send its request.

The other complication in the merging circuit is that a multiplexer must be used to merge the four data streams into a single output stream. The mux used is a standard mux that chooses which data to pass based on level control signals. The control signals from the FIFOs, however, are all transition signals. Using a fairly standard trick, an XOR is used to manufacture each level select signal from the transition control signals. In this case, the mux starts out with its select signals at 00 (all control signals are 0 after a master clear). Once the data from the *IN0* channel has been passed to the output, the *AOUT* is routed through the four-way Toggle and goes to the C-element of the *IN1* channel. On the way, that transition also changes the select lines of the mux to 01 through the XOR gate on the *S0* select line. The next signal, the *OUT1* signal from the four-way Toggle, changes the other mux select bit making the select lines 11. The next word changes the low-order mux select bit back to 0 making 10, and finally the transition from the next word causes the select to change back to 00 and the process repeats. Each *AOUT* that passes through the four-way Toggle sets the mux select lines to the next input in line. The Toggle module is responsible for making sure that data are removed from the parallel FIFOs in the same order in which they were entered.

Ignoring the overhead of the distribution and merging circuits, The parallel FIFO reduces latency by a factor equal to the number of parallel FIFOs. As can be seen from the figures, the overhead for the extra control

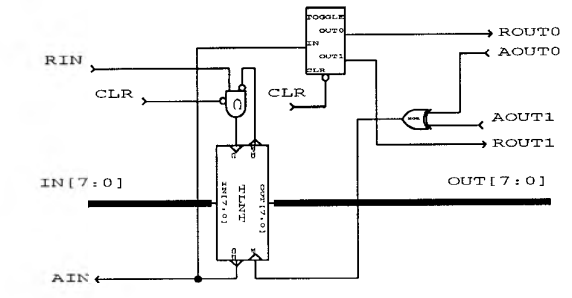

Figure 5. A two-way Toggle-Distribute Circuit

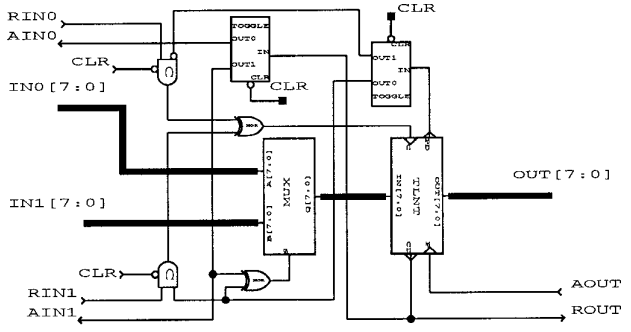

Figure 6. A two-way Toggle-Merge Circuit

is reasonably small so the final latency improvement is quite good. Parallel FIFOs with many short FIFOs in parallel can reduce the latency even more at the expense of more circuit overhead. In the extreme case, the data only need travel through a single FIFO cell from input to output and the distribution and merging circuits now look like address counters and decoders for a ring-buffer type FIFO. More details on the construction of the n-way toggle circuits required to build the distribution and merging circuits may be found in [4].

5: Tree FIFO

A tree FIFO is essentially a parallel FIFO but one in which each of the parallel FIFOs are also parallel. In this case, the tree is binary, although you could easily imagine a tree with a different branching factor. The data are fanned out to a binary tree of FIFO cells, and then collected in another binary tree to a single output cell. In a tree FIFO of this type each data word travels through $2log(n)$ stages from input to output rather than n stages. Two modifications of a single-stage FIFO cell are required: one to distribute incoming data to two outputs in an alternating fashion, and one to collect data from alternate inputs and merge them to a single output data path. The circuits, shown in Figures 5 and 6 are similar to the parallel FIFO circuits shown in the previous section, but perform only a two-way distribute and merge.

The main difference between the distribute and merge circuits in the parallel FIFO and these in the tree FIFO is that in the parallel FIFO the circuits are used only to distribute and merge the control signals. As no data are latched, the distribution and merging circuits in the parallel FIFO are not, themselves, FIFO stages that store data. In contrast, the circuits in the tree FIFO are actually FIFO stages. Each stage stores a single data word. The distribution circuit in Figure 5 receives data on a single input line and stores it in a latch. It

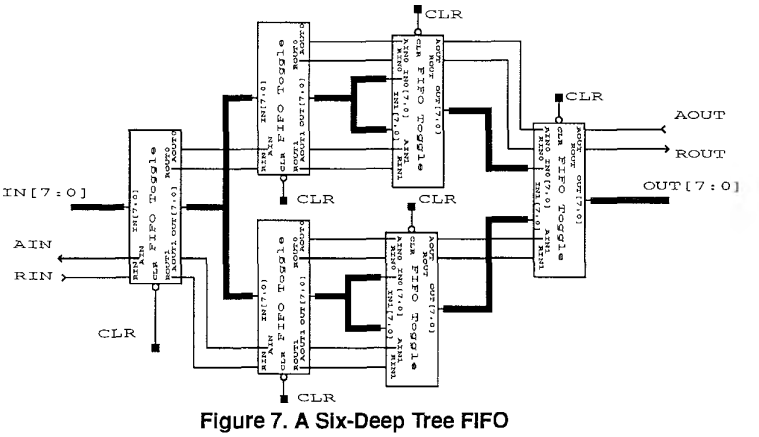

Figure 7. A Six-Deep Tree FIFO

then delivers that data alternately to two different outputs through a two-way Toggle.

The corresponding toggle-merge circuit in Figure 6 is also a FIFO stage. In this case, data words are taken from each of the two input ports in alternating fashion similar to the parallel FIFO merge circuit. Note that, as in the previous circuit, the top input port, which is the first to deliver data, has an inverted input on the C-element, and the other C-element does not. This makes the top C-element "half-cocked" which allows the first data word to make it through to the latch and start the merging process. The mux in this figure chooses which input is eventually stored in the latch. The select signal for the mux is generated using the XOR gate which generates a level signal from the transitions that control the latching of data. The select signal starts off at 0 allowing the top data path to be latched. When those data have been consumed by the following FIFO stage, the bottom C-element is enabled, and the mux select signal is switched to 1 allowing the bottom data path to be latched. The acknowledgment of the latching action turns the select signal back to 0 in anticipation of the next word to be received from the top. The control interface to the latch in Figure 6 goes through a Toggle element because the two input ports write into the latch alternately.

Once you have the toggle-distribute and toggle-merge FIFO stages, they can be combined in a binary tree to make a flow-through FIFO. An example of this connection is shown in Figure 7 where a six stage FIFO is built using three toggle-distribute modules and three toggle-merge modules. In this circuit, data are delivered through the tree by alternating between top and bottom paths in each FIFO element. The entire collection behaves exactly like a six-deep FIFO, but data pass through only four cells in their way from input to output instead of six as would be the case in a linear FIFO. This binary tree organization can, of course, be repeated in various ways to make tree FIFOs of any size.

In a tree FIFO of depth sixteen, for example, the cells can be organized so that the longest path from input to output passes though only eight FIFO stages instead of the full sixteen that would be required in the linear case. In general, the latency of a data word through a tree FIFO is $O(log(n))$ as opposed to the $O(n)$ latency of a linear flow-through FIFO.

6: Square FIFO

Although the advantages of a tree-structured FIFO are intriguing, one drawback is that the physical tree structure may not pack well onto the two-dimensional surface of an IC. Regular rectangular shapes might pack better in terms of VLSI layout than a tree-based design. One approach is to build a square array where the data travel through the array in an L-shaped pattern. This is actually somewhat similar to the parallel FIFO discussed in the previous section where data are distributed to a set of FIFOs in parallel. In the case of the square FIFO, as in the tree case, the individual cells that do the distribution are, themselves, FIFO cells storing

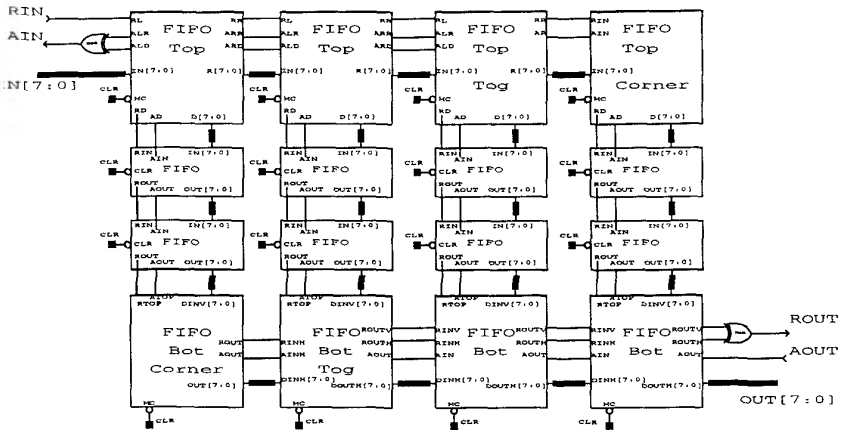

Figure 8. A 16-Deep Square FIFO

data that pass through them.

A square FIFO circuit is shown in Figure 8. The intuition for this circuit is that data go into the top FIFO as if it were a linear FIFO. The first word goes all the way to the right end before dropping into the vertical FIFO in the middle. The second goes to the third column, the next to the second column, and the next drops into the first column and the cycle repeats. The bottom row then picks up data from the columns in the same order and passes them out the bottom row as if that row were linear FIFO. It is as though there were a bit circulating in the top row that determines whether the data exits that cell to the right or to the bottom. That bit is passed one cell to the left after each data word is passed out the bottom, and then recirculates to the far-right cell. Thus, after a clear, the "drop bit" is in the far right cell in the top. After data move through that cell and drop out the bottom, the "drop bit" moves to the third cell in the top row. When data come into the third cell and drop out the bottom, the bit moves to the second cell, and so on. The bottom row collects data from the vertical FIFOs in the same order using the same idea.

The challenge is to generate the effect of this "drop bit" using two-phase bundled control and not by passing Boolean data between the FIFO cells. The first thing to notice is that the vertical FIFO columns are nothing but standard FIFO. Of course, you can implement this FIFO in any way you choose so it might very well be implemented as some form of lower latency FIFO. In any case, consider the top row that does the distribution of data to the columns. The top right corner is just standard FIFO. Any data that make it to the top right corner will drop out the bottom because that is the only place for it to go. That cell is simple micropipeline FIFO put in a new symbol to match the shape of the other cells in the top row.

Moving left on the top row, the next-to-last cell in the row toggles between sending data to the right and sending data to below. A Toggle-based circuit similar to the tree-distribute circuit is what is called for here. The difference is in how the information about whether to pass to the right or drop out the bottom is passed to the other cells in the row. The scheme used here is that each cell sends an output request to the cell to the right. Instead of receiving a simple acknowledgment of that request, the cells will actually receive one of two possible acks; an Ack-Left-Right (*ALR*) or Ack-Left-Down (*ALD*). An *ALR* means that the data are acknowledged, and that the next data should be passed to the right. the *ALD* means that the data are acknowledged, and the next data should be passed down. This is accomplished in the next to last cell in the top row using a Toggle module in much the same way as in the tree FIFO (see Figure 5).

The other FIFO cells in the top row obey the following protocol. First, they latch data in response to a request from the left. Now, depending on the type of the last acknowledgment they received from the left, pass the data either to the right or down. Because this is not a strictly alternating choice, the control cell is a Select rather than a Toggle. The Select cell will, in the normal case of the *SEL* signal being 0 as it is after a

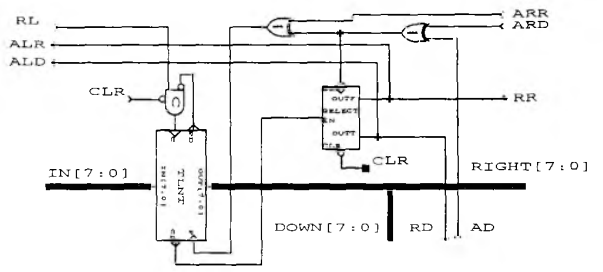

Figure 9. A Row FIFO for the Top Row of a Square FIFO

clear, pass the data to the right, and acknowledge that the next data in the previous cell should also be passed to the right using the *ALR* acknowledgment signal. As long as the acks from the right continue to be *ARR* acks, this cell will continue to pass data to the right, and to tell the previous cell to continue to also pass data to the right. If, however, the last ack from the right was an *ARD* meaning that the next data should be passed down, the *SEL* signal to the Select is changed to a 1, and the next data that come through the cell will be passed down. Furthermore, the ack to the left will be of the *ALD* variety telling the previous stage that its next data should go down rather than right. The ack from the bottom channel, *AD*, will cause the *SEL* signal to be reset to 0 so that the next data will be passed to the right again until the next *ARD* says otherwise. This circuit is shown in Figure 9.

These three cell types are combined to form the top row of a square FIFO. The first data make it all the way to the far right. The second data get dropped into the third vertical FIFO, the next to the second, and the next to the first. The fifth data entered from the top left will again make it all the way to the far right and the cycle repeats. Thus, data entered into the FIFO are distributed, through the top row FIFO, into the parallel columns of the square FIFO. As they make it through the parallel FIFO in the body of the square FIFO, they must be removed through the bottom row in the same order in which they were entered in the top.

The bottom row must collect the data from the parallel FIFO. As with the top row, there are three different types of FIFO cells in the bottom row. The bottom left corner is just a simple FIFO. Its job is to always take data from the vertical port and attempt to pass it to the right on the bottom row.

The next cell to the right of the corner is another form of toggle FIFO. This cell will take data alternately from the top port and the left port and pass them to the right in the bottom row. This action is very similar to the toggle-merge FIFO cells in the tree FIFO and includes the same complication of using a mux to choose which data path is latched into the internal latch (see Figure 6). The only difference is that this toggle-merge FIFO sends one of two requests to the right instead of one. These serve the same sort of purpose as the two acks in the top row. If the FIFO to the right receives an *ROUTH*, this means that after receiving this word, the next word should be received from the horizontal direction (i.e. from the left). If the request to the right is an *ROUTV*, then after receiving this word, the receiving cell should take the next word from the vertical (top) input port. These requests are generated easily in this case by using the outputs of the Toggle which become the *AIN0* and *AIN1* signals in Figure 6 also as the *ROUTV* and *ROUTH* signals.

The circuit for the rest of the bottom row is a little trickier. This is because of the required response to the two requests from the left. For example, receiving an *RINV* from the left (i.e. the signal produced as *ROUTV* in the cell to the left) means that *after* receiving that word from the left, the *next* word should be taken from the top instead of the left. Somehow this information must be stored so that behavior is modified the next time data are received. This is what the circuit in Figure 10 does. There are separate C-elements synchronizing the two input channels. Note that since the first data after a clear will come from the top, the C-element for the top channel is half-cocked. Thus, when the first data are available from the top having made it through the mux, they are stored in the latch. When the data are latched, the ack is sent to the cell on the right as *ROUTH* meaning that that cell should take the following word from the horizontal direction. Notice that this transition also changes the mux so that the next data will come from the right rather than the top, and also changes the top Select module so that until things change, further data will also result in an *ROUTH* request to the right. At this point the bottom Select is also set so that acks from the right will cause data to be taken

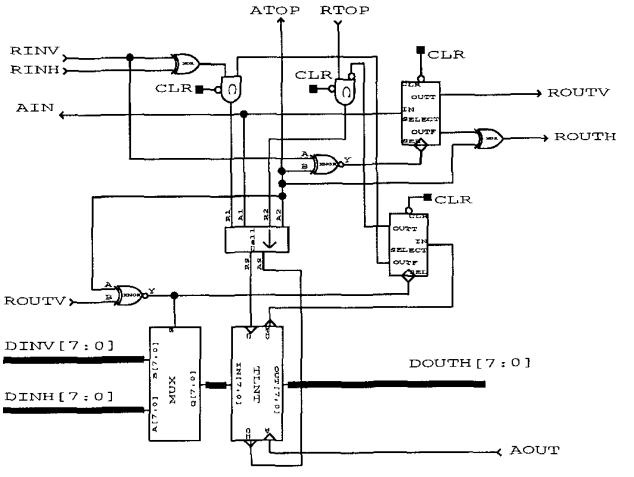

Figure 10. A Bottom-Row FIFO for a Square FIFO

from the horizontal input.

As long as requests from the left continue to be of the *RINH* variety, data will continue to be taken from the horizontal port. The ack from the latch through the Call will be sent through the top Select element and become an *ROUTH* signal to the right, and acks from the right will go through the bottom Select element and re-enable the C-element that accepts input from the left. If the request from the left is an *RINV*, however, things change. In that case, the data are latched from the left as usual, but the *RINV* signal changes the top Select element so that the outgoing request to the right will be an *ROUTV*. When that *ROUTV* signal is delivered to the right, it also switches the mux of the current stage so that the *next* data will be latched from the top channel instead of the right. This also changes the bottom Select so that the ack from the right will be routed to the top channel and thus wait for data from that channel through the C-element. Note that this must happen *after* the current data are latched because they came from the left. This is why the signal that changes the mux and bottom Select come from *ROUTV* rather than from *RINV* like the top Select.

Once the mux and Selects are set to select the next data from the top, data from the top are latched into the current latch. When these data are acknowledged, the ack from the latch resets the mux, and both of the Selects putting them back in horizontal mode.

These cells, arranged in the organization shown in Figure 8, make a FIFO where each data word passes through the FIFO in a right-angle pattern. Data are distributed into the vertical columns by the top FIFO, and collected from the columns by the bottom FIFO. The latency of a single word is related to $O(\sqrt{n})$ for this FIFO rather than $O(log(n))$ for the tree, and $O(n)$ for the linear FIFO. Furthermore, the square arrangement may be more amenable to planar VLSI layout than the tree.

7: Folded FIFO

The final type of FIFO presented here takes quite a different approach to reducing the latency of a flow-through FIFO. The intuition for this FIFO is that if the FIFO is empty, why should the data flow through any empty cells at all? It may as well skip over all the empty cells and go directly to the output cell. In order for this to happen, however, the FIFO cells need some way of knowing which cells are empty, and which are not. Checking this empty/full condition of other cells in an asynchronous FIFO will require arbitration of some sort because the processes attached to the input and output of the FIFO are not synchronized. Thus, the *full* condition of a FIFO word may be changing at the same time as it is being checked.

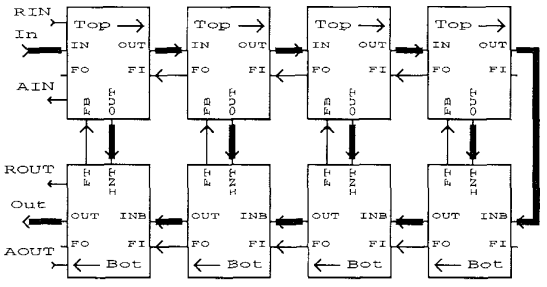

Figure 11. A Folded, Arbited FIFO

The organization proposed here is a folded FIFO where data travel up one side of the FIFO and down the other. If, at any time on the journey up the FIFO a cell notices that the cell in the corresponding output FIFO is empty, the data jump directly to the output side without traveling the whole length of the FIFO. This organization can be seen in Figure 11. Imagine this eight-deep FIFO as two four-deep FIFOS stacked on top of each other: one moving data from the FIFO input to the right on the top, and the other moving data on the bottom towards the output on the left. The data move into the FIFO on the top and begin traveling to the right. If, at any time during their movement, they see that a cell in the bottom FIFO is empty (and therefore, because of the way things are organized, the other cells between the present cell and the bottom cell are also empty), they jump to the bottom and skip whatever empty cells are between. For example, in an empty FIFO the first word entered in the FIFO will notice that the last cell before the output is empty and so it will jump around the rest of the FIFO and go directly to the output.

In order to do this, the first cell needs to look at the corresponding cell in the output side to see if it is empty. Furthermore, it needs to know that there are no data currently in the FIFO between it and the empty cell it wants to occupy so that it doesn't jump ahead of data already in the FIFO and violate FIFO order. In order to know this, the *full/empty* status of the FIFO is passed in daisy-chain fashion along both the top and bottom rows using the full-in (*FI*) and full-out (*FO*) signals shown in Figure 11. The top row signal is true if there are any full cells in the top row to the right of the current cell, and the bottom row signal is true if that cell is full or if there are any full cells to the right of the current cell in the bottom row. Data in the top row must not jump down to the bottom row if either of these *full* indicators is true. Fortunately, when data are removed from a self-timed FIFO, both cells that cooperate in the transfer of data to the output indicate full status until the data are successfully transferred. This means that a daisy-chained *full* indicator can be built that will not glitch the *full/empty* status when data are being removed from the FIFO. This feature is similarly exploited in the register scoreboard FIFO of the Amulet microprocessor [7], a micropipelined version of an ARM microprocessor.

In order to do this, two main types of FIFO cell are needed. The first is the top-cell which is on the input side of the FIFO, and in the top half of Figure 11. This cell uses a form of arbiter to sense an incoming level signal from the bottom cell and decide whether that cell is full. The input data for this cell always come into the cell from the left, and will then be passed out either to the bottom if none of the cells further on in the FIFO are full, or to the right if this is not the case. The circuit for the FIFO cell is shown in Figure 12. The Q-Select is an arbited Select element that samples the *full* indicators and makes the choice of which way to pass the data. A Q-Select is a Select element where the *SEL* input is not required to be bundled with respect to the transition input. The circuit can be built using either an arbiter, or a Q-flop [8] to make the choice. The Q-select used in the FPGA version of the FIFO uses an approximation of a Q-flop that simply waits for a fixed time to allow the flip flip to settle. The CMOS version uses an arbiter built using a simple four-phase mutual exclusion element added to the ITD standard cell library.

Note that the choice of when to jump may either be made before the data are latched in the current FIFO cell in the top row as shown in Figure 12, or after the data are latched in the top cell. The advantage of the former is that the latency from data being entered in the FIFO to the time when they are available at the output

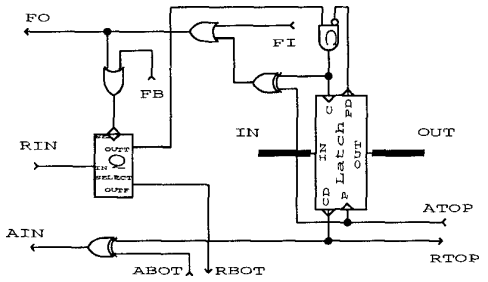

Figure 12. A Top-Row Input Cell for an Arbited FIFO

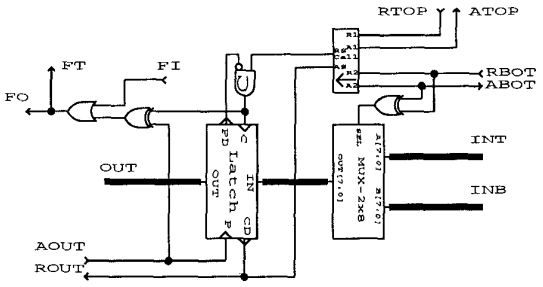

Figure 13. A Bottom-Row Output Cell for an Arbited FIFO

is minimized. In the second case, the latency is slightly longer (the data go through two latches rather than one in the best case), but the throughput may be higher because the acknowledgment from entering the data in the top row comes faster. This circuit is constructed using the same elements shown in Figure 12 but by moving the Q-select circuit to the right of the latch.

In the bottom row, cells must either take data from the right (remember that the data make a U-turn at the end of the FIFO and are now traveling right to left in the bottom part of the FIFO), or from the top to allow data to bypass the empty cells. If everything is working correctly, the choice will be mutually exclusive. This is because a top cell will never send a word to the bottom row unless there is nothing in the rest of the FIFO cells between them. A circuit is shown in Figure 13. Either side may request to enter data into the cell's latch through the Call element. Note that this requires a mux because there are two possible data paths, and so requires another use of the XOR trick to generate the mux signal. In this case, the left channel changes the mux on a request and resets the mux when it acks.

The next feature that is required in this circuit is a full-detector. The full condition is tested easily enough by using an XOR on the *C* (capture) and *P* (pass) inputs to the latch as seen in the figure. Recall that these signals are transition signals. If the capture and pass signals are at different levels, then the latch is opaque and the FIFO cell is full. However, it is not enough to know whether this individual cell is full. You really need to know if *any* of the cells in the row are full. To do this, a distributed full-detector is built using an XOR/OR combination. The cell is full if it is full, or if any cell previous to this cell in the output side of the FIFO is full. This full detector is not delay-insensitive. Delays on the daisy-chained *full* detector can, of course, cause false readings. However, the protocol used with these two-phase micropipelines is such that as data are passed from stage to stage, both stages will consider themselves full for a time before the earlier stage finally goes to empty. Thus, with a little care, a distributed *full* detector should be fairly easy to implement in the style shown here.

Table 1. Comparison of Sixteen-Deep FIFOs using Actel FPGAs

FIFO Type	# of Gates	% Size Increase	Latency (Gates)	% Latency (Gates)
Linear	258	0%	48	100%
Parallel	292	13%	20	42%
Tree	312	21%	27	56%
Square	303	17%	32-27	67%
Arbited	368	43%	14	29%

These blocks of arbited FIFO cells may be cascaded into a FIFO of any desired length as shown in the eight-deep FIFO in Figure 11 The latency of the arbited FIFO depends on the average amount of data present in the FIFO. In an empty FIFO, the data must pass through either exactly one or exactly two cells from input to output depending on the organization of the top cell. As the FIFO begins to fill, the latency grows, but will grow only in response to the FIFO filling and not with the overall size of the FIFO.

8 Performance Comparison

In order to evaluate the performance of the various FIFO organizations presented here, circuits were designed and simulated using an Actel FPGA gate library [1] and a library of self-timed cells for Actel [2]. The circuits were drawn using ViewDraw, and simulated using ViewSim. Although the number of gates is heavily dependent on the particular technology chosen, they are presented in Table 1 for comparison's sake. The numbers in the first column are the total number of Actel basic macros used to build a sixteen-deep eight bit wide FIFO. The Percent Size Increase is a measure of how many more modules that FIFO used than a simple linear FIFO. These numbers and overheads will certainly change depending on the choice of technology. The Actel circuits, for example, use extra macros to take care of unknown bundling delays caused by routing and other artifacts of their technology. The arbited FIFO is probably the hardest of all to compare using FPGAs since the Actel approximation of the Q-Select is very different than a CMOS version. Take the numbers in the table with at least a grain of salt. For this reason, timings are not presented here other than a very simplified measure in terms of number of gates. The number in the Latency column refers to the number of gate delays between input and output for an empty FIFO using the circuits as shown in this paper and with a simplistic count of number of gate delays inside the various control modules. The Percent Latency column gives the latency of that FIFO as related to the latency of a straight flow-through FIFO.

Because they represent the ends of the spectrum of choices presented here, and because the arbiter is particularly difficult to approximate using FPGAs, both the linear and and the folded self-timed flow-through FIFOs were also assembled in CMOS using the ITD CMOS standard cell library (dlmV3.2) [6] and the LagerIV Silicon Assembly System. Each of these circuits are eight-word FIFOs with eight bits per word. The resulting layouts were compared using HSpice assuming a typical MOSIS 2μ CMOS process. The results are summarized in Table 2. There are two measures of interest: the latency between sending an input request (RIN) to the FIFO and the indication that the data are available at the output (ROUT), and also the time between the input request(RIN) and the input acknowledge reporting that the data have been accepted into the FIFO (AIN) which is related to throughput. The figures reported in the the table are the average of the simulated rising-transition time and falling-transition time for an empty FIFO. These figures are reported for the linear FIFO, and for versions of each of the two types of folded FIFOs. Type1 folded FIFO uses the top cell from Figure 12 so that the data go through only a single FIFO stage in the best case (an empty FIFO), and Type2 checks the *full/empty* status after latching the data into the top FIFO and so passes data through two FIFO cells in the best case. Based on these HSpice results, estimates of the latency for a 16-word FIFO of each type are also listed. Note that although the latency increases linearly for the linear FIFO, the latencies for the folded FIFO are independent of the FIFO size and do not change.

The CMOS overhead for using the folded FIFO is also summarized in Table 2 in terms of total transistor count for the eight-word, eight-bit FIFOs using the ITD CMOS standard cell library [6] augmented with my C-element and Mutex cells. Note that depending on the application, the entire FIFO need not consist of arbited

Table 2. FIFO Latency Comparison

FIFO Type	Eight-Word FIFO		16-Word FIFO		Total
	Rin-Rout	Rin-Ain	Rin-Rout	Rin-Ain	Transistors
Linear	53.3ns	6.5ns	107.0ns	6.5ns	1732
Folded, T1	16.9ns	20.8ns	16.9ns	20.8ns	2792
Folded, T2	24.0ns	6.8ns	24.0ns	6.8ns	2896

FIFO cells. A FIFO that has very low latency in the empty or almost empty case, but has lower overhead could be built using some arbited cells near the input and output and smaller linear FIFO cells, or any type of self-timed FIFO for that matter, for the middle portion of the FIFO. As long as the FIFO is nearly empty, data would flow very quickly. If, however, the FIFO began to fill because of different processing rates, the latency would increase due to the linear FIFO section.

9: Conclusions

This paper has explored some circuits for building self-timed flow-through FIFOs that have lower latency than a standard linear micropipeline FIFO. In general these FIFOs distribute data into a number of parallel FIFOs so that data do not have to pass through every FIFO stage on their way from input to output as in the linear case. This reduces the latency, and should also reduce the power consumption as fewer signal transitions are needed to pass data through the FIFO.

The parallel FIFO uses a distribution circuit to control the delivery of data to each of a set of parallel FIFOs, and a corresponding merge circuit to merge the data into a single stream at the output. Parallel FIFOs may be built with any number of FIFOs running in parallel using a generalization of the two-way Toggle into an N-way Toggle. In this FIFO, the latency is reduced by a factor equal to the number of parallel FIFOs the data are distributed into. Of course, as in each of the FIFOs considered, there is latency overhead in the extra circuitry needed to modify the FIFO so maximum theoretical improvements will not be matched in practice. Still, there should be real reductions in latency and power consumption.

The tree FIFO distributes the data into a binary tree of FIFOs, and then merges them back into a single stream in another binary tree. Each tree FIFO cell is a true FIFO cell in that each node in the tree contains storage in addition to the distribution and merging circuitry. Because of the tree structure of this FIFO, latency should be reduced to $O(log(n))$ where n is the total depth of the FIFO rather than $O(n)$ as in the linear case.

The square FIFO is similar to the parallel FIFO in that it distributes data to a set of parallel FIFOs. The difference is that the circuits that do the distribution are themselves FIFO cells. Data are stored in the top and bottom FIFOs as those cells are doing the distribution. The data take an L-shaped path through the square array of FIFO cells so the latency is $O(\sqrt{n})$. This organization may have better layout properties than the tree-structured FIFO although it has slightly higher latency.

The folded FIFO takes a different approach than the others. It is structured as a U-shaped arrangement with data entering the FIFO on top and traveling to the right until it gets to the end of the array where it makes a U-turn and heads back to the left towards the exit. At each cell in the top FIFO, if the bottom FIFO cell is empty, the data may jump directly to the bottom row and skip all the intermediate empty cells. This requires an arbiter in each of the top cells that checks whether the cells between the top cell and bottom cell are empty and thus the jump can be made. This arbiter is required because the status of the bottom cell may be changing as the top cell is checking it. The latency of this FIFO is potentially very low for a mostly-empty FIFO. The average latency will depend on the average number of data words in the FIFO at any time. In the empty case, the data travel through only one or two FIFO cells (depending on which top cell you use).

Of course, combinations of the above FIFOs are possible. The parallel FIFO, for example, might have its parallel arms made up of tree FIFO for even further reductions in latency. Another possibly interesting scheme would be to have a small number of arbited FIFO cells at the front of the FIFO, and some larger, parallel-style FIFO making up the bulk of the FIFO. In this way, an empty FIFO would be very quick, but when it begins to fill, the capacity is increased using another type of FIFO with lower overhead. The optimal arrangement will likely depend on how low the latency must be, and the area and power budget available.

10: Acknowledgments

The ideas for these various FIFO circuits have been cobbled together from discussions with many other people. The parallel and tree FIFOs have their roots in discussions with Ivan Sutherland and Bob Sproull at a time when micropipeline FIFOs were first being developed. The square FIFO grew from an idea of Jo Ebergen's, and the idea for the folded FIFO came from a comment by Al Davis. What I have tried to do in this paper is to give each of these FIFO organizations a realistic circuit rather than just an idea.

References

[1] Actel Corporation. *Actel FPGA Data Book and Design Guide*, 1994.

[2] Erik Brunvand. A cell set for self-timed design using actel FPGAs. Technical Report UUCS–91–013, University of Utah, 1991.

[3] Erik Brunvand. Using FPGAs to implement self-timed systems. *Journal of VLSI Signal Processing*, 6, 1993. Special issue on field programmable logic.

[4] Erik Brunvand. Reduced latency self-timed FIFO circuits. Technical Report UUCS-94-037, University of Utah, 1994.

[5] Erik Brunvand, Nick Michell, and Kent Smith. A comparison of self-timed design using FPGA, CMOS, and GaAs technologies. In *International Conference on Computer Design*, Cambridge, Mass., October 1992.

[6] Scalable CMOS (SCMOS) standard cell library, dlmV2.3, 1990. The Institute for Technology Development, Advanced Microelectronics Division.

[7] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *International Conference on Computer Design*, Cambridge, Mass., October 1992.

[8] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9), Sept 1988.

[9] C. L. Seitz. System timing. In *Mead and Conway, Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[10] Ivan Sutherland. Micropipelines. *CACM*, 32(6), 1989.