Reduced Latency Self-Timed FIFO Circuits

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

UUCS-94-037

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 FIFOs of this type because new data can be sent to the FIFO after communicating locally with only the first element of the FIFO. Therefore the throughput is independent of the depth of the FIFO. 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 report 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 an arbited 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 implementation using a single C-element as the control for each FIFO stage [6]. 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 size of the buffer increases, so does the latency.

In this report, I present some alternative approaches to organizing a self-timed flow-through FIFO. These circuits attempt to retain the good 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, the data must travel though every cell in the FIFO on its way from input to output. In a full case, removing a word means that every word in the FIFO must move up one slot and again every cell in the FIFO must pass data. In a lower-latency FIFO this power savings should be noticeable when the FIFO is nearly empty as the data take a shorter path from input to output, but should also be noticeable when the FIFO is fuller since not all data must move when a single item is removed from the FIFO.

Five different circuits are presented:

Linear FIFO — This is the standard micropipeline FIFO [6]. 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 linear FIFOs, and then collected into a single stream at the output. The input 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.

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.

Arbited FIFO — In this FIFO the data attempt to jump 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 latency of this type of FIFO is dependent on the number of items in the FIFO. The latency is lowest when the FIFO is empty. This should optimize the latency of the FIFO in a way which matches its use. If the FIFO is empty or nearly empty, then the consumer is keeping up with the data generation and low latency is key. If the FIFO is filling up, then the consumer is not keeping up and latency is less important than throughput.

The circuits described here were developed using the Viewlogic schematic capture and simulation tools, and Actel FPGAs as target devices. As such, absolute performance comparisons are not terribly meaningful. Performance results in terms of absolute time are influenced by the speed of the FPGA, and performance in terms of gates delays are influenced by the availability of particular gate types on the FPGA. In particular, the Arbited FIFO requires an arbiter circuit that can only be approximated using the FPGAs so timing based on that approximation is particularly suspect. A more meaningful comparison of custom CMOS layout of each of the circuits would be nice.

2 Linear FIFO

Linear self-timed flow-through FIFO is described in detail in Sutherland's paper [6]. 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.

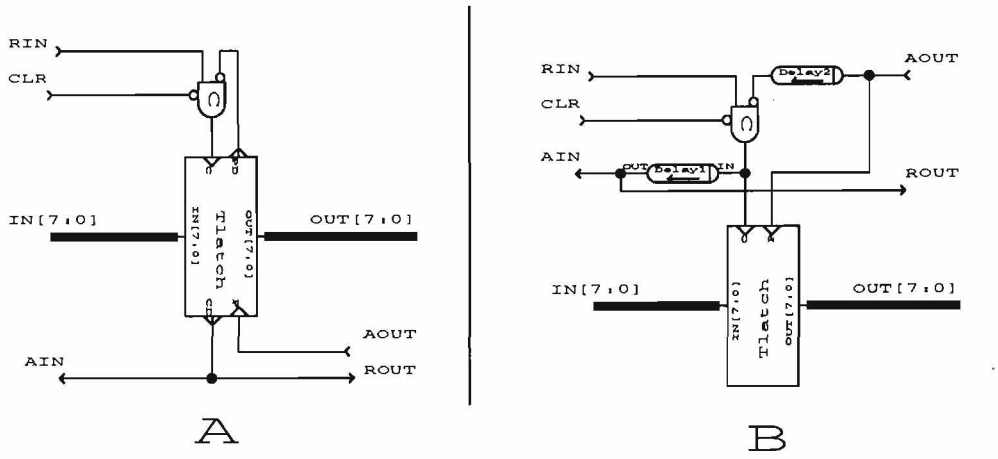

Figure 1: Basic Self-Timed Flow-Through FIFO Stages

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.

One stage of FIFO as described by Sutherland is shown as Figure 1A. The 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 feature is approximated in the Actel version of the FIFO cell by including delays on the *C* and *P* wires to allow the circuit time to operate before the FIFO cell reports completion. This version of the FIFO cell is shown in Figure 1B.

The control for 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 levels, the output is held at its previous level. Note that one input of the C-element used in Figure 1 is inverted. Thus, assuming that all the control signals start low, the leftmost C-element will produce a transition to the capture input of the leftmost latch when the *Req-In* line first goes from low to high. The acknowledge from the latch (either *Cd* or the delayed *C*) will produce a similar request through the next C-element to the right. Meanwhile, the leftmost C-element will not produce another request to the leftmost latch until there are transitions both on *Req-In* (signaling that there are more data to be accepted) and the Ack from the next latch to the right (signaling that the next stage has finished with the current data). Each pipe 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 Ack 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.

A linear flow-through FIFO of length 4 is shown in Figure 2 as four connected copies of the circuit shown in Figure 1B. These four-deep FIFOs are further combined into a 16-deep linear FIFO as shown in Figure 3.

These are the circuits by which the performance of the alternative FIFOs will be judged.

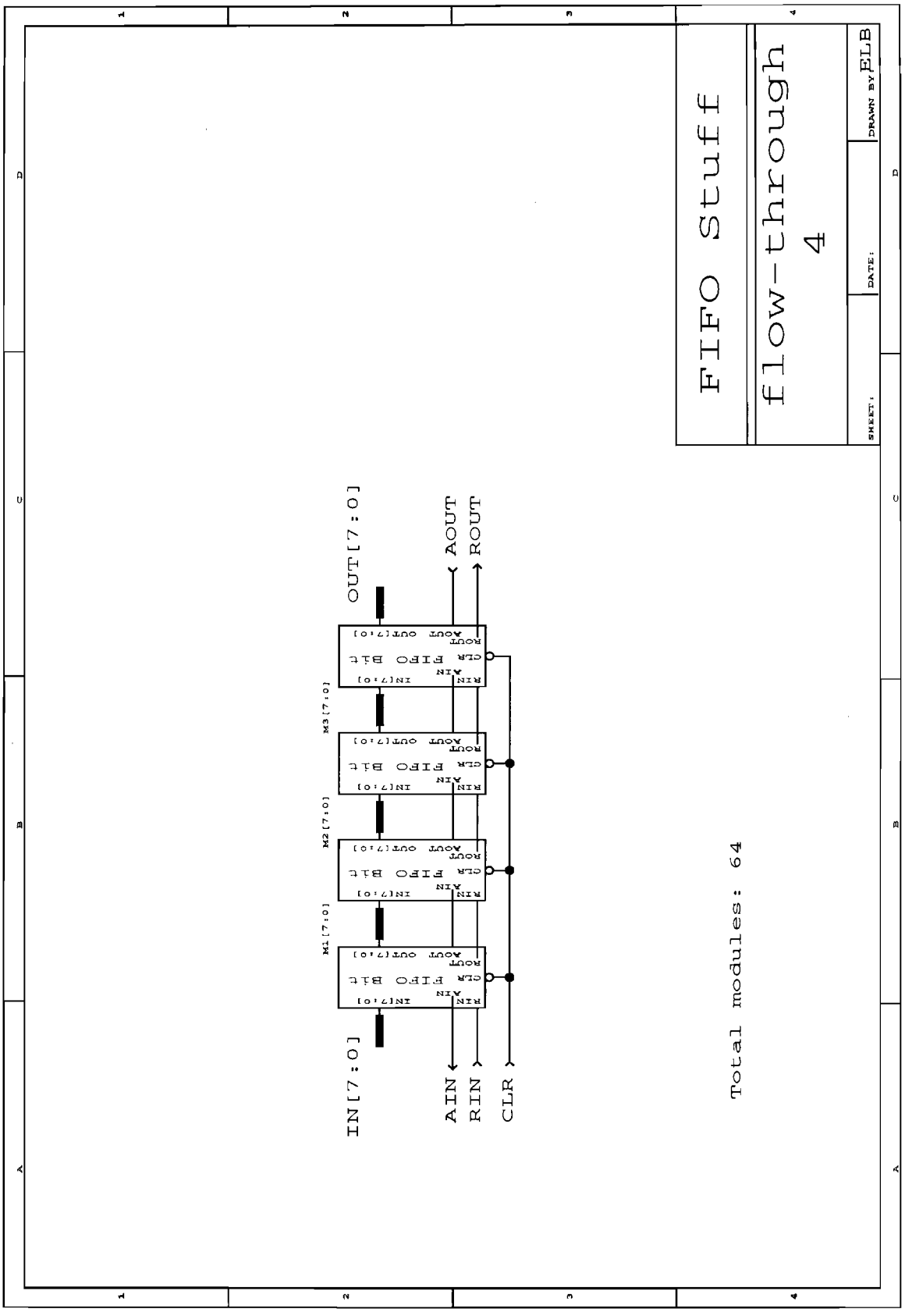

Figure 2: A 4-Deep Linear Flow-Through FIFO

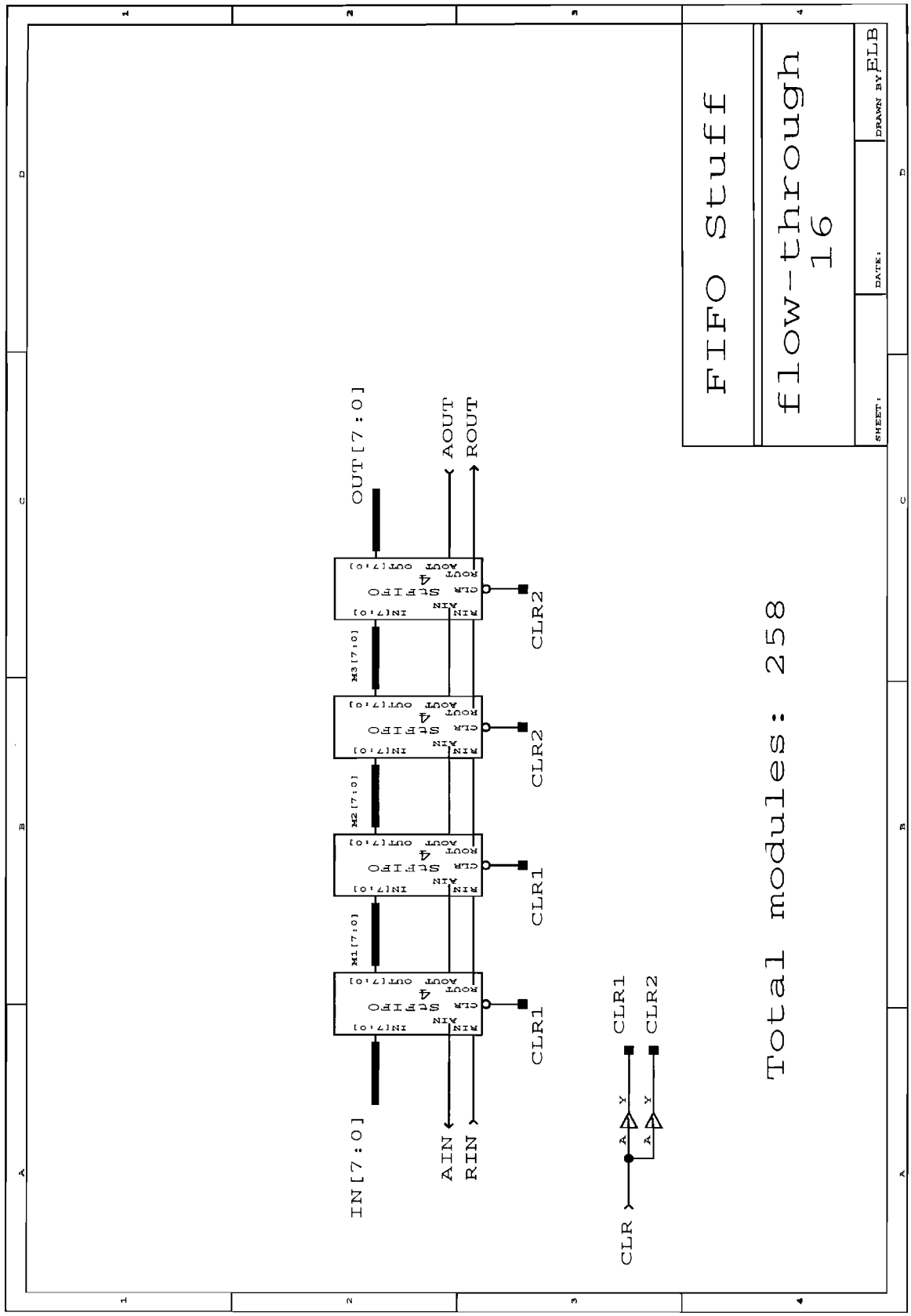

Figure 3: A 16-Deep Linear Flow-Through FIFO

3 Parallel FIFO

In a parallel flow-through FIFO, data are delivered to a number of flow-through 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, splitting the data into four parallel FIFOs, each data word travels through only four cells on the way from input to output reducing latency by approximately a factor of four. 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 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 4 where each of the parallel FIFOs are simple linear FIFOs. The circuit used to distribute the data at the input end of the FIFO is shown in Figure 5, and the circuit used to merge the four streams back into one is shown in Figure 6.

Figure 5 shows the four-way toggle-distribute circuit. This circuit takes the *RIN* (request-in) and sends it 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.

The four-way toggle merge shown in Figure 6 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 (making it the only "half-cocked" C-element in the bunch). 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 to the output based on level control signals. The control signals from the FIFOs, however, are all transition signals. Using a fairly standard trick, an XOR (or in this case, a collection of XORs) is used to manufacture level select signals from 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 have been passed to the output, the *AOUT* routed through the four-way toggle goes to the C-element of the *in1* channel. On the way, it also flips the select lines of the mux to 01 through the XOR network. The next signal (the *out1* signal from the four-way toggle) flips both bits of the mux select and makes the select 10. The next word flips the mux to 11, and finally the next word goes back to 00 and the process repeats. Each *AOUT* that passes through the four-way mux 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.

The performance of the parallel FIFO is determined by the number of parallel data streams. The latency reduction is proportional to that number, with the addition of some constant overhead for the splitting and merging circuits. In the example here with four parallel streams, the latency is reduced

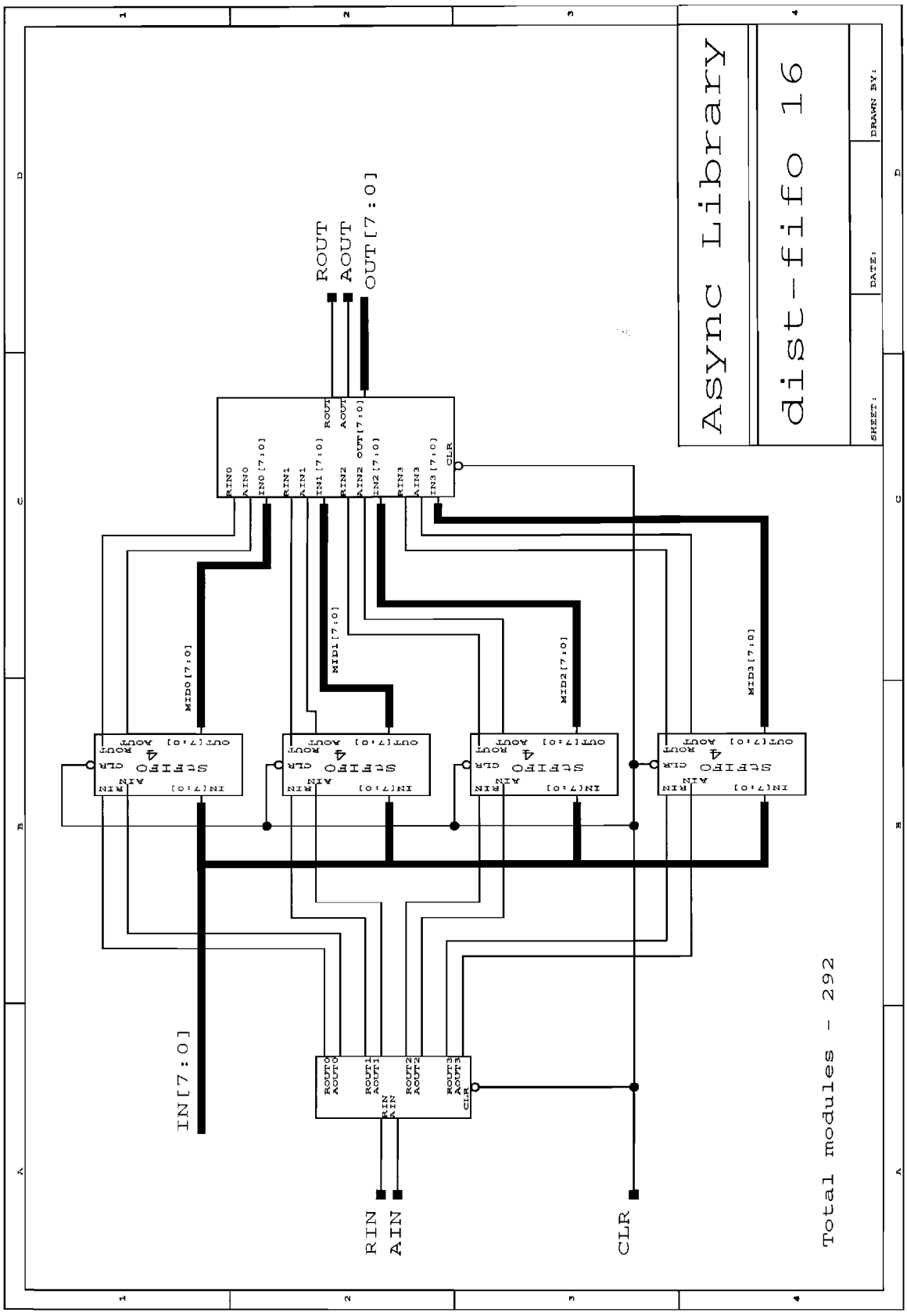

Figure 4: A 16-Deep Four-way Parallel Flow-Through FIFO

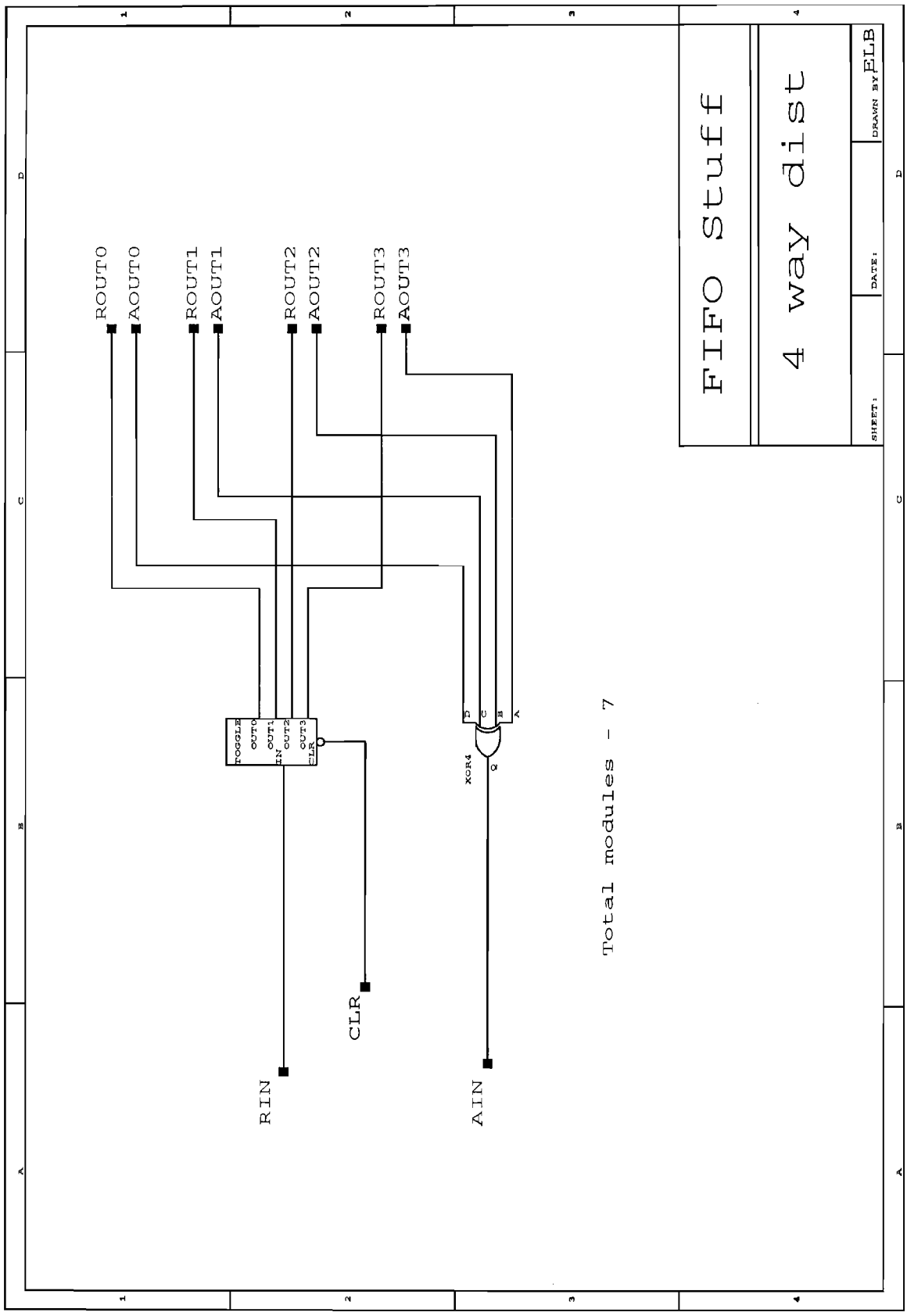

Figure 5: A Four-way Toggle-Distribute Circuit

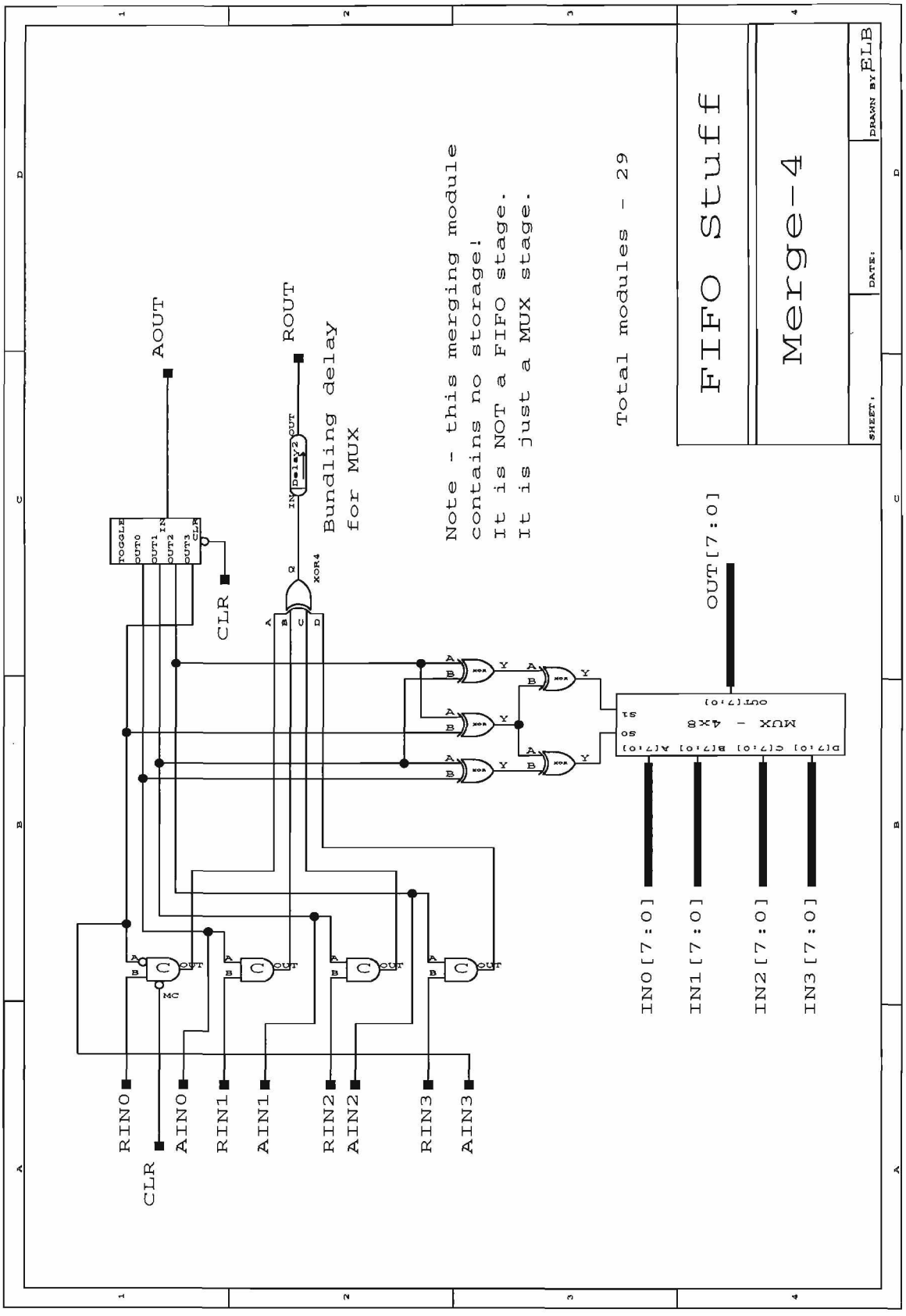

Figure 6: A Four-way Toggle-Merge Circuit

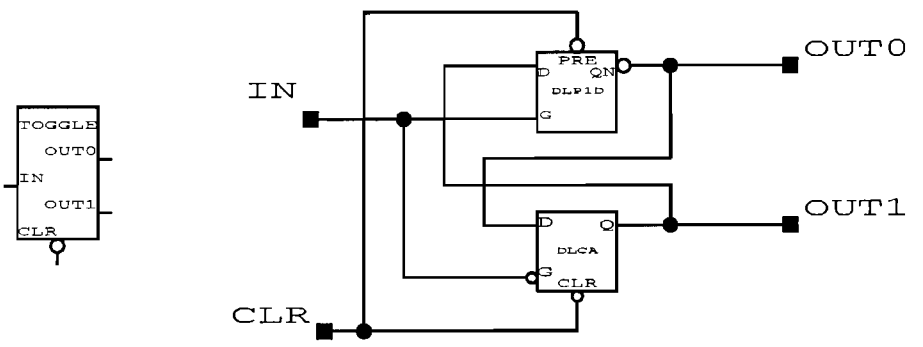

Figure 7: Two-way Toggle Circuit

by a factor of four, not counting the overhead.

3.1 Multi-Way Toggle Circuits

An interesting subcircuit that is required here is the four-way toggle module itself. The toggle module in the standard library [3] is a two-way toggle module. This module, along with its implementation in terms of standard gated latches is shown in Figure 7. The circuits labeled DLCA and DLP1D in the figure are different types of gated latches from the Actel cell library. Because the gate signals on the latches are of opposite polarity, the input transition alternates which latch is transparent and which is opaque. In the state after a master clear, the outputs of both latches is 0, and the transition input is also 0. This means that the bottom latch is transparent and the top latch is opaque. Because of the single inversion in the data stream that is fed between the latches, when the rising input transition causes the gates to be 1, the top latch, now transparent, presents a 1 at the output and at the input of the bottom latch. The bottom latch, which is now opaque, does not yet pass the 1. Succeeding transitions on the input line switch the roles of the top and bottom latches and thus cause the outputs to occur in the proper sequence. Of course, if the delays in the circuit are such that both latches are transparent for a sufficient time, the toggle will cycle through many states instead of making a single state transition. Even though the toggle can be considered a delay-insensitive module at the interface, great care must be taken with the timing internal to the module to keep mistakes from happening.

The parallel FIFO requires a toggle element larger than a simple two-way toggle. In the example presented here, the toggle is a four-way toggle. A four-way toggle can be constructed simply by using a tree of two-way toggles as shown in Figure 8. Transitions on the input cause transitions on each of the toggle outputs starting with *out0* and proceeding in numerical order to *out3* before repeating.

Another technique for building a four-way toggle is to look at the internals of a two-way toggle. A circuit for a four-way toggle may be constructed by cascading the gated latches remembering to alternate the positively and negatively gated latches. As in the two-way toggle, there is a single inversion in the top latch. This implements a four-way toggle in a smaller and quicker circuit and generalizes to a toggle of any even size. A circuit for a four-way toggle built in this way is shown in Figure 9.

An odd-sized toggle must use a slightly different technique. Figure 10 shows one way to accomplish this. This circuit is a four-way toggle where only the first three outputs are presented to the interface of the circuit. The fourth output is fed back to the input through an XOR. Thus, instead of causing a transition at an output, the fourth output is fed back to the circuit input and mimics another

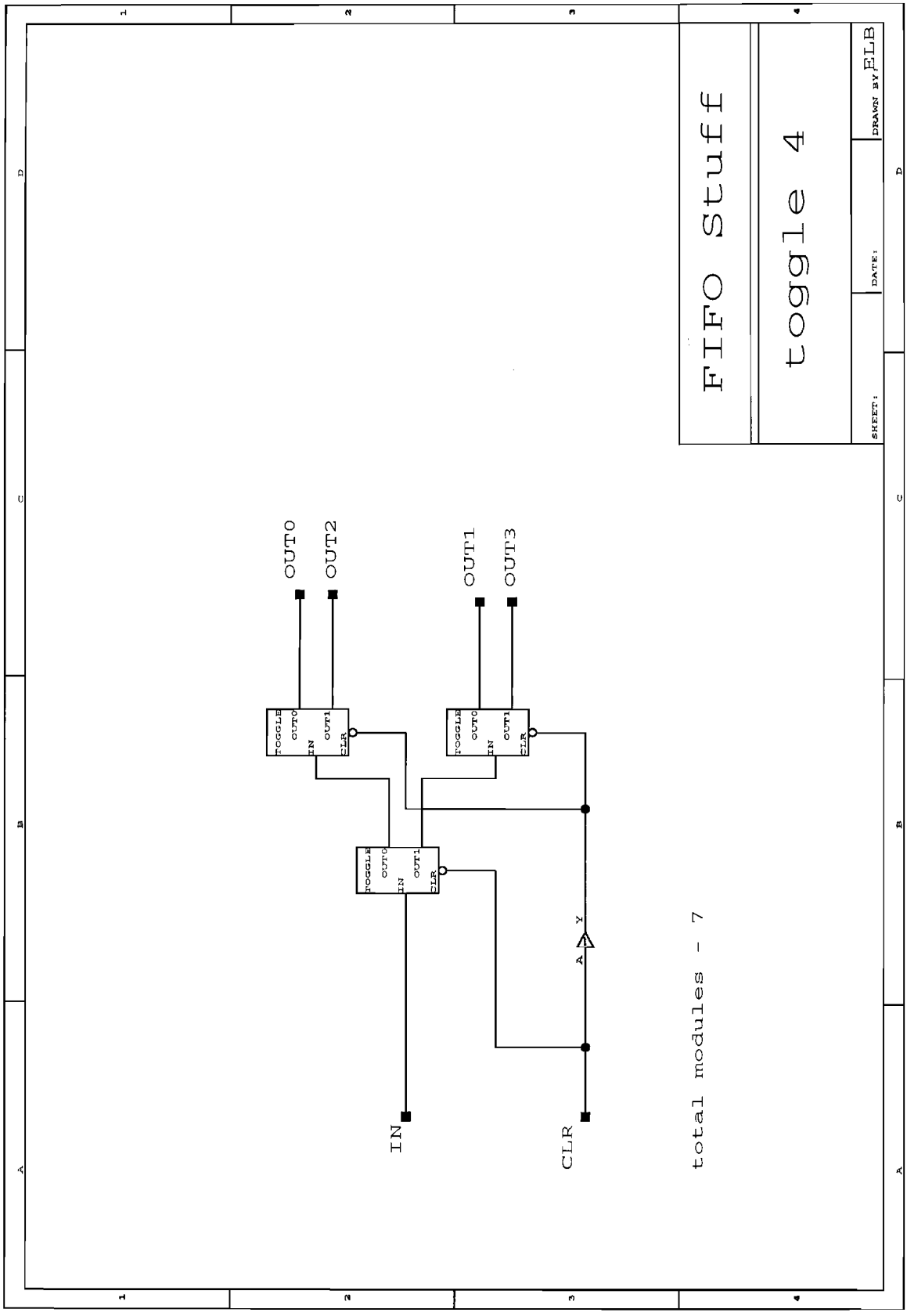

Figure 8: A four-way Cascaded Toggle Circuit

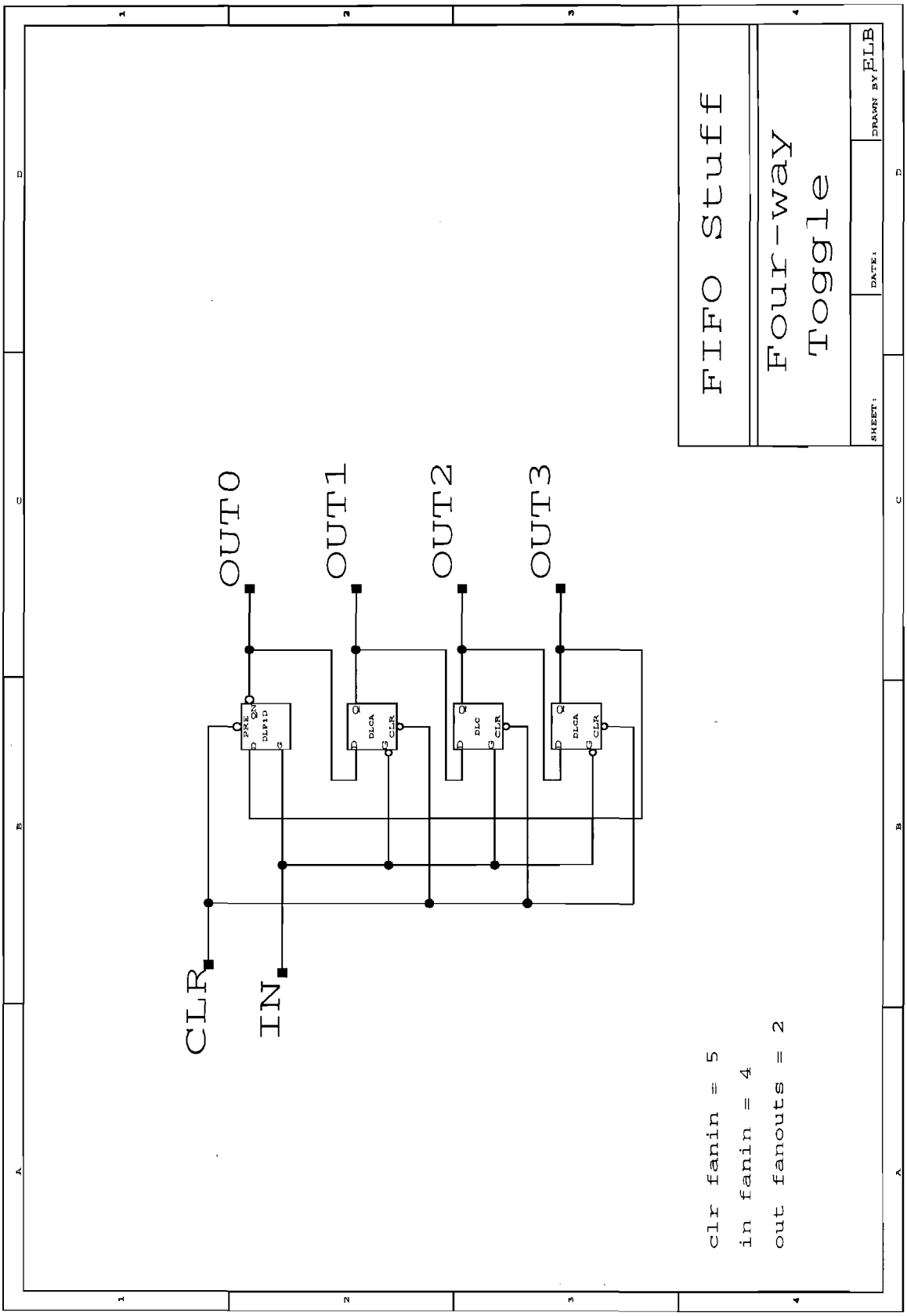

Figure 9: A Four-way Toggle Circuit using Latches

circuit input. The toggle then starts over by causing a transition at output *out0*. The only problem is that the circuit slows down each fourth time you cause a transition at the input since two passes through the circuit are needed to produce the desired output. This technique does, however, scale to a toggle of any odd size.

Another toggle circuit for an odd-sized toggle is possible using C-element instead of gated latches. An example of a three-way toggle built in this way is shown in Figure 11. In this circuit the C-elements, with inputs possibly inverted, serve the same purpose as the alternating positive and negative gates in the gated latch circuits. The inversion in the data path is provided by the top C-element which has an inverted input on the side that the circulating data path enters in. Notice that only the C-elements with one inverted input require a clear input. This is because the transition input to the toggle will be cleared to 0 on a *CLR* along with the other C-elements. Thus, both inputs to the "plain" C-elements will be 0 on a *CLR* and no explicit clearing signal is needed. This circuit generalizes to a toggle of any odd size by repeating the two lower C-elements in the arrangement implied by the figure.

Note than another name for a multi-way toggle circuit as defined here is a "Johnson counter". This is simply a counter that counts in the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000. Thus any circuit that builds this type of counter could be used to build a toggle. The only difference is that the transition toggle form of a Johnson counter counts on both edges of the clock input. Perhaps a toggle circuit could be called a dual-edge-triggered Johnson counter.

4 Tree FIFO

A tree FIFO is essentially a parallel FIFO but where each of the parallel FIFOs are also parallel [5, 2]. In this case, the tree is binary. The data are fanned out to a binary tree of FIFO cells, and then collected in another binary tree to a single output node. Each data word travels through $log(n)$ stages from input to output rather than n stages in the linear case. 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 12 and 13 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 were used only to distribute and merge the control signals. As no data are latched, the 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 12 receives data on a single input line and stores it in a latch. It then delivers that data alternately to two different outputs through a two-way toggle.

The corresponding toggle-merge circuit in Figure 13 is also a FIFO stage. In this case, data words are taken from each of the two input ports in an 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 latch in this figure is a latch-multiplexer combination. 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

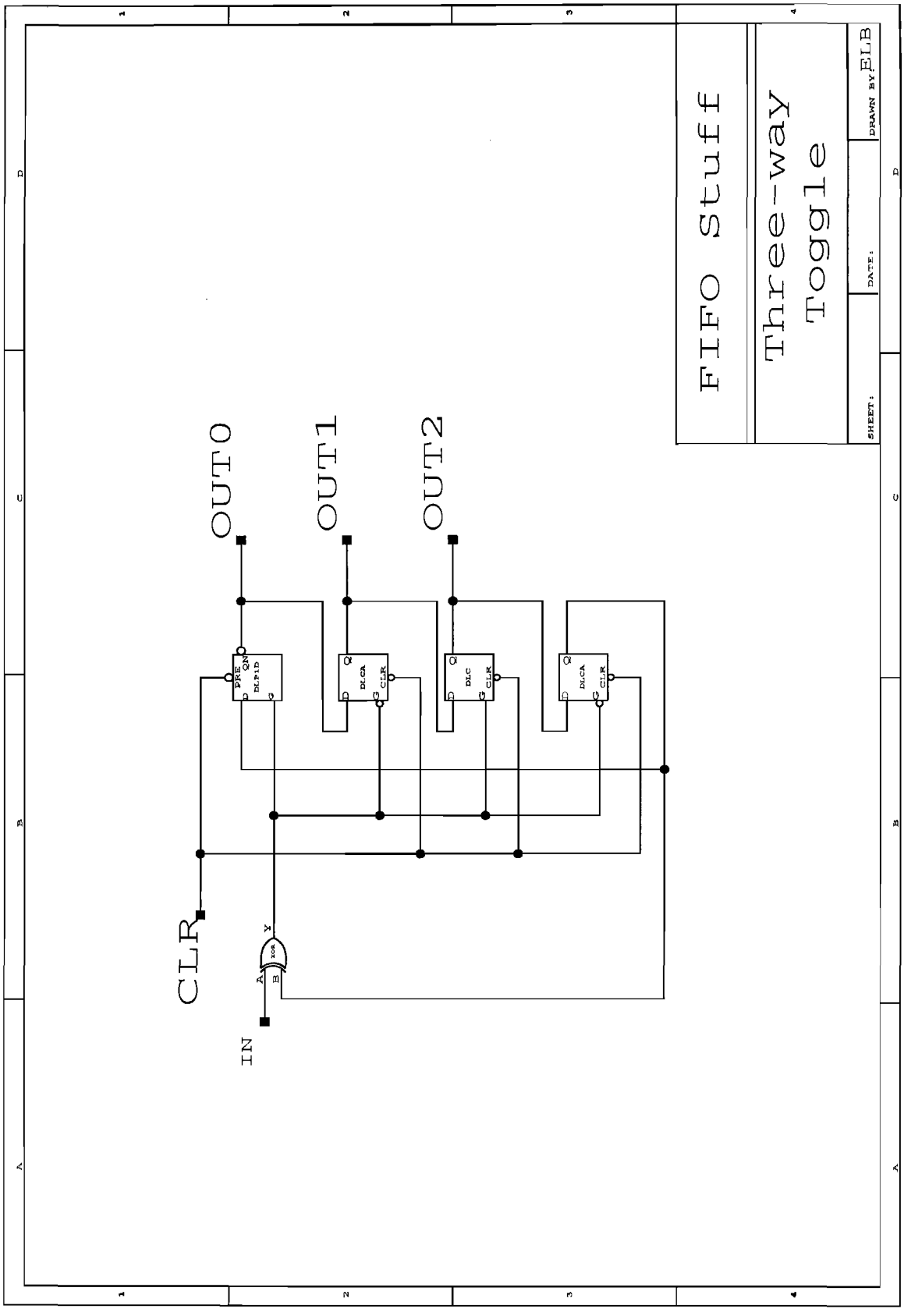

Figure 10: A Three-way Toggle Circuit using Latches

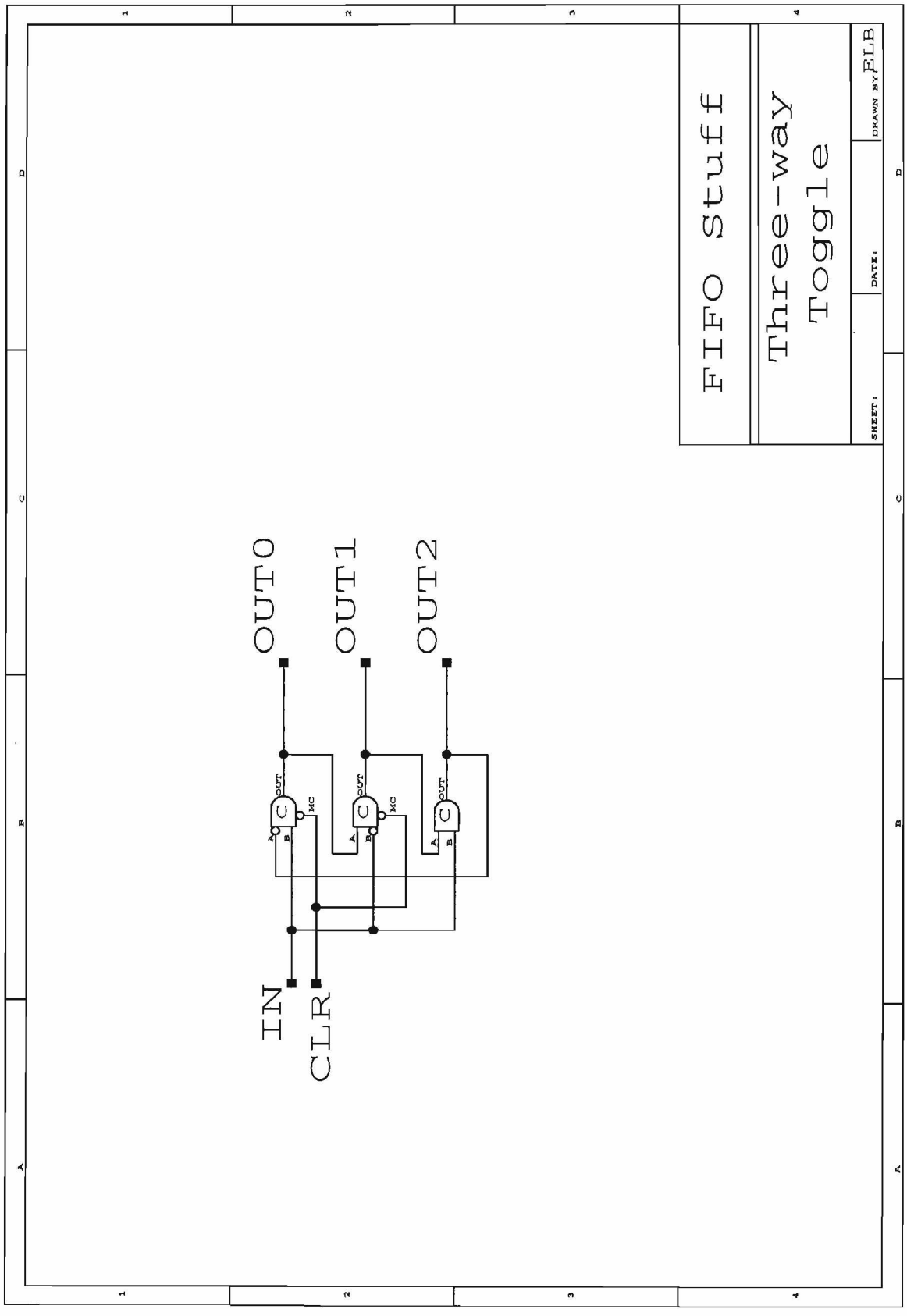

Figure 11: A Three-way Toggle Circuit using C-Elements

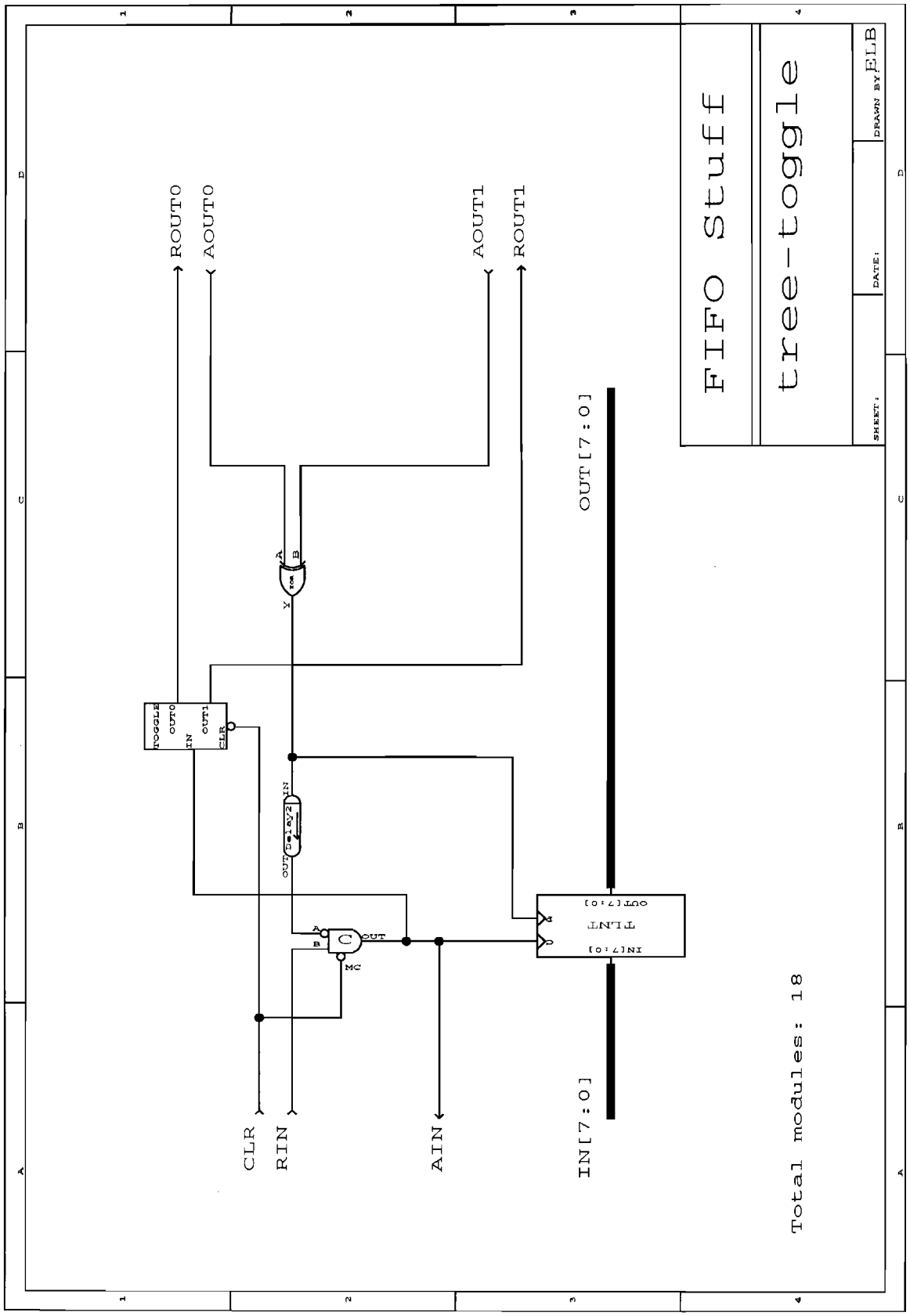

Figure 12: A two-way Toggle-Distribute Circuit

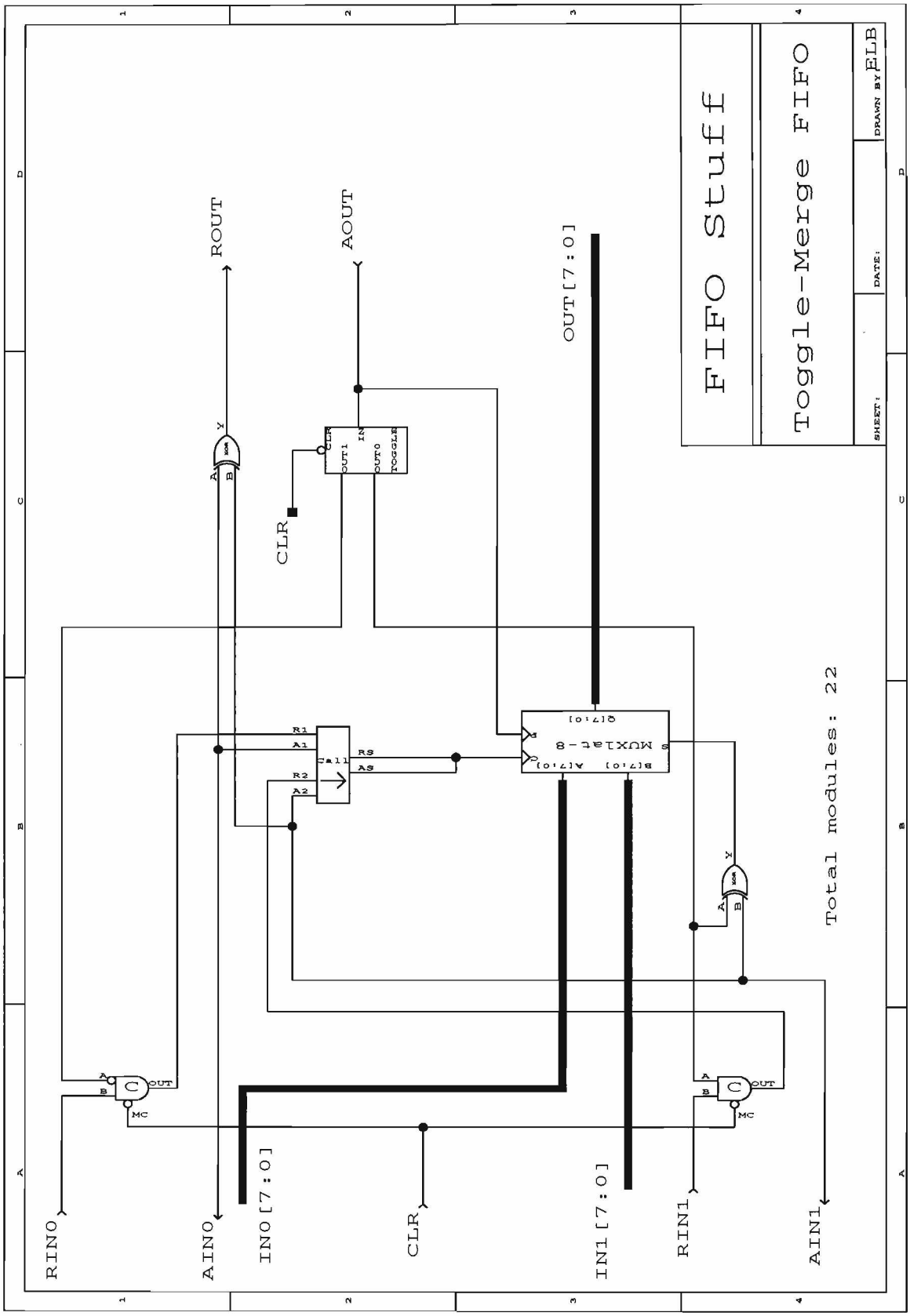

Figure 13: A two-way Toggle-Merge Circuit

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 13 goes through a Call element because either of the two input ports might write into the latch. In the general case, a Call is needed to make sure that the acknowledgment is routed back to the appropriate requester. In this case, however, the access pattern to the Call is known to alternate between the two requesters. Because of this, a toggle-merge combination could also have been used as shown in Figure 14. This circuit has exactly the same functionality as the previous figure, but may have a slightly more efficient circuit realization depending on the chosen implementation technology. In the FPGA gate-level circuits used here, for example, a Call element uses five gates, and a toggle-merge combination uses only three.

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 15 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 as 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 to make tree FIFOs of any size. Using the six-deep tree FIFO as building block a circuit for a 14-deep tree FIFO is shown in Figure 16. This FIFO simply expands the input tree and output tree by one level. The latency of a single data word through the FIFO is six stages rather than the full 14 required by a linear arrangement. In order to compare to the other FIFOs presented here, which are all 16 stages deep, a 16-stage tree FIFO can be constructed by adding simple linear FIFO to the input and output of the 14-deep tree FIFO. This circuit is shown in Figure 17.

Still another possible arrangement where the leaf nodes of the tree are single-stage flow-through FIFO circuits is shown in Figure 18. In this case, the simple-FIFO leaf nodes are shared between the fan-out and fan-in trees. The capacity of this FIFO is ten words although the latency for a single data word is only five stages. In general, the latency of a data word through a tree FIFO is $O(log(n))$ (assuming a binary tree fan-out and fan-in) as opposed to the $O(n)$ latency of a linear flow-through FIFO.

5 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 integrated circuit. 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 linear 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 data that pass through them.

A square FIFO circuit is shown in Figure 19. The intuition for this circuit is that data are entered 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 next word 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

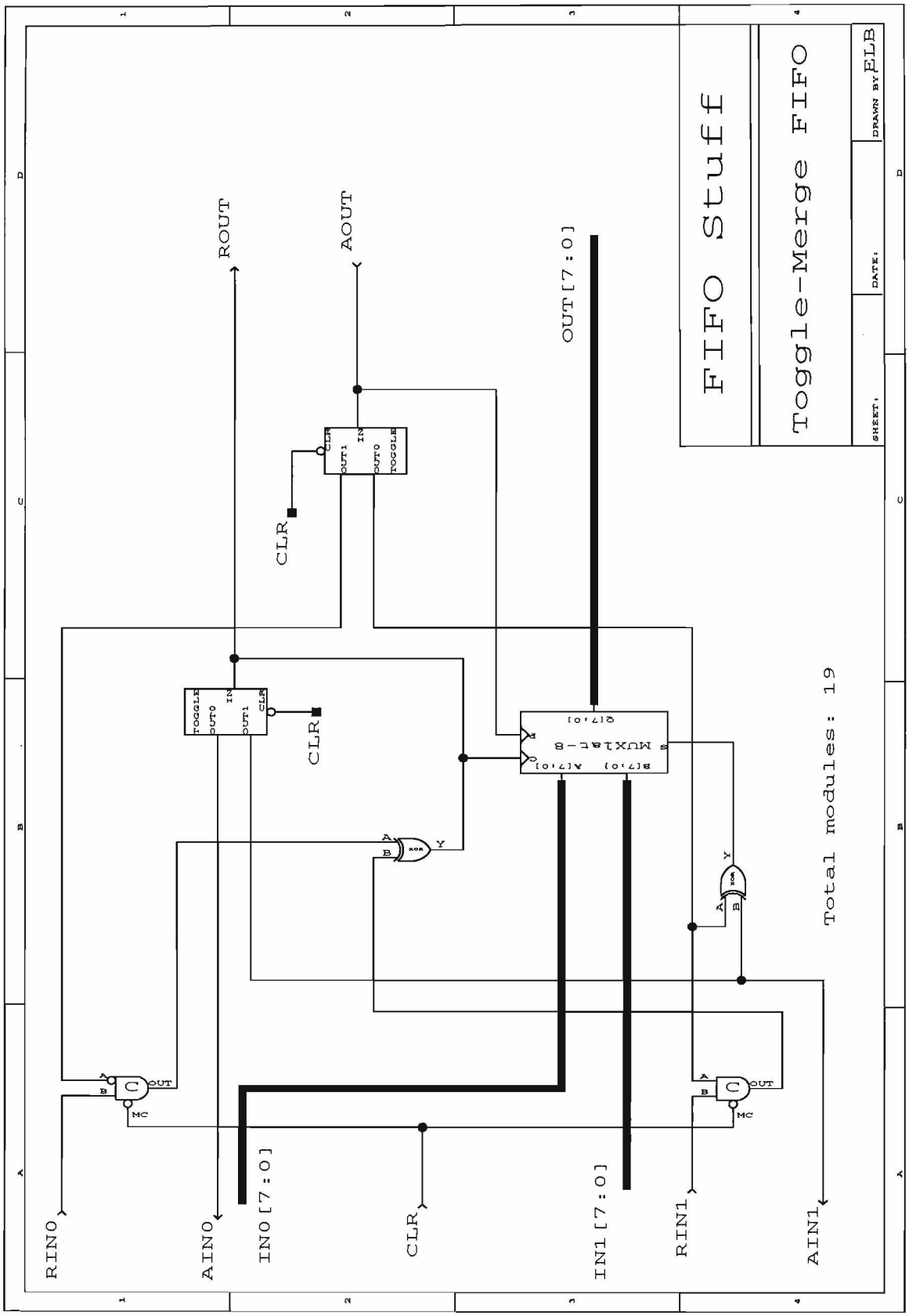

Figure 14: Another two-way Toggle-Merge Circuit

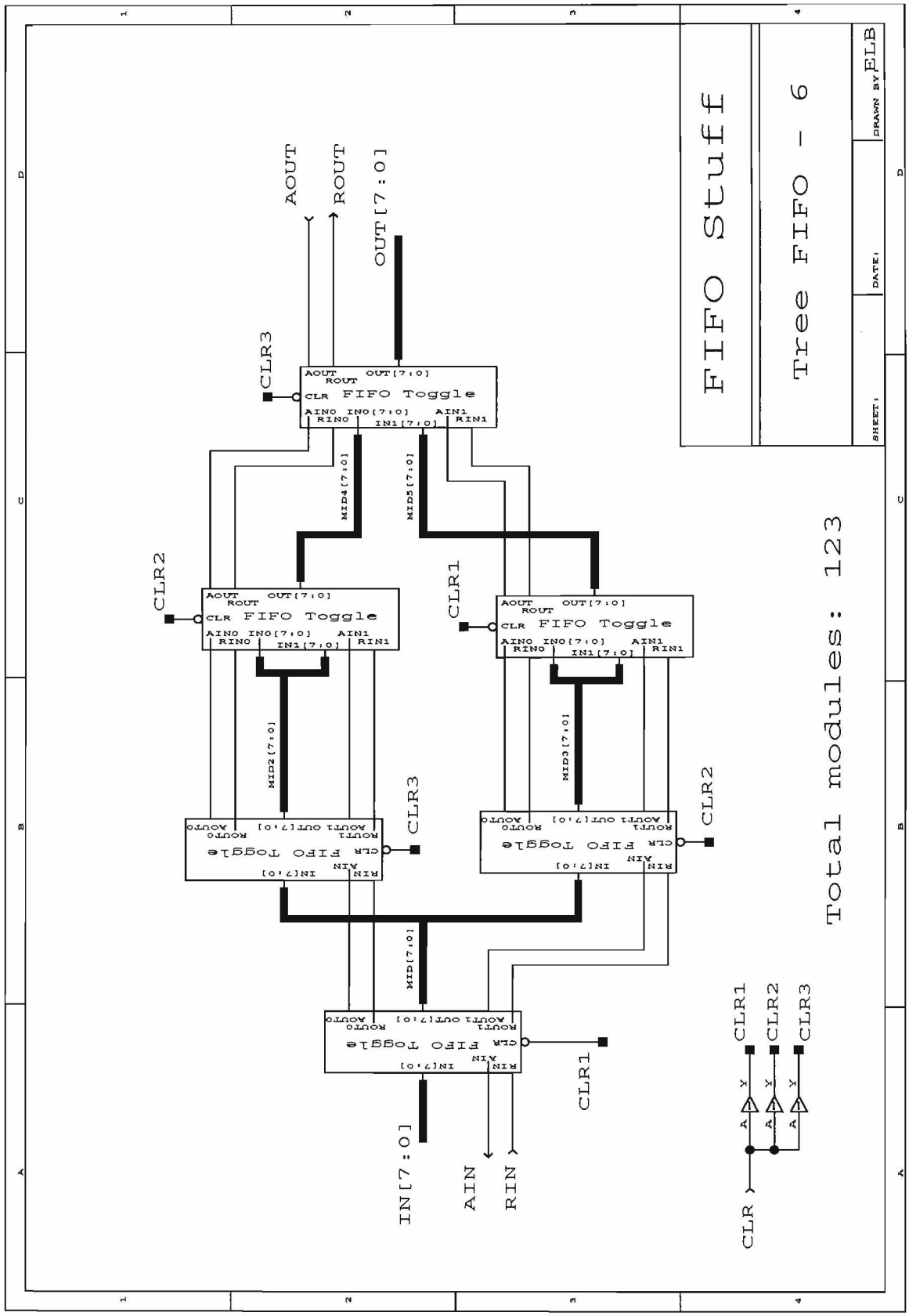

Figure 15: A Six-Deep Tree FIFO

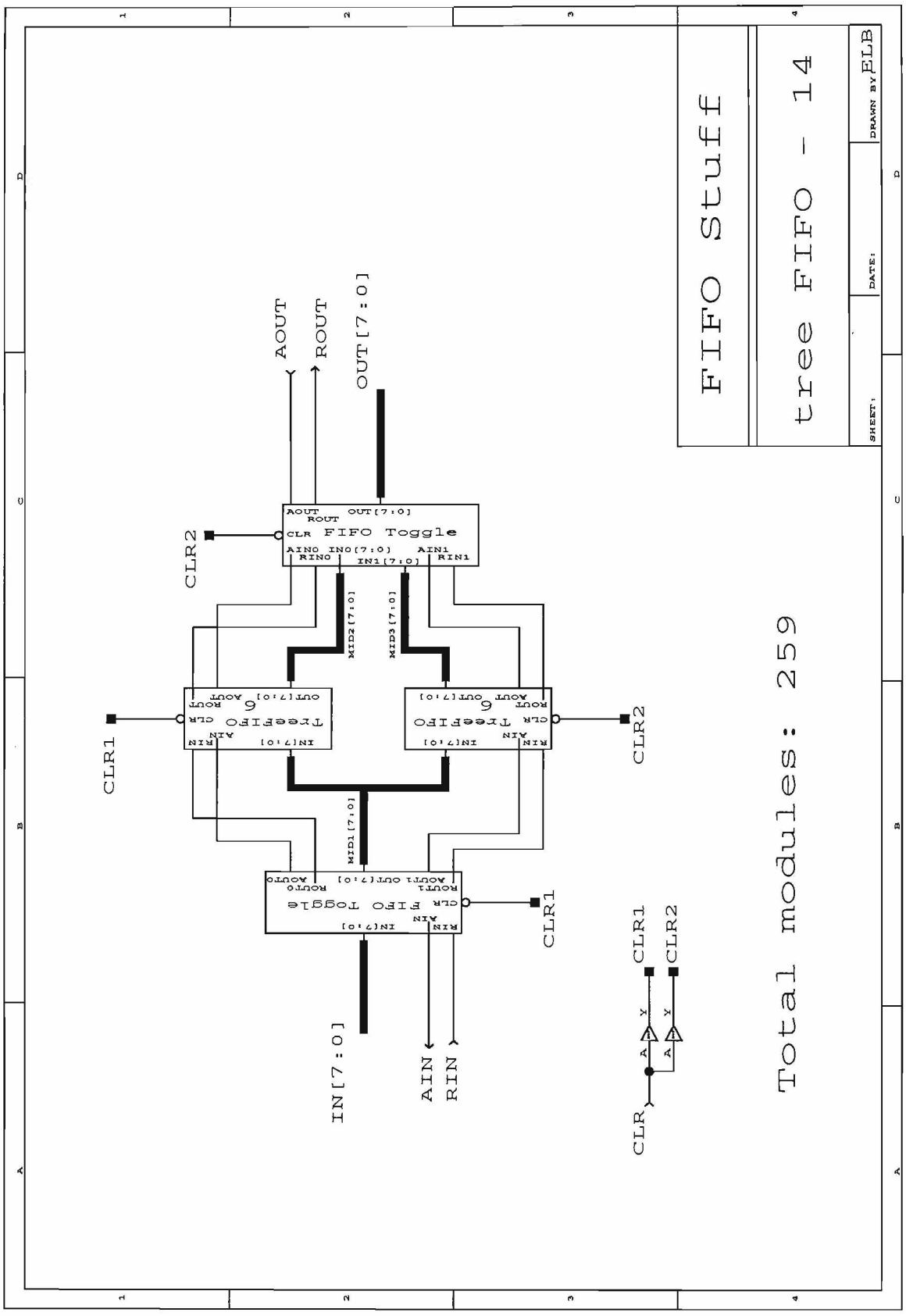

Figure 16: A 14-Deep Tree FIFO

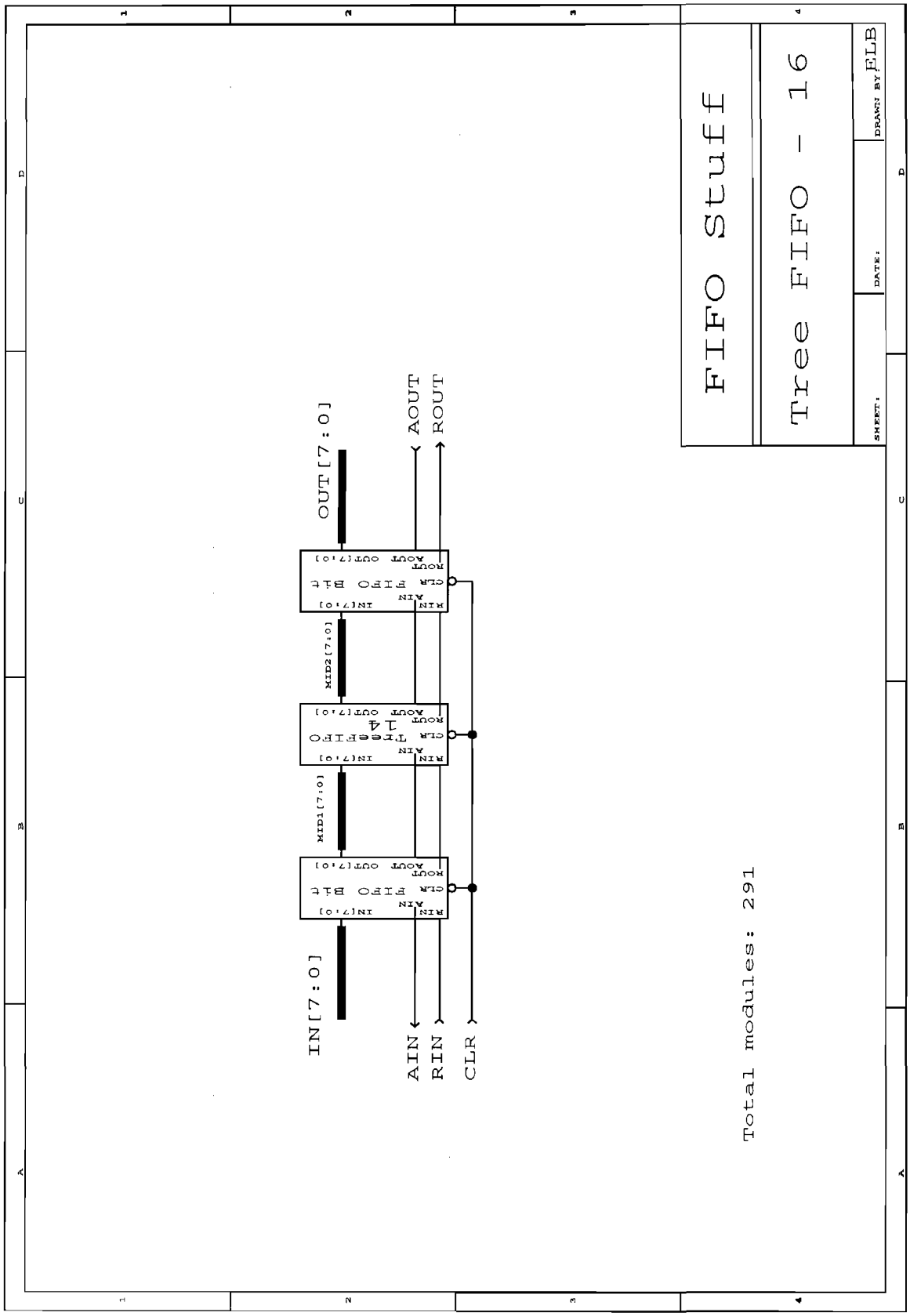

Figure 17: A 16-Deep Tree FIFO

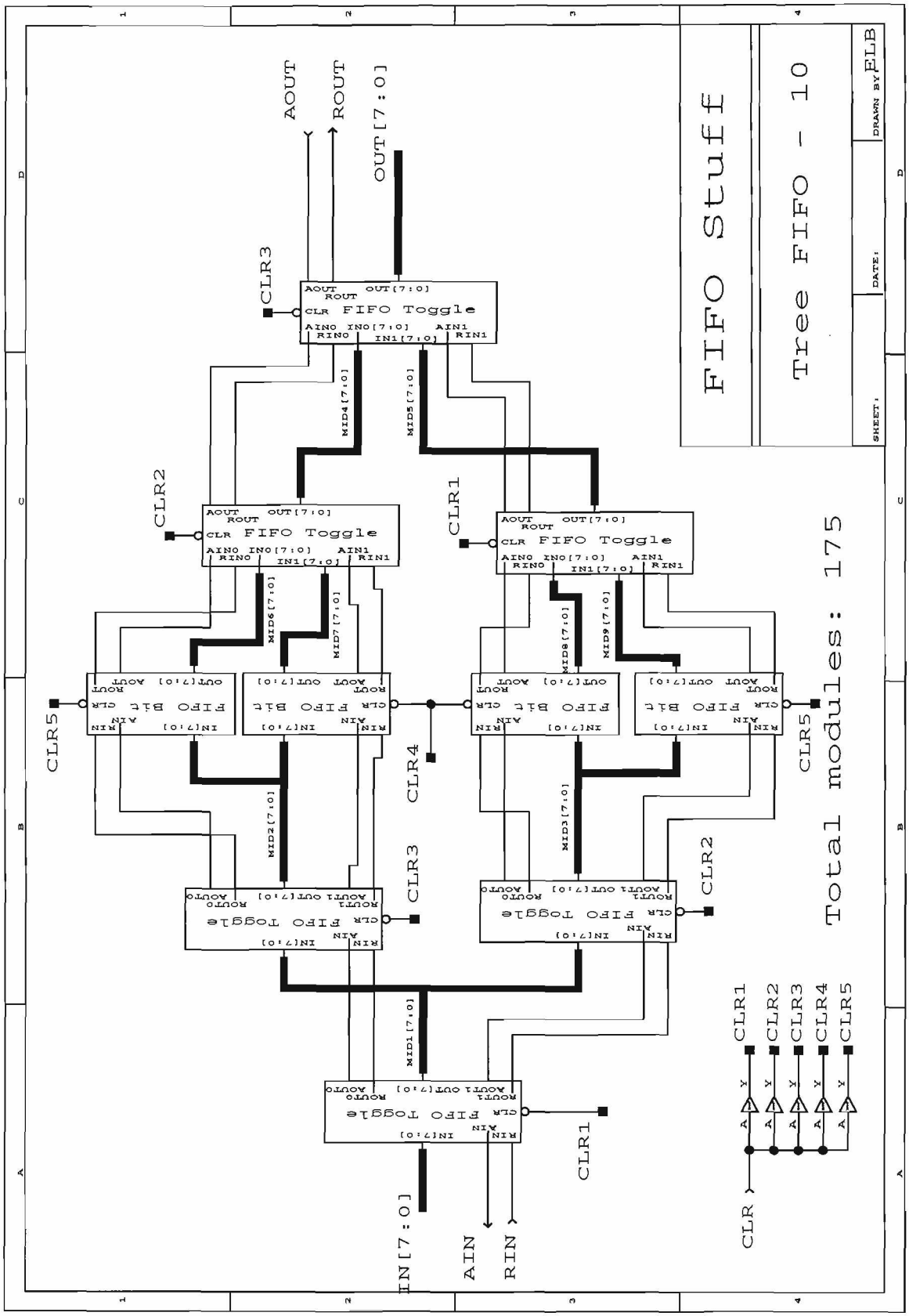

Figure 18: A 10-Deep Tree FIFO

data word comes through, 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 go 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, it 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 linear 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.

Note how this works in the circuit in Figure 20 which is the circuit of the next-to-last cell in the top row. Data are latched into the buffer on receipt of a Req-Left (*RL*) signal. These data are passed alternately to the right and down using the toggle module. The acknowledgments to the left (previous) FIFO stage depend on which direction the outgoing data are sent. If the data are sent to the right, then the next data to be sent from the left should also be to the right. If the data are sent down, then the next data received by the FIFO to the left should be sent down instead of to here. Notice also the *rename* boxes in the figure. These are just wire-renamers that allow the single output bus to have two different names at the module's interface. They add no logic to the circuit and are only a consequence of how Viewlogic treats net names.

The other FIFO cells in the top row obey the following protocol. 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 select signal being 0 as it is after a 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 21. As in the previous circuit, the *rename* modules contain no circuitry and are only there to allow the output bus to have two different labels.

These three cells are combined to form the top row of a square FIFO. The first data make it all the way to the far right. The next 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

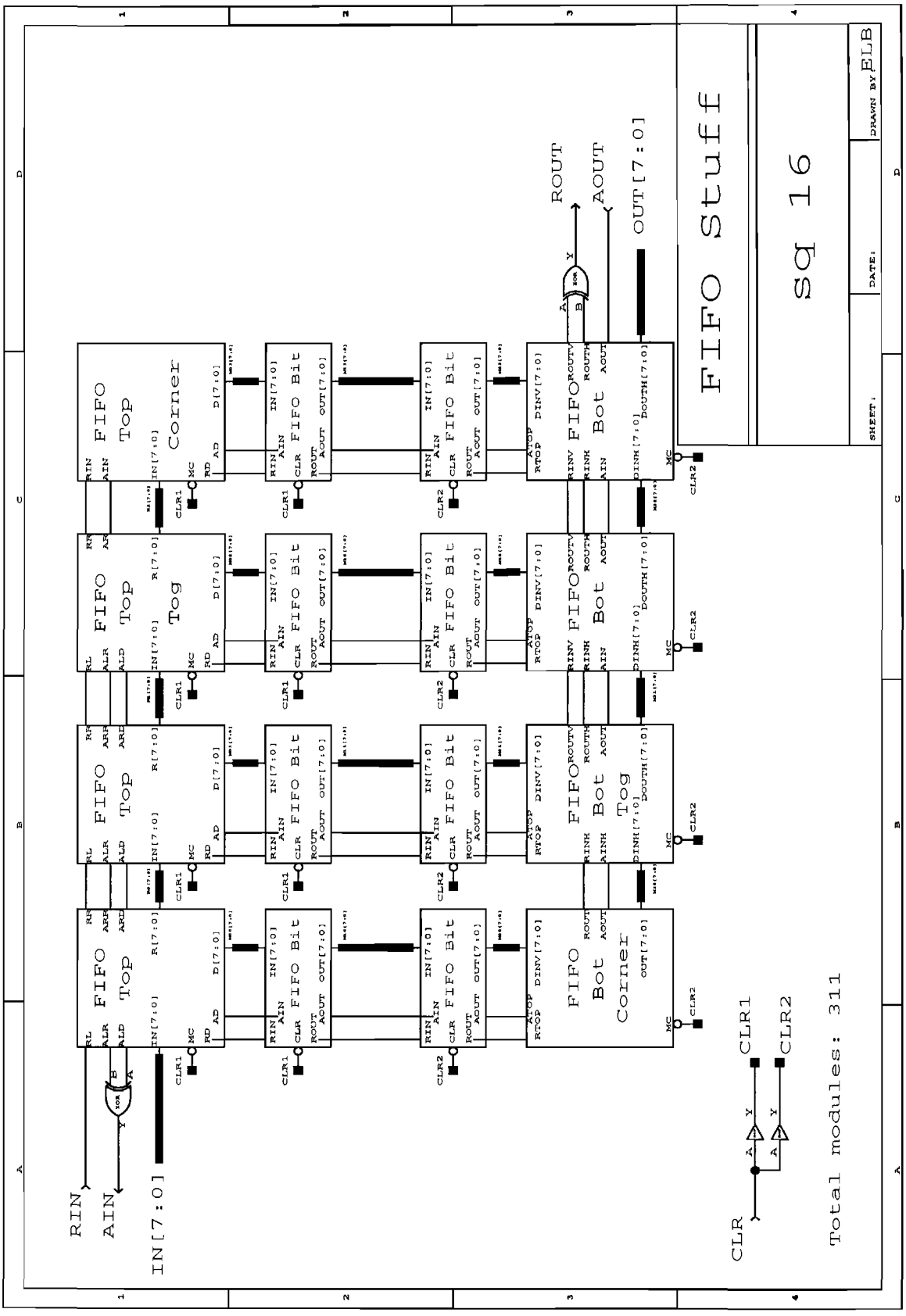

Figure 19: A 16-Deep Square FIFO

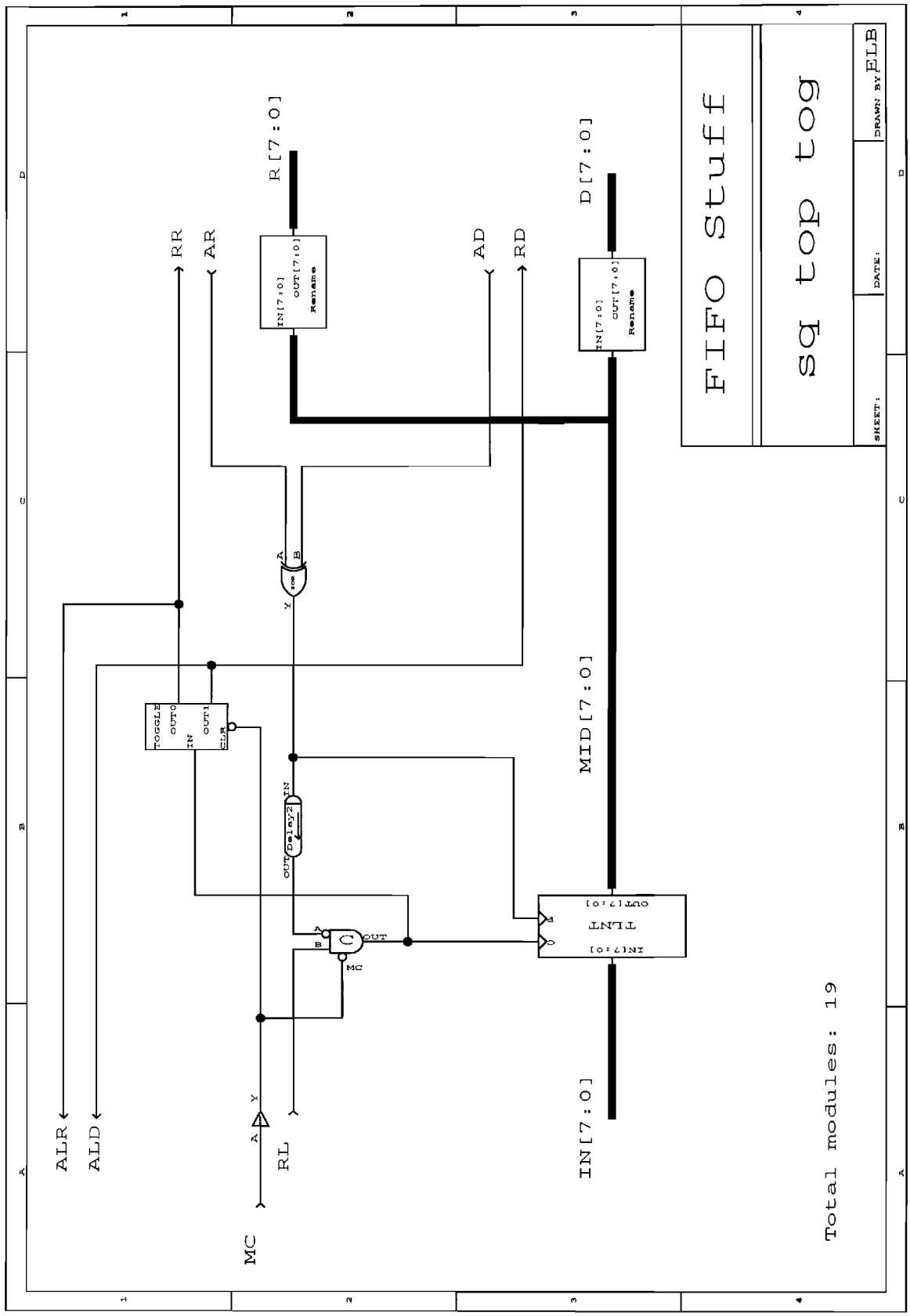

Figure 20: A Toggle FIFO for the Top Row of a Square FIFO

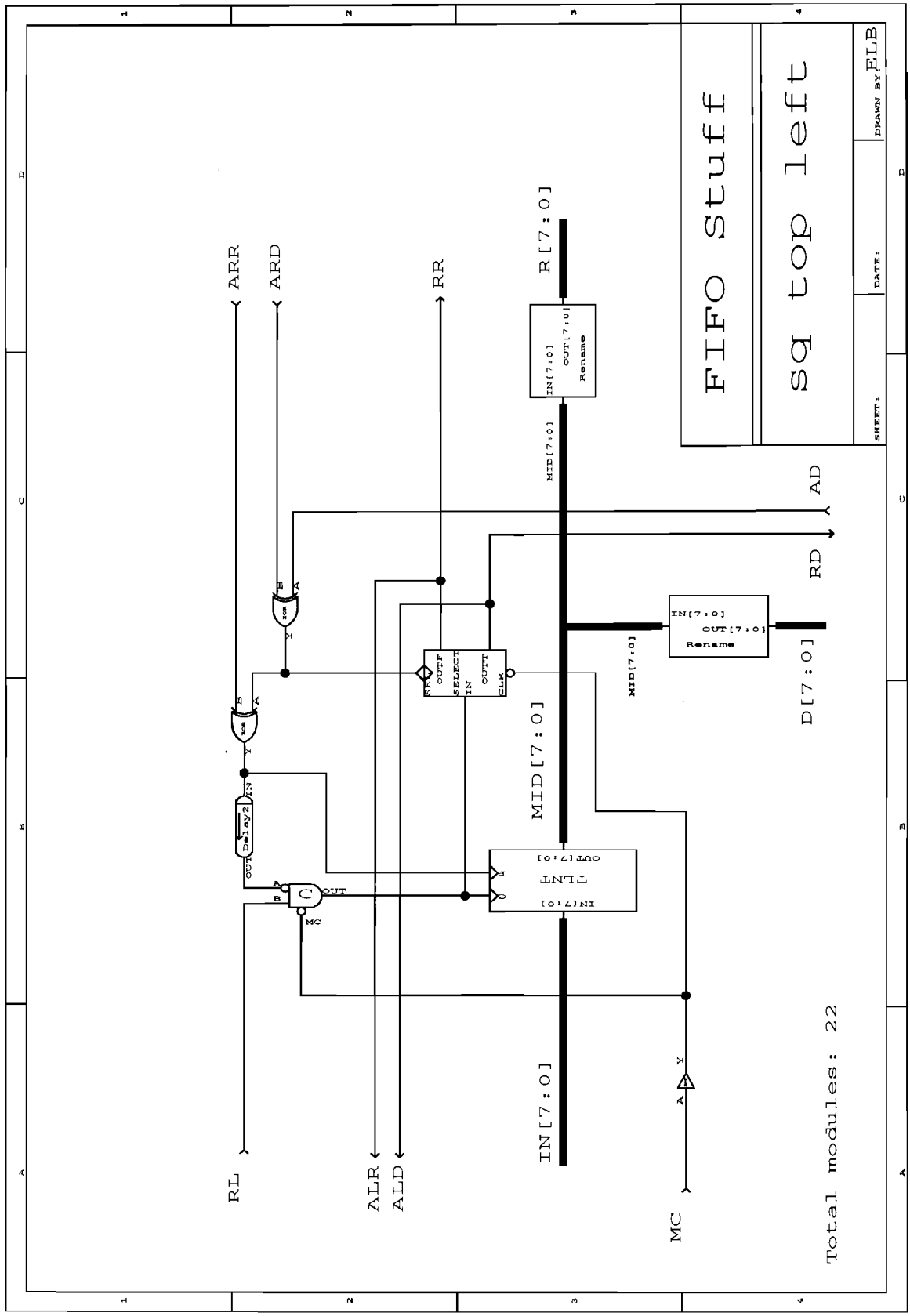

Figure 21: A Row FIFO for the Top Row of a Square FIFO

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 a 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. The only difference is that this toggle-merge FIFO sends one of two possible requests to the right instead of just 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 the toggle case by not combining the outputs of the internal toggle in an XOR, but by letting each of them be a different request output as shown in Figure 22.

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 the behavior is modified the next time data are received. This is what the circuit in Figure 23 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, the data are latched in the mux-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 data 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 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 mux-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 data 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 the current data came from the left. This is why the signal that changes the mux and bottom select come from *ROUTV* rather than from *RINV* as in the top select.

Once the mux and selects are set to choose 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 latching resets the data mux, and both of the selectors, and puts them back in horizontal mode.

These cells, arranged in the organization shown in Figure 19, make a FIFO where each data word

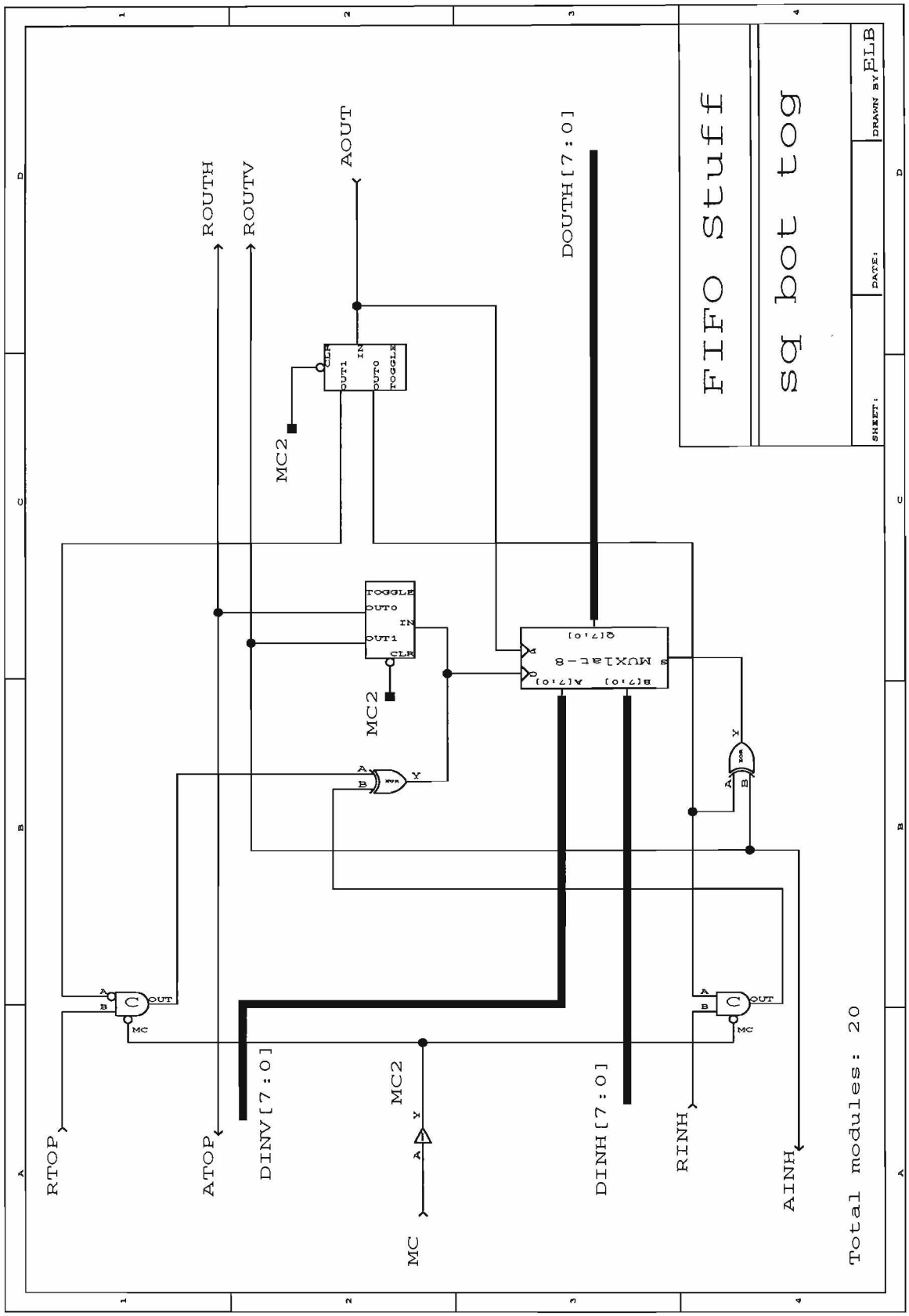

Figure 22: A Toggle-Merge FIFO for a Square FIFO Bottom Row

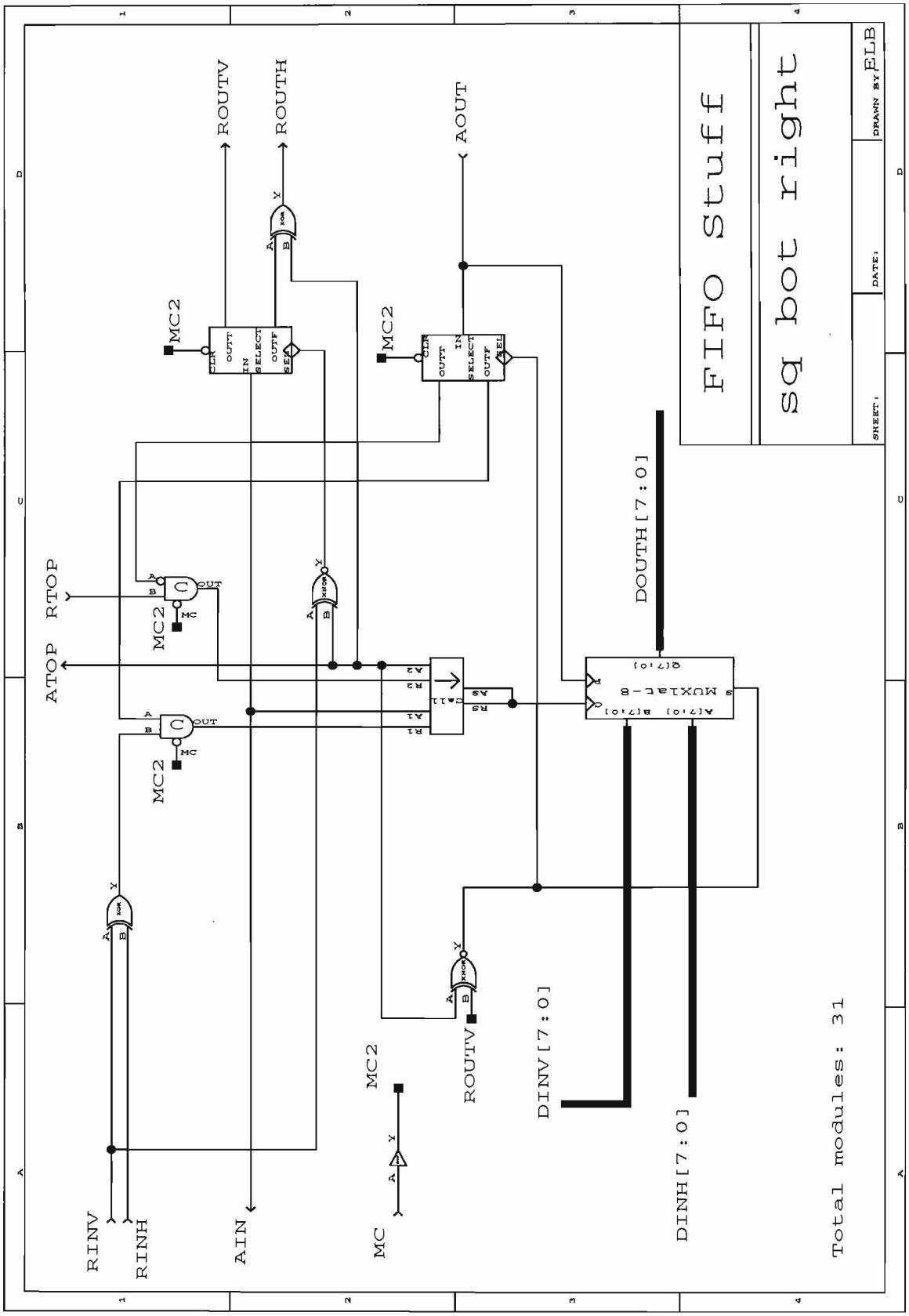

Figure 23: A Bottom-Row FIFO for a Square FIFO

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.

6 Arbited 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 as the condition being checked may be changing at the same time as it is being checked. 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 jumps directly to the output side without traveling the whole length of the FIFO. This organization can be seen in Figure 24. Imagine this 16-deep FIFO as two eight-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 from the FIFO to the output on the left on the bottom. The data move into the FIFO on the top and begin traveling to the right. If, at amy time during their movement, they see that a cell in the bottom FIFO is empty, they jump to the bottom and skip whatever empty cells are between (the data path for this jump from top to bottom is hidden inside the boxes of Figure 24). 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 (do not pass "GO", do not collect $200).

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 in the rest of the FIFO between its current location and the cell that it is about to jump into. This data, if it exists, could be in transit in another part of the FIFO. It may not have jumped to the bottom yet because there was no space in the bottom FIFO when it made its check, and while it is moving forward in the top, data was removed in the bottom thus clearing out space.

In any case, If there were data in the FIFO between the current location and the jump-to location, and the data jumped, then the data would be delivered to the output out of order. In addition to generating a full/empty indicator for each FIFO stage, these indicators must be passed from cell to cell in a way that allows any cell to know the status of *all* other cells between it and the possible short-cut cell in the bottom row.

Generating an full/empty status of an individual cell is accomplished easily using an XOR on the *C* (capture) and *P* (pass) signals of the latch. If those signals are at different levels, the data is captured in the latch and it is full. If they are at the same level, the latch is transparent and the cell is empty. These full/empty indicators can be passed in daisy-chain fashion between the cells in the top row and in the cells in the bottom row. Note that this will *not* be delay-insensitive. The full/empty indicator daisy-chain must be constructed to be fast relative to the data being entered. Fortunately, FIFO cells using C-elements for control have the property that if the FIFO is full, and data are removed, then all cells continue to overlap their full condition as data are being removed. This feature is similar to the condition exploited in the Amulet register file scoreboard FIFO [4].

If all the cells in the output side of the FIFO are full, then the data must travel through the entire FIFO just like a linear FIFO. However, if the FIFO is getting full, that means that the process taking

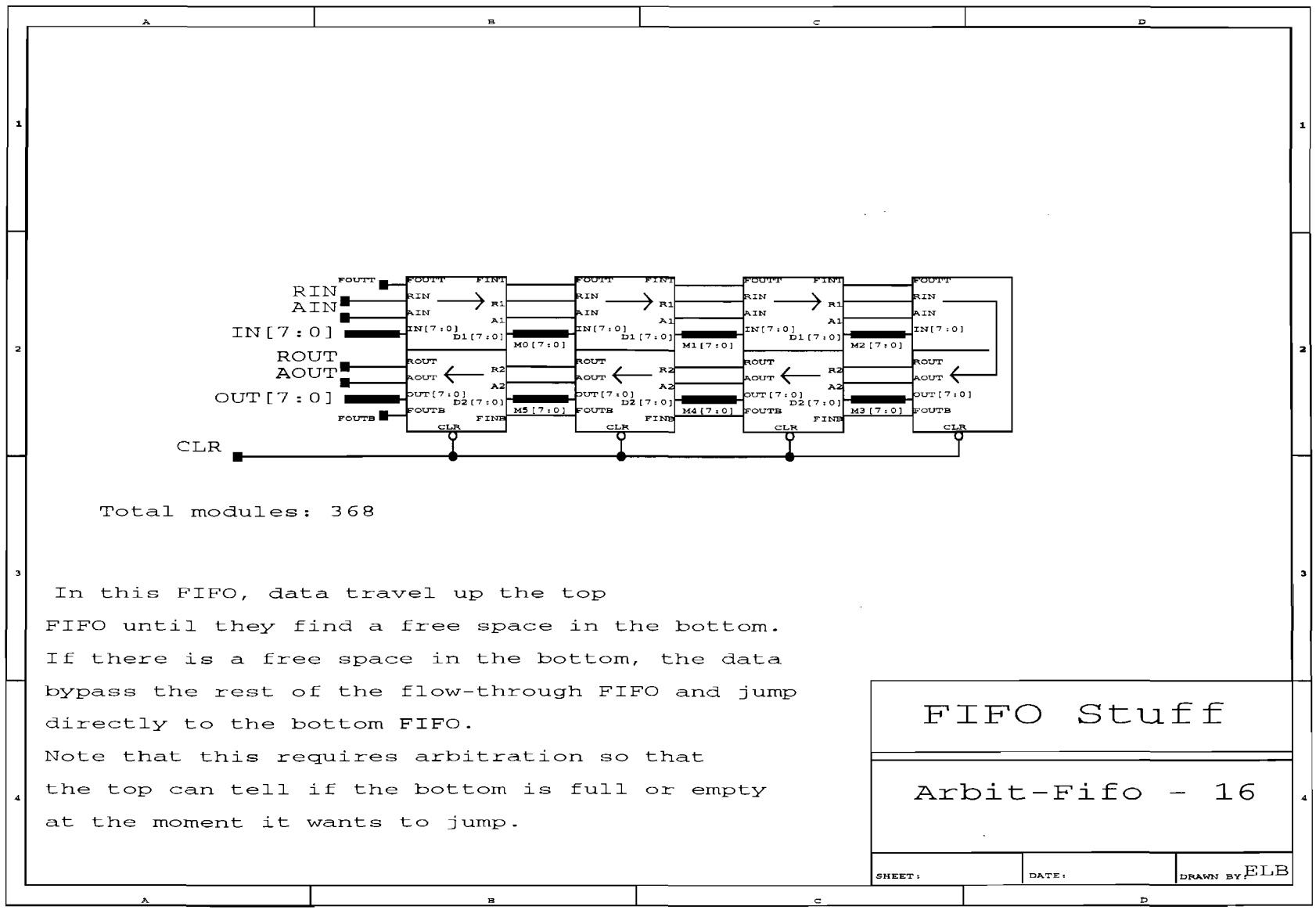

Figure 24: A 16-deep Arbited FIFO

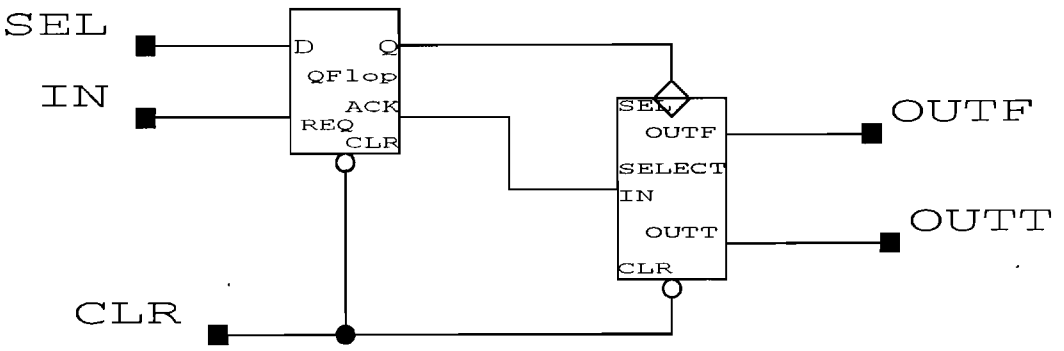

Figure 25: A Q-Select Based on a Q-Flop

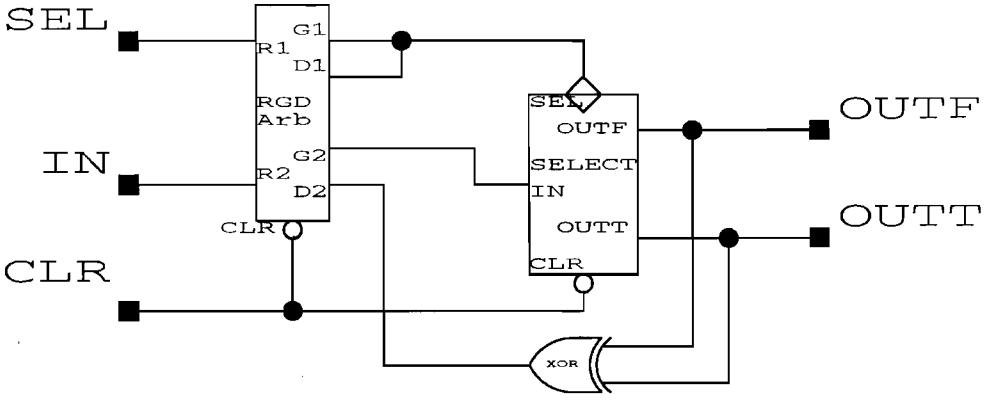

Figure 26: A Q-Select Based on an RGD Arbiter

things out of the FIFO is slower at this point than the process putting data in, so the latency is not really an issue. This should optimize the latency of the FIFO in a way which matches its use. If the FIFO is empty or nearly empty, then the consumer is keeping up with the data generation and low latency is key. If the FIFO is filling up, then the consumer is not keeping up and latency is less important than throughput.

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 24. This cell must use an arbiter of some sort to check the full/empty status of the rest of the FIFO and decide which direction to send the data. The arbiter used here is a Q-select module. A Q-select module is like a select module except that the *Sel* signal need not be bundled with respect to the transition input. A simple way of building a Q-select is to use a two-phase Q-flop to sample the *Sel* signal followed by a simple select module as shown in Figure 25. Another technique is to use a two-phase RGD arbiter to sample the *Sel* and the transition input. Using this circuit, as shown in Figure 26, the arbiter insures that either the *Sel* is changing, or the transition input is changing, but never both at the same time.

Once you have a Q-select module, you can use it to check the status of the full/empty condition and decide which direction to send the data from the current top-row cell. There are two choices for when to do this: either check the condition first as the data are being entered into the cell, but before they are latched in the first cell (type-1), or latch the data in the first top-row cell and *then* check where to send them (type-2). The trade off is that in the type-1 case the data travel through only a single FIFO stage in the best case (i.e. the incoming data jump right to the output cell in the

bottom) thus the *RIN* to *ROUT* time is minimized and the latency is lowest. The drawback is that the throughput is lessened slightly because the *RIN* to *AIN* time when the data are acknowledged to the process that is sending data to the FIFO is lengthened because the *AIN* will not be generated until after the data are latched in the bottom row and thus more control circuitry is in that path. In the type-2 case, the data are acknowledged very quickly in the top FIFO (*AIN* comes quickly after *RIN*), but the *RIN* to *ROUT* time that determines the best-case latency is a little longer because in the case of an empty FIFO the data travel through two FIFO stages instead of one.

A circuit for a type-1 top-row cell is shown in Figure 27, and a type-2 top-row cell is shown in Figure 28. Note that in each case the top-row cell checks the full/empty status of the cell beneath it which is true if the cell beneath is full, or any cell to the right of that in the bottom row is full, and also checks the incoming full/empty status from the right in the top row. This signal is true if any cell in the top row to the right of the current cell is full. If neither of these two signals is true, then the top-row cell knows that it is safe to jump directly to the bottom row. If either of these signals is not true, then the data must move to the right in the top row.

The Q-select shown in these figures is not quite as simple as mentioned in a previous paragraph. The problem is that this cell library came from a set designed to be used with an Actel FPGA [3, 1], and there is no way to build a "real" Q-flop or arbiter using an Actel FPGA. The Q-select in this library uses a pair of transition latches. The *Sel* signal is latched in the first latch upon receipt of the incoming request signal. After some amount of time, hopefully enough to have a good chance of letting any metastability resolve, the *Sel* latched into the second latch. After this, the now stable *Sel* signal is used as the select input to the selector, and the delayed *Req* is used as the transition input to the select giving a transition on either the *Tout* or *Fout*.

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, their choices will be mutually exclusive so no arbitration is needed in the bottom row. That is, you know you will get a request for data from one of those two directions, but you should not get a request from both. 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 29. A Call element on the input C-element should allow data to be entered from either direction. Note that this requires a mux-latch 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 right channel changes the mux on a request and resets the mux when it acks.

All that is left is to connect these cells in a folded FIFO organization. The details of this interconnection are shown for a four-deep section of FIFO in Figure 30. In this figure you can see the interconnection of top-row cells as a linear FIFO running left to right, and the bottom row cells in a linear FIFO running from right to left, with skip-paths from each top-row cell to a bottom-row cell. Notice also that full/empty status is passed both in the bottom row and top row in daisy-chain fashion.

These blocks of four FIFO cells may be cascaded into a FIFO of any desired length as shown in the 16-deep FIFO in Figure 24. There is, however, a small adjustment possible at the end where the FIFO makes its U-turn. Once data has made it to the end of the top FIFO, it must jump to the bottom FIFO. There is no need to check and see which direction the data should move. In this case, a standard FIFO cell will suffice as shown in Figure 31. Notice that the last FIFO cells on the top and at the bottom are standard FIFO cells. The only modification is that these FIFO cells should be part of the chained full detector so that it is not possible for data in the upper FIFO to be fooled into thinking that it can jump to the bottom when there are actually data in the standard FIFO cells that have not yet made it around the corner. The modified FIFO cell simply adds the XOR/OR combination found in the bottom cells to a standard FIFO cell. This circuit is shown in Figure 32.

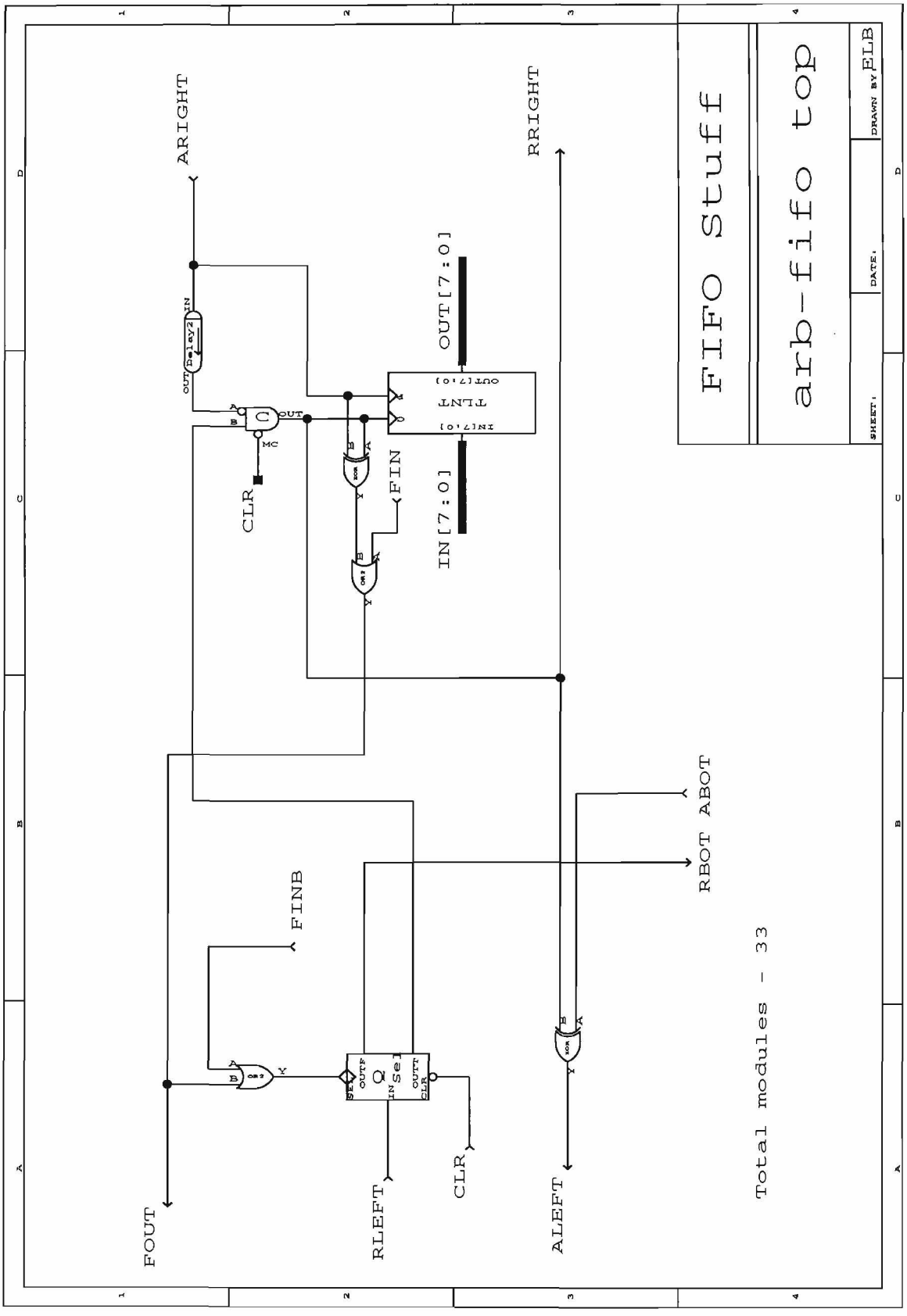

Figure 27: A Type-1 Top-Row Cell for an Arbited FIFO

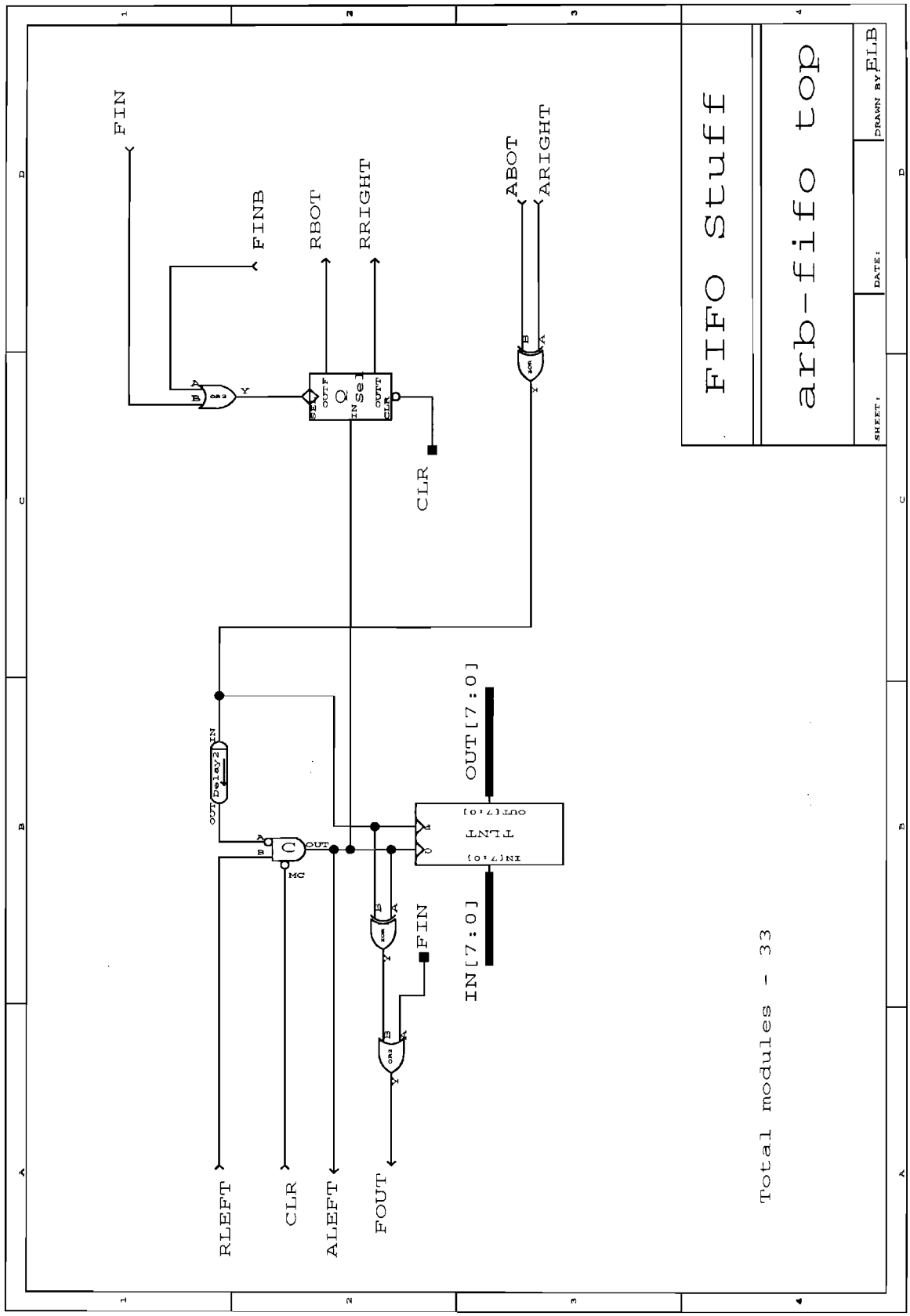

Figure 28: A Type-2 Top-Row Cell for an Arbited FIFO

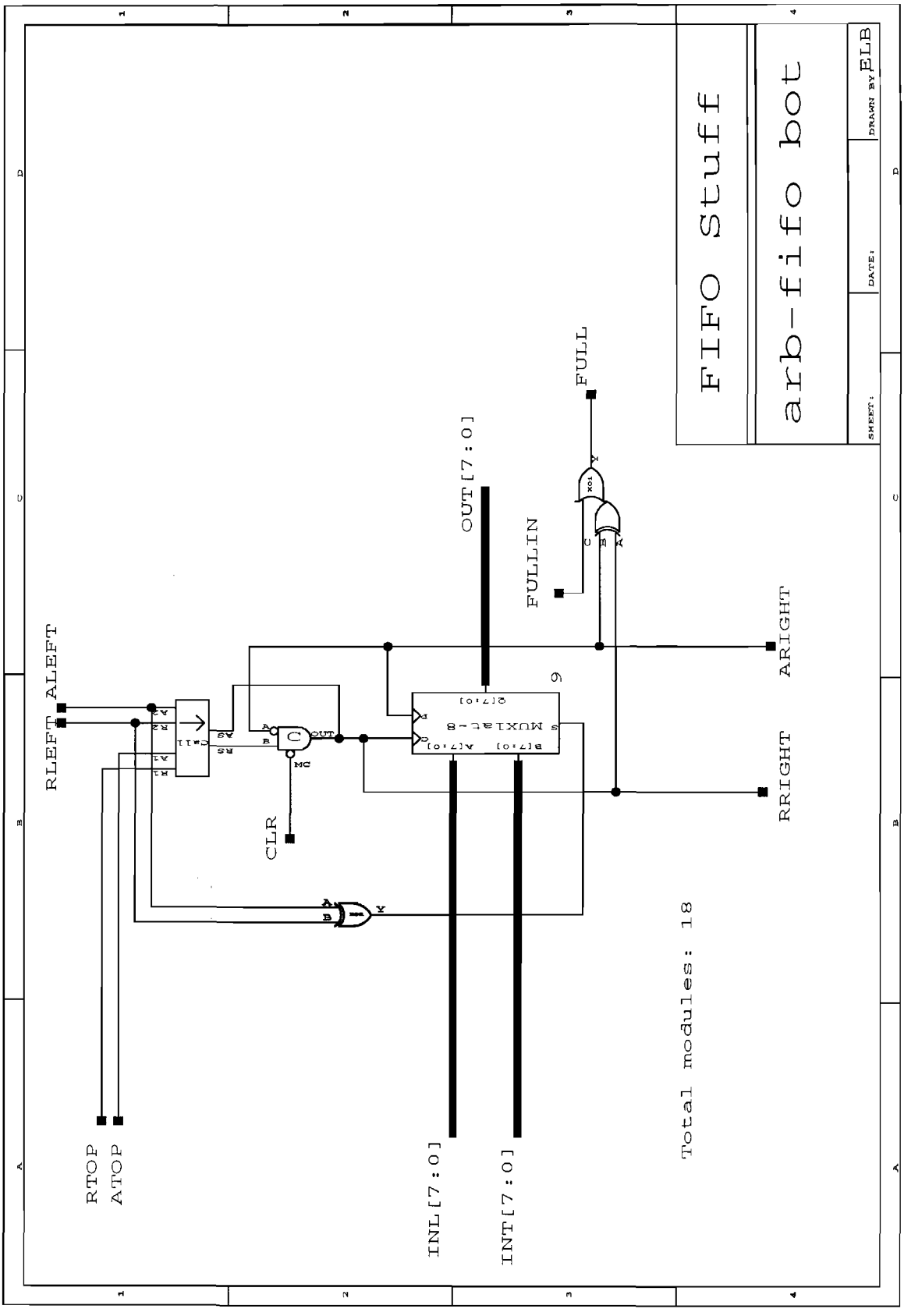

Figure 29: A Bottom-Row Cell for an Arbited FIFO

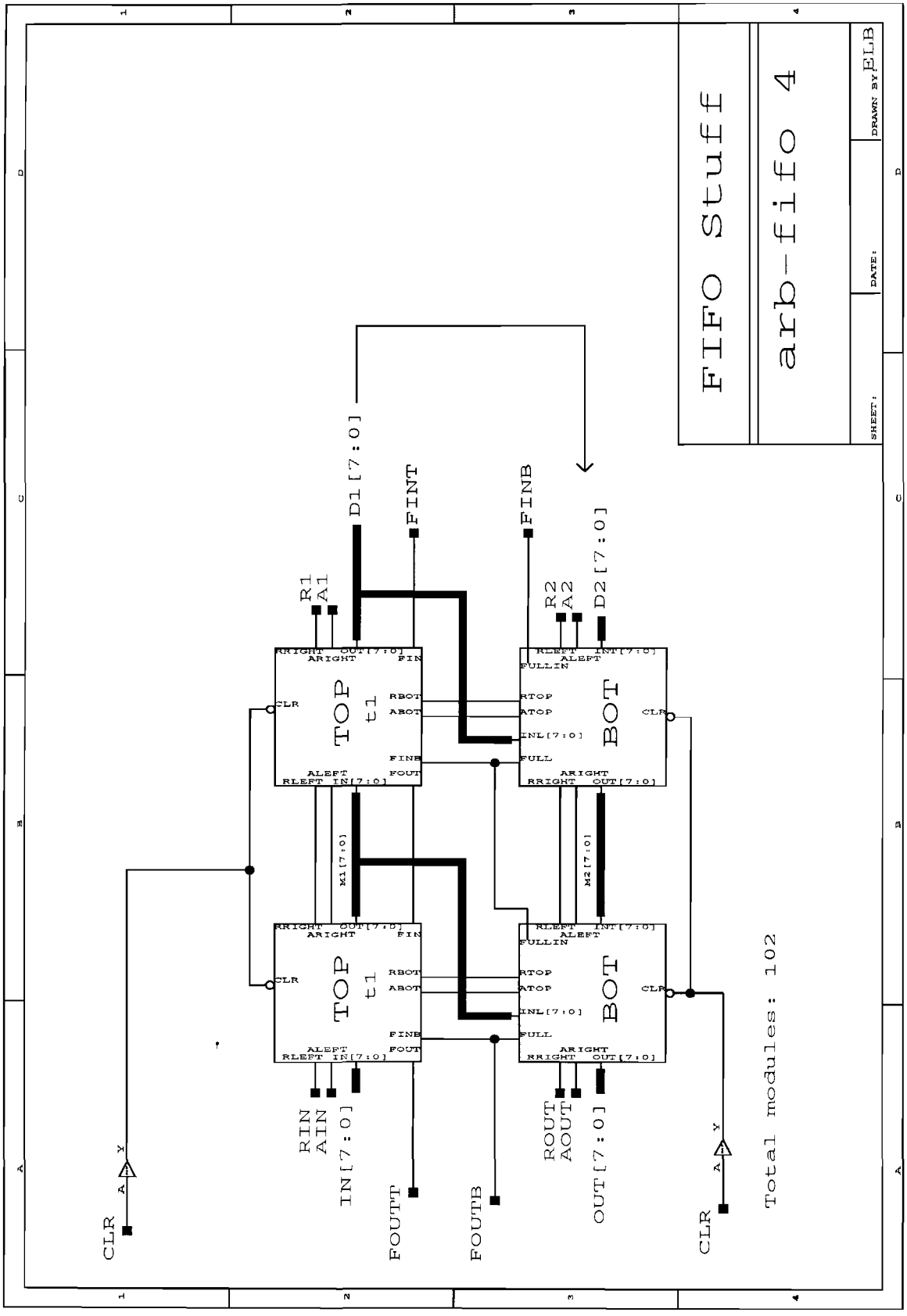

Figure 30: A Four-Deep Section of an Arbited FIFO

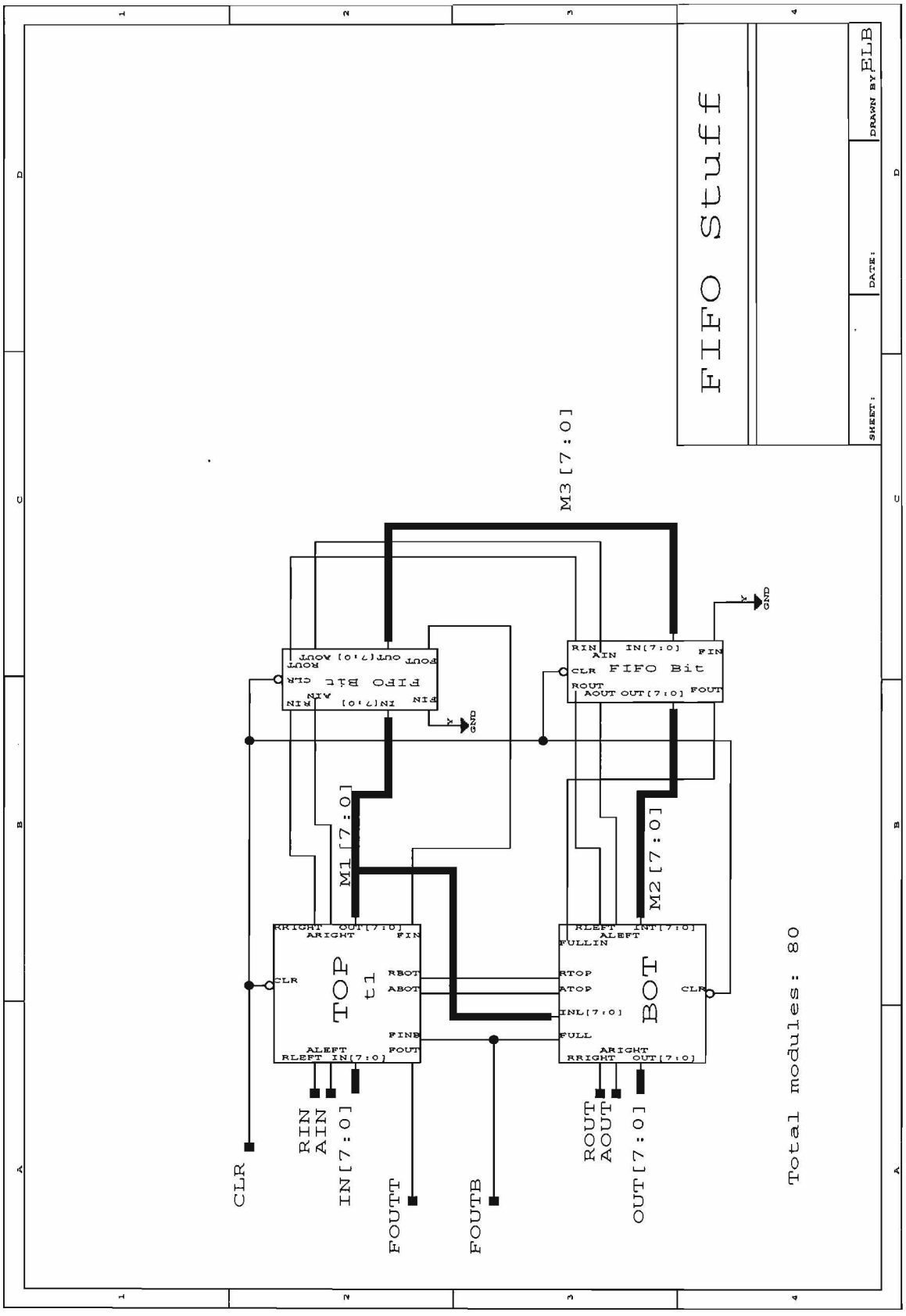

Figure 31: The Four-Deep U-Turn Section of an Arbited FIFO

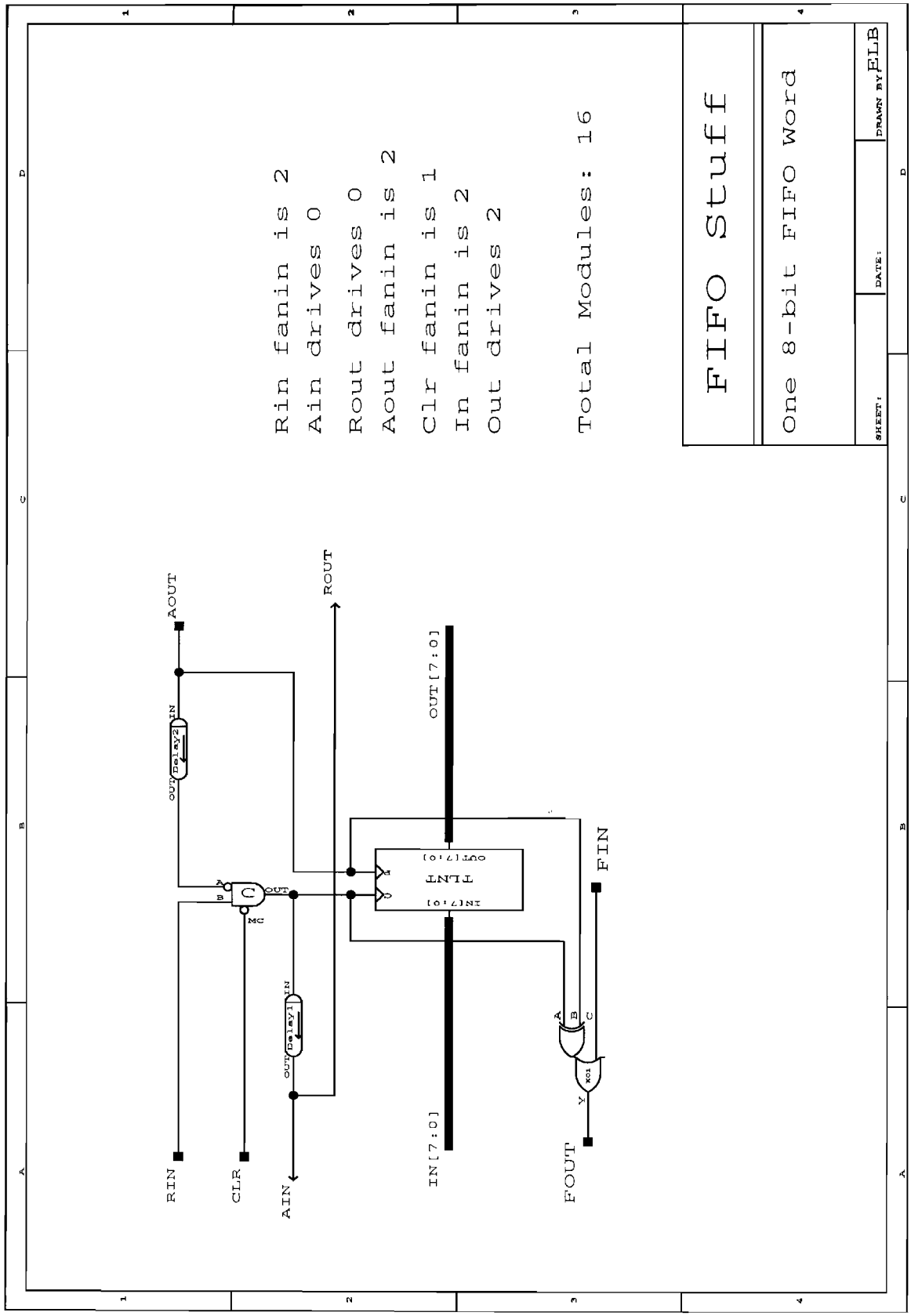

Figure 32: Standard Eight-Bit FIFO with a Full Detector

7 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 distribute 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 it reduced by $O(k)$ where k is 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 the theoretically maximum 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 network. Each tree FIFO cell is a true FIFO cell in the sense 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 they 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 theoretical latency.

The arbited FIFO takes a different approach than the others. It is structured in a folded 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, and there are no full FIFO cells between the current cell and the jump-to cell in the bottom, 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 intervening FIFO is 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 the type of top cell used.

Of course, combinations of the above FIFOs are possible, and may even be desirable. 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, and 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.

Each of these FIFOs has been implemented as a circuit using the Actel gate library and the async library of control cells. 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 are the total number of Actel basic macros used to build a 16-deep FIFO in both 8-bit and 32-bit versions. The percent overhead is a measure of

Table 1: Sizes for 16-Deep FIFO in Two Different Bit Widths

FIFO Type	8-bit		32-bit			
	# of Macros	% Size Increase	# of Macros	% Size Increase	Latency	% Latency
Linear	258	0%	786	0%	48	100%
Parallel	292	13%	820	4%	20	42%
Tree	320	24%	848	8%	27	56%
Square	311	20%	839	7%	32-27	67%
Arbited, type-1	390	51%	918	17%	10	21%
Arbited, type-2	390	51%	918	17%	13	27%

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. It is also the case that a wider FIFO word will should lower overhead since the extra circuits are all in the control circuitry. A CMOS implementation will likely look quite different, and even the Actel circuits use extra macros to take care of unknown bundling delays and other artifacts of their technology. The arbited FIFO is probably the hardest of all to compare since the Actel version of the Q-selector 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 document and with a simplistic count of number of gate delays inside the various control modules. The Percent Latency column give the latency of that FIFO as related to the latency of a straight flow-through FIFO. Actual timing using an FPGA will not have much to do with the timing of a CMOS implementation. If these FIFOs are of interest to anyone, perhaps a CMOS implementation, even if just to run Spice, would provide interesting information about realizable latency gains as well as power reduction possible with these FIFOs.

8 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 arbited FIFO was based on a comment by Al Davis. What I have done in this paper is simply tried to give each of these FIFO organizations a realistic circuit rather than just an idea.

References

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

[2] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991. Available as Technical Report CMU-CS-91-198.

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

[4] 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.

[5] Robert F. Sproull and Ivan E. Sutherland. Modules for branching FIFOs. Technical Memo SSA-4632, Sutherland, Sproull, and Associates, 1986. Chapter 4 of Asynchronous Systems, Technical Memo SSA-4706.

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