THE DESIGN OF A SELF-TIMED LOW POWER FIFO USING A WORD-SLICE STRUCTURE

A thesis submitted to the University of Manchester

for the degree of Master of Philosophy

in the Faculty of Science and Engineering

September 1998

By

KyoungKeun Yi

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

A new structure for a FIFO (First In First Out memory), the **Word-Slice** FIFO, has been developed in order to attain high performance and low power.

Conventional **asynchronous micropipeline** structures have two major problems when used as a FIFO. The first problem is high power consumption and the second is low performance. The serial structure of the data path and the control mechanism of the micropipeline cause the problems.

This dissertation presents solutions to these problems using a new FIFO structure. Each word of the memory element has its own local controller, which allows the FIFO structure to be as simple as a micropipeline FIFO. All memory elements are arranged in parallel for high performance. Only one local controller and one memory element are activated for a data transfer for low power consumption.

Post-layout simulation, in a 0.35 micron technology, shows that a Word-Slice FIFO which has 16 words of 32-bit memory consumes 183.5 pJ to transfer data when half the data bits are toggled. In a corresponding micropipeline FIFO, the energy used for the same data transfer is measured as 388.1 pJ. Data delays from input to output are 3.8 ns and 12.4 ns for the Word-Slice and micropipeline structure respectively.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to thank my wife, JeongHye, who has been with me throughout the difficult times of my study and gave birth to our second baby during that time.

Special thanks to my supervisor, Professor Steve Furber, for much precious advice for my study on the interesting world of asynchronous design. I would also like to thank to my advisor, David Lloyd, for his great help during the writing of this thesis. Thanks to William John Bainbridge for his comments on my writing.

Finally, I would like to thank Andrew Bardsley, Dr. Philip Endecott and all the smart members in the Amulet group who helped me to understand asynchronous design.

Chapter 1

Introduction

1.1 Overview

This thesis presents the design and evaluation of a new structure for a low-power high-performance asynchronous FIFO (First In First Out memory). The two main existing designs for FIFOs are analysed. The advantages and disadvantages of both are identified, leading to the new structure.

1.1.1 FIFO

A FIFO is a memory component which is widely used in digital systems. It can store consecutive data values in order, and output the stored data in the stored order. The input and the output of a FIFO can have different data transfer rates. Consequently, they are frequently used in VLSI design to maximise the utilisation of a communication channel, by smoothing the fluctuations in the data

production rate.

There are three requirements for the VLSI implementation of a FIFO. Firstly, we need simple structure: when the size of a FIFO can be scaled easily, it is easy to build a physical implementation of the circuit. Secondly, we need low power to reduce the cost of packaging: high power consumption in a VLSI device generates high temperatures which need to be cooled down by an expensive heat-emitting package. Finally, high performance is also necessary for a VLSI design.

1.1.2 An overview of the problems in current FIFO structures

There are two main structures used for implementing FIFOs: micropipeline and ring-buffer structures.

Micropipeline FIFO

Conventional *Micropipeline* [26] structures which are described in section 4.1 on page 38 have two major problems when they are used as a FIFO: high power consumption and low performance.

High power consumption is caused by the unnecessary activity of the memory elements and the control elements in a micropipeline FIFO. Input data have to travel along a serial data path, through multiple stages of memory elements. This means that all the memory elements of a FIFO are activated, which causes high power consumption. All the control signals are activated for a data transfer. This

activity adds to the power consumption.

The data delay increases proportionally to the depth of the FIFO as it is increased. This incremental delay causes significant degradation of the performance as the FIFO depth increases.

Ring-buffer FIFO

A solution to these problems of high power consumption and low performance is to adopt the conventional ring-buffer FIFO [27] which is described in section 4.2 on page 43 to reconstruct the micropipeline FIFO function in another structure. However, as Sutherland mentioned in his seminal lecture, **Micropipelines** [26], the ring-buffer FIFO is very hard to design. The centralised controller makes the VLSI design hard because the FIFO size is not scalable.

1.1.3 A new FIFO structure

A solution to these problems, a new FIFO structure, the Word-Slice FIFO, is presented in this thesis. This structure has been developed in order to interface high performance asynchronous and synchronous systems efficiently. The new structure has several benefits: low power consumption, high performance, easy modification and a simple interface.

For the new ***Word-Slice*** FIFO, each word of the memory element must have its own local controller to build a simple structure, just as each stage of

the memory element has in a micropipeline FIFO. However, only one local controller and one memory element are activated for a data transfer, unlike the usual micropipeline FIFO controller.

This design, therefore, requires all memory elements to be arranged in parallel for high performance and for low power consumption.

1.2 Contribution

The new FIFO structure described in this thesis has three characteristics which have not been fully satisfied with current FIFO structures.

Firstly, the new structure is simple. It is easy to modify the FIFO size in physical layout in a VLSI design. We can build a FIFO from the prepared library of a FIFO element by simply connecting the elements in series. Consequently, it is possible to build a data-path library for the FIFO design, thus designers do not need to worry about designing various sizes of FIFOs by themselves.

Secondly, the new FIFO consumes less power. The high cost for a heat-emitting package for a high performance VLSI device can be avoided. The battery life time can be extended when the structure is used in the design of portable devices.

Finally, the new FIFO has low latency, thus we can make a high performance system.

1.3 Thesis structure

In chapter 1, the purpose of using a FIFO in a high speed system interface was mentioned: a digital system uses FIFOs to maximise the utility of the bandwidth in a communication channel. An overview of the problems in current FIFO structures was discussed followed by an overview of the suggested solution.

In chapter 2, basic concepts of asynchronous design will be described. The characteristics of asynchronous logic will be considered.

In chapter 3, the operations of the basic elements used to build a micropipeline will be described.

In chapter 4, a new FIFO structure will be considered. First, detailed problems with existing FIFO structures will be mentioned. Next, the detailed solutions to the problems will be discussed. Finally, we will implement the new idea as a physical design.

In chapter 5, the design methods used in this work will be introduced. Before the evaluation of the design, two major FIFO structures will be implemented: *micropipeline* and $word - slice$ structures. An analysis of the energy efficiency will be presented for each of the two different FIFO structures.

In chapter 6, we will compare the performance of different FIFO structures.

In chapter 7, we will summarise the analysed data and the results. This is followed by the conclusions from this work.

Chapter 2

Concepts and terms in asynchronous logic

2.1 Synchronous logic

Digital logic design can be divided into two categories: synchronous design and asynchronous design. Synchronous design uses one or more global synchronising signals, called clock signals. Clock signals are connected to the memory elements, flipflops and latches. Figure 2.1 shows synchronous logic which has a global clock signal, *CK*. The state decoder derives the next state, *NS*, from the present state, *PS*, and/or from the input signals, *In*. The output decoder derives the output signals, *Out*, from the present state, *PS*, and/or from the input signals, *In*.

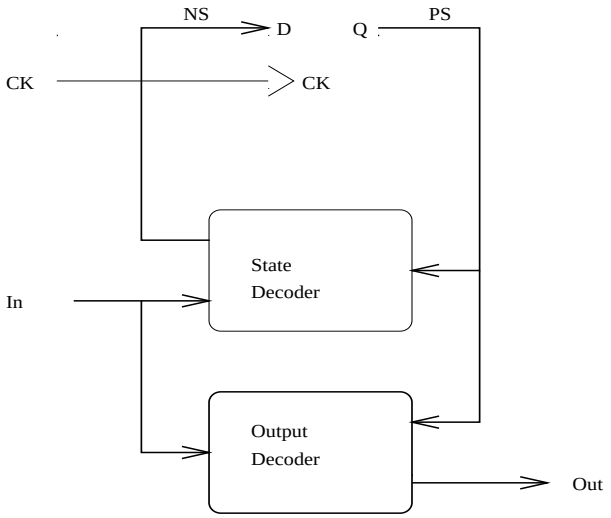


Figure 2.1: Synchronous logic

2.2 Asynchronous logic

Asynchronous designs do not use clocks, or any global synchronising signals, unlike synchronous designs. We can see an example of asynchronous logic in Figure 2.2.

The processing module is for data processing. The *Start* signal denotes the request to start processing. The *End* signal notifies the end of current processing in the module. The transition sequence controller defines the order of all of the internal signals' transitions.

The delay is one of the most important factors for digital logic designs regardless of whether a logic design is asynchronous or synchronous. Asynchronous design methodologies/concepts are based upon assumptions on the characteristics of the delay. Delay characteristics are grouped into one of the two models outlined below.

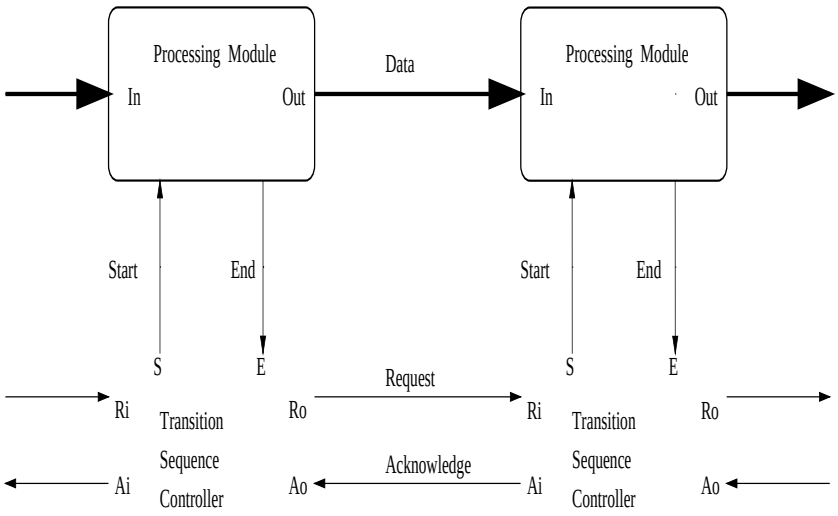


Figure 2.2: An example of asynchronous logic

- Bounded delay model [21]

  A delay model which has lower and/or upper bounds on the delay of a circuit element or wire is called a *bounded* delay model.

- Unbounded delay model [21]

  A delay model which has no upper or lower bound on the delay of a circuit element or wire is called an *unbounded* delay model.

These delay models are used to define the following four models of circuit operation.

- Huffman model (Fundamental-mode) [17, 10, 25]

  One class of asynchronous models which assume bounded delays on their wires and circuit elements are called fundamental-mode FSM (Finite State Machine) or the Huffman models. The inputs of the circuit must not be changed until all of the internal states are stable. If an input is changed

while the internal state of the circuit is transient, the final state of the logic is unknown. As a result, the interval between transitions on each input should be greater than the maximum delay of the feed-back loops or shorter than the minimum delay of the logic to avoid disturbances to the sequence of the internal states. Thus the delay must be bounded.

- Speed-independent model [11, 17]

  The Huffman model uses only inputs and outputs of a circuit to represent the state. However, in the speed independent model, all of the internal nodes' states are considered. The speed independent model represents the circuit operations by using precedence relations between all of the internal signals. Thus we can define the operation of a signal by a subset of the total state of transitions of all signals that effect the transition of the signal ([7] pp. 106-107). The delays caused by the interconnecting wires are assumed to be zero in this model.

- Delay-insensitive model

  While the speed independent model assumes zero delay on all the wires in the circuit, the delay insensitive model assumes unbounded but finite delays on both the circuit elements and wires in the circuit.

- Quasi delay-insensitive model

  The difference in the delays at each end of a wire fork causes a conflict in CMOS operation. To avoid this conflict, the quasi delay insensitive model

has been proposed. This model assumes that the wires in the circuit only have isochronic forks.

An isochronic fork is a set of interconnecting wires where the delay difference between the branches is zero or negligible compared to the circuit element delays.

The speed-independent model and the bounded delay model are used for the controllers in the new FIFO design, while the bounded delay model is used to build the data path.

The micropipeline described in the following section uses the bounded delay model for the data path, while it uses the speed-independent model for its controller.

2.3 Micropipeline

There are two methods used in asynchronous logic to indicate when the outputs from a combinatorial logic block are valid. The first encodes validity information with data using a scheme such as dual-rail encoding to produce a completion signal. The second uses delay matching [4] logic where, in a micropipeline, the data is assumed to be valid within a bounded delay so that a constant delay element can be used to indicate the data validity.

The main approach used for the circuit is based on the **Micropipelines** Turing Award lecture of 1988 [26]. The proposed design methodology is composed

of a bounded-delay data path controlled by delay-insensitive circuits [25].

2.3.1 Synchronous vs. asynchronous pipelines

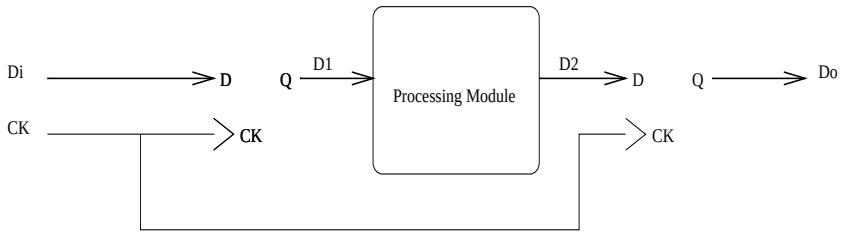


Figure 2.3: Synchronous pipeline

Figure 2.3 shows the typical structure for a synchronous pipeline. The period of the clock, *CK*, should be greater than the sum of the delay from the *CK* to *D1*, from *D1* to *D2* and the setup time of the flip-flop. The performance of a pipeline with a processing module is a constant determined by the clock. The maximum performance of a synchronous circuit is normally dependent not only on the slowest part of the data processing modules but also on the slowest part of the control modules.

A micropipeline is shown in Figure 2.4. In contrast to the synchronous pipeline, a stage of a micropipeline consists of a processing module, a control module and memory elements such as latches. The clock distribution network is replaced by the control module. When the data *D1* is valid, an input request is generated on the *Ri* signal by the control module at the left side. An input handshake cycle is initiated by this request. When the control module at the right side receives a signal transition on the *REQi* signal, it sends a start signal,

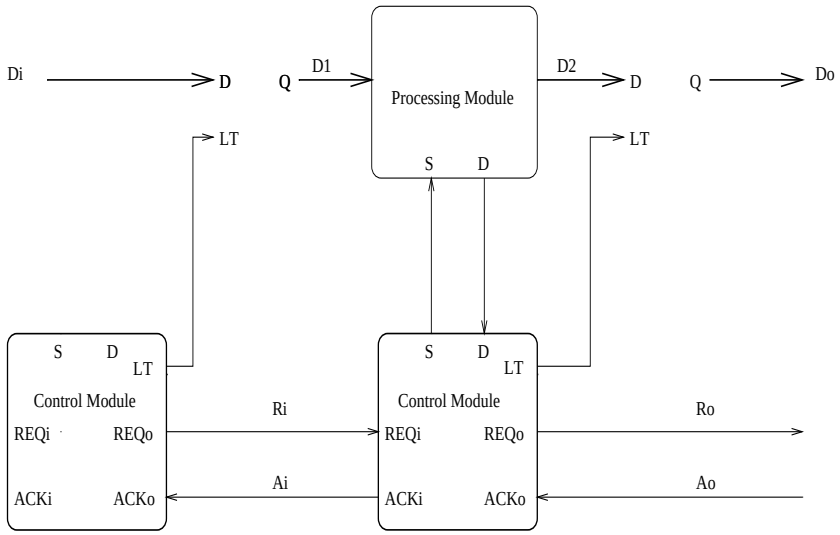


Figure 2.4: Asynchronous pipeline

S, to the processing module. After the completion of data processing in the processing module, it returns a done signal, *D*, to the control module. After that, an input acknowledge signal, *Ai*, is changed to indicate the completion of data reception. At the same time, the data *D2* is latched in the right latch. After the latch operation, an output request signal is sent to the next stage through the *Ro* signal to initiate an output handshake cycle.

2.4 Handshake protocols

Handshaking protocols are used to maintain the correct ordering of data and operations. When data are transfered, they are bundled with a control signal which represents the timing information to indicate data validity.

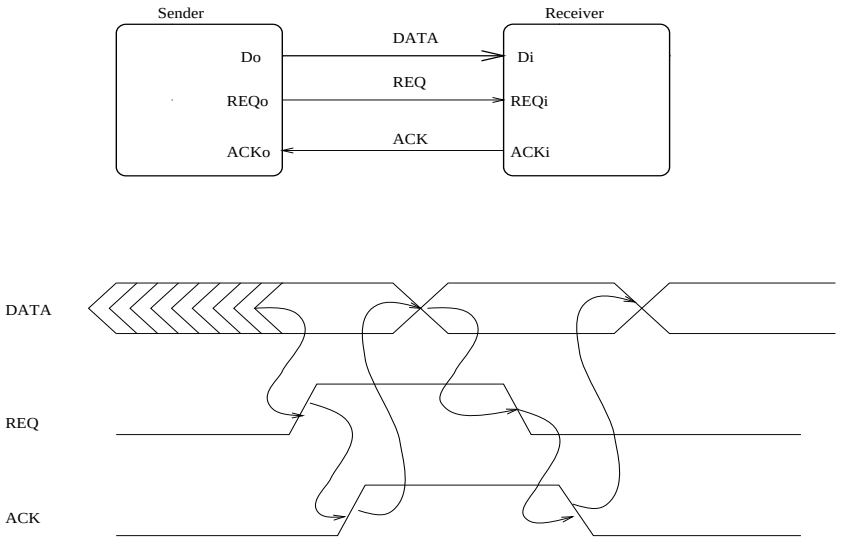


Figure 2.5: The two-phase Handshake protocol

2.4.1 Two-phase handshake protocol

Figure 2.5 shows an asynchronous handshake structure and a timing diagram for a two-phase handshake protocol.

The data are accompanied by a pair of handshake control signals. When the data are valid, the sender makes an event on the REQ signal. The receiver returns an event (a signal transition) to the sender using the ACK signal to indicate that the data have been received. After the event on the ACK signal, the sender starts to produce the next data on the $DATA$ signals.

The constraints at the receiver inputs dictate that the changed control signal, REQ, should arrive after all of the changes in the data are transmitted to the receiver: the data at the output of the sender can be changed only after the change of the acknowledge control signal, ACK, from the receiver.

2.4.2 Four-phase handshake protocol

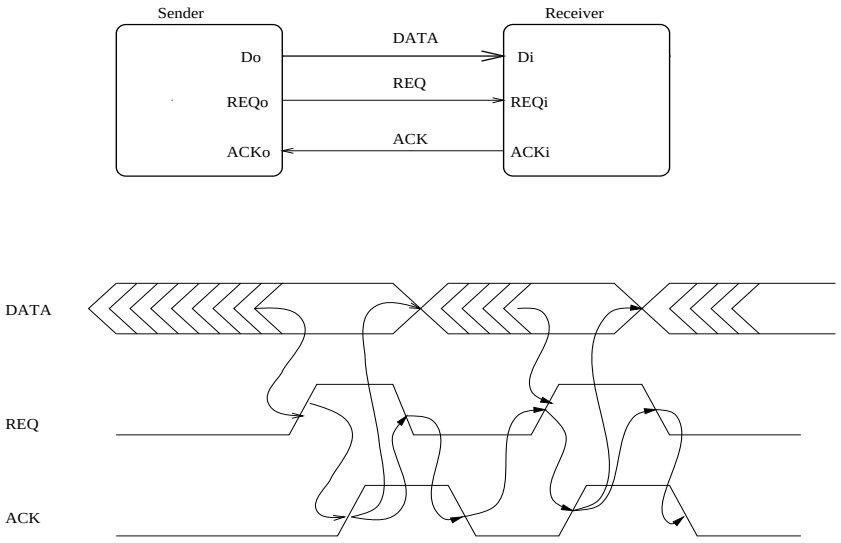


Figure 2.6: The four-phase handshake protocol

A four-phase handshake protocol is illustrated in figure 2.6. (The other four-phase handshake protocols are described in [5, 18].) Two transitions on each handshake control are needed to complete a data transfer. A low-to-high transition on the request signal, *REQ*, initiates a handshake operation. The rising edge of the *REQ* signal indicates that the data are available. The rising edge of the *ACK* signal indicates that the data have been latched into the receiver. The falling edges of the handshake control signals are return to zero actions before the start of the next handshake operation.

2.5 The characteristics of asynchronous design

2.5.1 Local handshake without global clock signal

The design of the physical layout of a synchronous circuit is becoming more demanding because of the global constraints on the clock-distribution net. To avoid degradation of the performance as a result of global clock skew, the designer must take great care with the clock signals in VLSI design.

The power consumed by a capacitive load is

$$P = CV^2F \tag{2.1}$$

where C is the load capacitance of the CMOS logic, V is the operating voltage and F is the operating frequency. According to the above equation, a high operating clock frequency requires more power to drive the capacitive load of the clock network. So the power consumption supporting the clock distribution network increases as the performance of synchronous systems increases.

In contrast, self-timed design uses local handshake control mechanisms which are required to satisfy the local constraints described in section 2.4. Only local constraints have to be satisfied by designers when developing physical layout.

2.5.2 Average performance

It is possible to end a handshake operation early when the operation in a processing element in a micropipeline stage finishes early. However, synchronous pipeline operation is controlled by a clock which permits only a fixed data transfer rate in the pipeline. Consequently, the performance in a pipeline can be greatly affected by whether the pipeline control mechanism is synchronous or asynchronous.

The fixed performance demanded by the maximum processing delay in synchronous design can be compared to the variable performance in a stage of an asynchronous pipeline. In a study by Mark [20], the performance of an asynchronous pipeline demonstrated the conditions necessary to obtain average case performance. The longest delay in multiple stages of an asynchronous micropipeline is the most important factor which defines the spontaneous average performance of the pipeline. Consequently, the average performance in multiple micropipeline stages does not always lead to the high performance which can be achieved by a single micropipeline stage.

2.5.3 Electro-magnetic emissions and noise.

A synchronous system usually needs delicate shields to block electro-magnetic emissions. The periodical clock causes simultaneous switching of the devices in a digital hardware system. Simultaneous switching leads to a concentration of the emission energy around periodical frequency/time regions. This saturated energy causes disturbances to other devices, e.g. noise to an audio system, undesirable

effects to medical equipment and noise to mobile communications.

However, an asynchronous system spreads the energy relatively evenly over time and frequency because the operations of the devices are not synchronised to a periodical clock. As a result, the electro-magnetic energy radiated from an asynchronous system can be spread relatively evenly over the time/frequency domain.

Chapter 3

Basic elements for building asynchronous circuits

Ivan Sutherland, in his Turing award lecture of 1988 [26], introduced the concept of **Micropipelines**. He showed how basic circuit elements can be combined to provide the control for complex micropipeline systems.

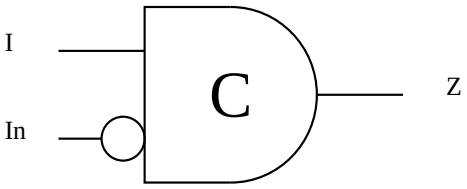
To describe the operation of a micropipeline, the Muller C-element and an event-controlled storage element must be introduced. A simple asynchronous pipeline can be built with these two basic elements.

3.1 C-element

An **event** is a signal transition on a control signal in an asynchronous circuit. The Muller C-element [26, 11] acts as an AND function for events.

A simple two-input C-element is illustrated in figure 3.1. A bubble on the

< C-element >



< State Transitions >

I	In	Z
0	0	Z
0	1	0
1	0	1
1	1	Z

Figure 3.1: C-element

left, input side means an inversion. Thus the output copies the state of the upper input if the inputs differ in state (signal level): otherwise the output holds the previous state [26].

3.2 Event-controlled storage element

Normal memory elements such as flipflops transfer the data using only one of the high-to-low or low-to-high transitions of a control signal. Event-controlled logic may need to transfer data on both of the transitions on the control signal. An event controlled storage element [26] is illustrated in figure 3.2.

There are two signals to control the data: capture, C, and pass, P. Each of these signals is connected to two nodes, one input and one output, to make handshakes between the adjacent stages in a pipeline. For example, the capture

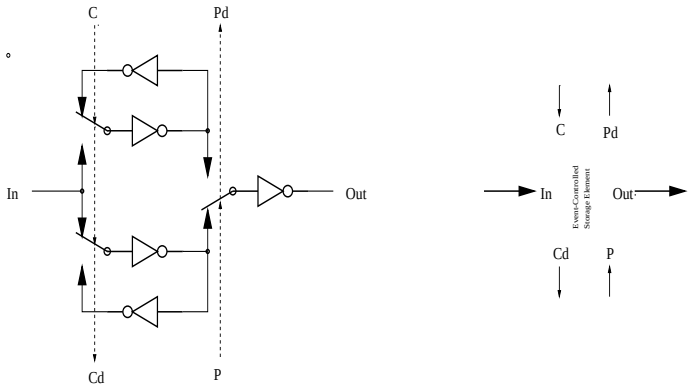


Figure 3.2: Event-controlled storage element

signal is connected to both the capture requesting signal, *C*, and the capture-acknowledging signal, *Cd*. The data transfer procedure through the event-controlled storage element is as follows.

- Initial state:

  The circuit is in transparent mode as shown in the figure 3.2.

- Capture:

  The sender at the input side places data on the input, *Din*. The input signal, *C*, receives an event from the sender which requests the latch to capture the data input. The input switches are flipped to the lower position to form a state-retaining loop.

- Capture done:

  The input, *In*, is captured into the lower inverter loop when an event is introduced at the capture signal, *C*. An acknowledge event, *Cd*, is sent back to the sender after the capture has completed.

- Pass:

When the data at the output is captured by the next stage, an event is sent to the P input. The output switch is flipped to the upper position.

- Pass done:

  The storage element is in transparent mode again. An event is returned to the control unit through the signal Pd to acknowledge that the storage element is in transparent mode and ready to capture the next input data.

- Complementary state:

  The storage element is now in the complementary state to the initial state.

In the final complementary state above, the storage element is actually in the same state as the initial state taking into account that the high-to-low and low-to-high transitions have the same meaning in event-controlled logic.

3.3 Structure of the micropipeline

There are area overheads for circuits used to indicate data validity, such as dual rail encoding. A micropipeline uses the bounded-delay model for the processing module to eliminate the overhead of the redundant circuitry for indicating data validity. In the case of the control module, which is normally much smaller than the processing module, a micropipeline uses the speed-independent model.

Figure 3.3 shows an example of a micropipeline without processing elements. The thick signal lines are data paths, while dashed lines are the control signal feedbacks for the handshaking acknowledges.

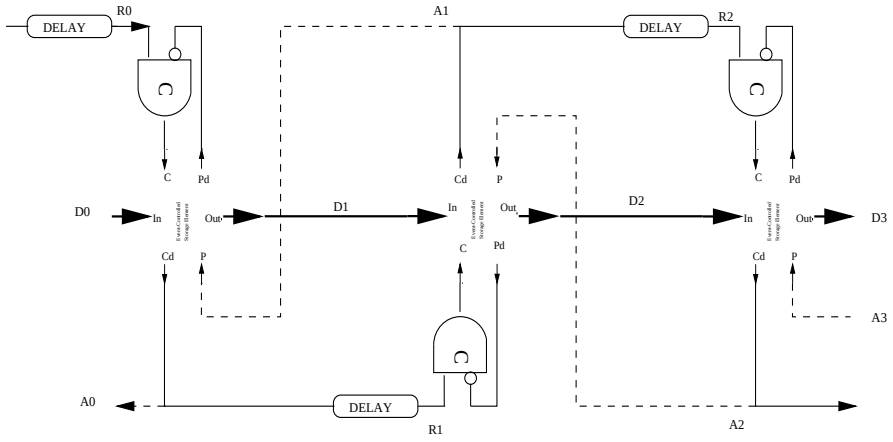


Figure 3.3: The structure of the micropipeline

The thin solid lines are carrying the control signals forward for the handshaking requests.

The handshaking loops (request/acknowledge pair) make the propagation of the control and data as follows.

- Initial state:

  All of the control signals are assumed to be low in the initial state in normal operation. So, all of the storage elements are in transparent mode.

- Data input Captured:

  When a request input, *R0*, receives an event which means that new data are introduced at *D0*, then the output of the left-most C-element is set to high. Thereby, the data, *D0*, is captured into the left-side storage element.

- Acknowledge 0:

  At the same time, the event on the C input is transfered to the Cd output making an acknowledge event to *A0*.

- Data output passed:

  After a wire delay, the *Cd* output is transmitted to the signal *R1*.

- Acknowledge 1:

  Using the same procedure as in the first stage of pipeline, the second C-element produces an event on its output and returns it to the first storage element through the signal *A1*.

- Ready to capture the next data:

  At this time, the first C-element is ready to accept the next request event through *R0*. If the next data has already arrived on *D0* and the request on *R0*, the next capture operation should wait for an event on *A1*.

Chapter 4

New FIFO structure :

Word-Slice FIFO

In this section, a new structure and concept in designing a FIFO (First In First Out) memory element will be introduced. The new structure, the *Word-Slice FIFO*, can make FIFO design easier while maintaining low power consumption with high performance.

Firstly, the structures and operation of asynchronous FIFOs will be examined. Detailed problems will be discussed next. Finally, the new FIFO structure will be introduced.

4.1 Micropipeline FIFO

Figure 4.1 shows the structure of a micropipeline without processing elements (a micropipeline FIFO). The memory element and control element within the

dashed circle make a micropipeline FIFO element. The depth of the FIFO shown in the figure is two, because it has two memory elements. We can increase the FIFO depth by simply adding FIFO elements in series. Each memory element is accompanied by a control element. The same memory and control element are used for each FIFO element.

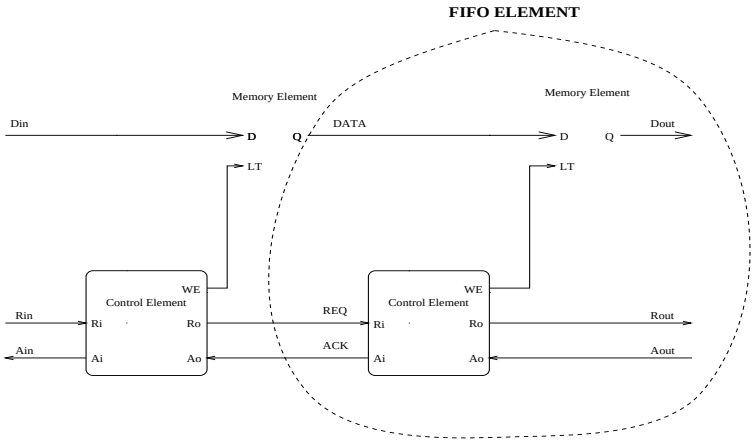


Figure 4.1: A micropipeline FIFO

To transfer data from the input *Din* to the output *Dout*, the input data must pass through the left memory element: the input data are transfered from the *Din* bus to the *DATA* bus by setting the *LT* (latch enable) signal to high. The data on the *DATA* bus is transfered to the *Dout* bus through the right memory element in the same way. With this structure all data in every memory element in the FIFO must be toggled for a data transfer from the *Din* bus to the *Dout* bus, when the input data are toggled. The total delay for data to pass from the input to the output is the sum of the delays in every memory element. Consequently, the total delay will increase proportionally to the depth of the FIFO.

The control elements are used for the handshake operation as described in

section 2.4. To transfer control from the input *Rin* (input request signal) to the output *Rout* (output request signal), the input request signal *Rin* passes through the left control element at first: an event (transition) at *Rin* signal causes a transition at *REQ*. The event on the signal *REQ* is transfered to the signal *Rout* through the right control element in the same way.

To transfer control from the input *Aout* to the output *Ain*, the output acknowledge signal *Aout* passes through the right control element at first: an event (transition) at *Aout* signal causes a transition at *ACK*. The event on the signal *ACK* is transfered to the signal *Ain* through the left control element in the same way.

With this structure, all request and acknowledge signals (*Ri*, *Ro*, *Ai* and *Ao*) in every control element in the FIFO must be toggled to control the data transfer from *Din* to *Dout*. The total data delay from a control input to a control output is the sum of the delays in every control element. Consequently, the total delay of the control signals will increase proportionally to the depth of the FIFO.

4.1.1 Normally open latch control

Figure 4.2 shows the STG (Signal Transition Graph) notation of a normally open latch control circuit (BRF latch control [18]) used for a micropipeline. *A+* denotes a low-to-high transition of the signal *A*. *A-* denotes a high-to-low transition of the signal *A*. An arrowed arc represents the order of the signal transition: the transition *A+* precedes the transition *Ai+*.

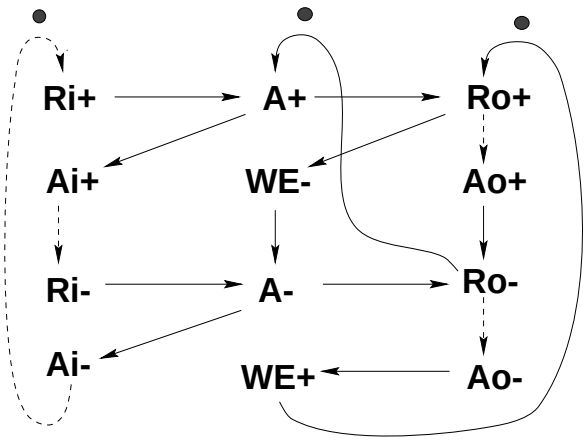


Figure 4.2: STG of a micropipeline FIFO control element

This control circuit uses a 4-phase handshake protocol. *Ri* and *Ai* are the input request and acknowledge control signals. *Ro* and *Ao* are the output request and acknowledge control signals. *WE* is a latch control signal. Initially, the *WE* signal is high to make the latch open. Therefore in a micropipeline FIFO which uses the latch control circuit as a control element, all the memory elements (latches) in the FIFO are open initially.

In an empty micropipeline FIFO, the first input data will pass through the FIFO to occupy the last memory slot. Thus the data lines of all the memory elements along the data path must be toggled when the input data are toggled. These unnecessary transitions add an extra power consumption factor.

We can avoid the extra power consumption, caused by the spurious transitions, by using a normally closed latch control.

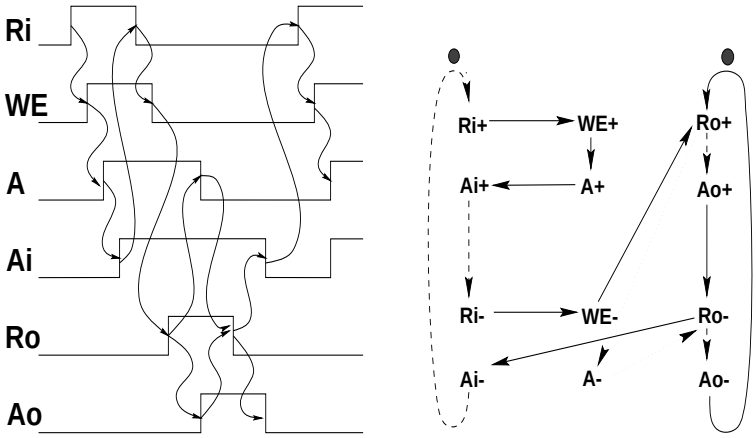


Figure 4.3: STG of normally closed latch control

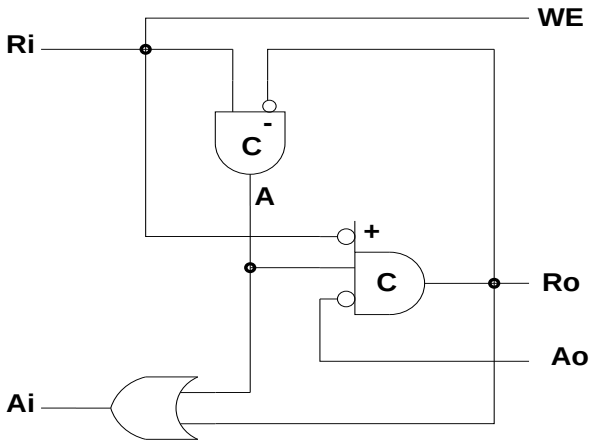


Figure 4.4: Normally closed latch control circuit

4.1.2 Normally closed latch control

Figure 4.3 shows the STG of a normally closed latch control element. Figure 4.4 shows the circuit translated from the boolean equations, generated by the Petrify tool [16]. As shown in the figures, the latch control signal, *WE*, is initially low so

that the latches in the memory elements are closed when they are empty.

However, the normally open latch control has a benefit in performance. Because the latches in an empty FIFO are open initially, it is not necessary to open the latch along the data path to transfer data through the empty memory elements.

If the *WE* signal is low initially (a normally closed latch control), the low-to-high transition of the input request signal *Ri* is followed by a low-to-high transition of the *WE* signal to open the latches. This transition causes a delay due to the capacitive load at the *WE* signal. Consequently, by removing the initial transition on the latch control signal using the normally open latch control, we can achieve higher performance.

4.2 Ring Buffer FIFO

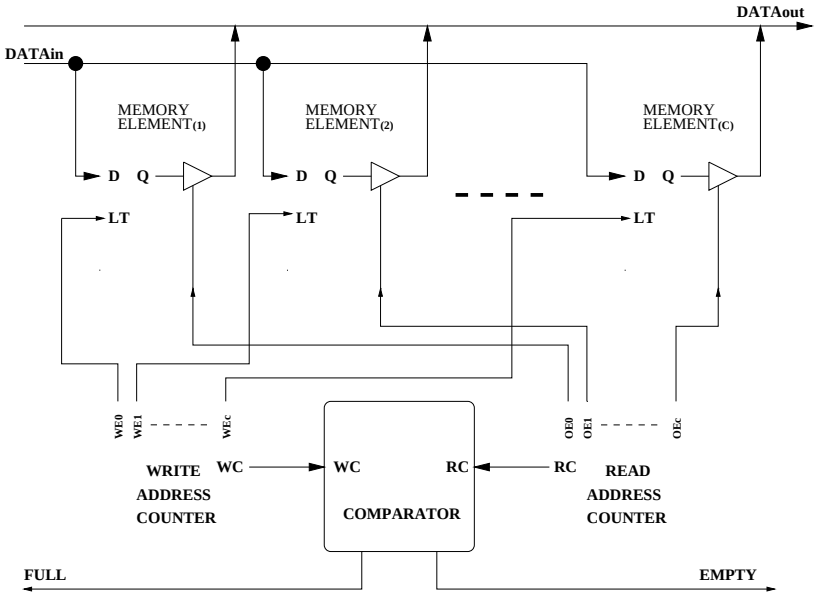


Figure 4.5: Ring buffer FIFO structure

Figure 4.5 shows the structure of a conventional ring buffer FIFO. Each memory element in the FIFO consists of a latch and a tri-state output buffer. The write address counter selects one memory element at a time, to write input data to. The read address counter selects one memory element at a time, from which a word of data can be output. The comparator compares the counter values, WC (Write Counter) and RC (Read Counter) to generate the FIFO $FULL$ or the FIFO $EMPTY$ signals.

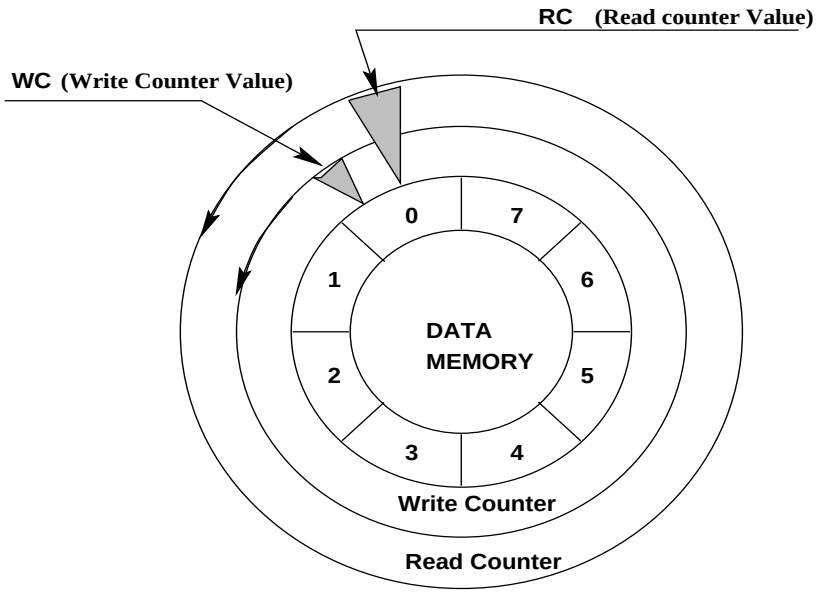


Figure 4.6: Write and read counter in a ring buffer

Figure 4.6 illustrates the operation of a conventional ring buffer FIFO. The write counter value, WC, and the read counter value, RC, are set to zero initially: the FIFO is empty. Thus the $EMPTY$ signal is high whereas the $FULL$ signal is low because the FIFO is not full.

After a write operation, the write counter value is incremented by one. At this time, the comparator compares WC and RC. If those two values are different,

the comparator sets the *EMPTY* signal to low: the FIFO is not empty.

As data are repeatedly written to the FIFO, the *WC* increases until it has the same value as *RC*. If those two values are the same after the final write operation, the FIFO is full. At this moment, the comparator sets the *FULL* signal to high. After the *FULL* signal becomes high, the next write operation is prohibited until the *FULL* signal is returned to low by a read operation.

The read address cannot increase its counter value until the *EMPTY* signal becomes low, by a write operation to the FIFO. As data are repeatedly read from the FIFO, *RC* increases until it has the same value as *WC*. If those two values are the same after the final read operation, the FIFO is empty. At this moment, the comparator sets the *EMPTY* signal to high. After the *EMPTY* signal becomes high, the next read operation is prohibited until the *EMPTY* signal is returned to low by a write operation.

We can notice that the *FULL* and the *EMPTY* signals have the same condition (*WC* and *RC* are the same) to be set to high. To distinguish the two states of the FIFO, with the same information from the two counters, we need extra information: **almost full** and **almost empty**. This information can be generated by subtracting *RC* from *WC*, or by subtracting *WC* from *RC*.

As a result, to design a conventional FIFO, we need four pieces of information: empty, almost-empty, almost-full and full. An empty state can arise if and only if the previous state is almost-empty. A full state can arise if and only if the previous state is almost-full. By keeping the previous state in the comparator

shown in the figure, it is possible for the comparator to tell the *FULL* from the *EMPTY* state.

4.3 Current problems

In the following subsections, we will discuss several problems with the micropipeline and ring buffer FIFO structures.

4.3.1 Micropipeline FIFO

Micropipeline [26] structures have two major problems when used as a FIFO. The first problem is high power consumption and the second is low performance due to the structure.

Power consumption

There are three reasons for the high power consumption in a micropipeline FIFO.

- Normally open latch control: memory element

  Firstly, spurious transitions can travel through the serial data path along multiple stages of the memory elements or latches, making a transition on every memory element. If the FIFO controllers make the latches normally open and all the latches open when the FIFO is empty, a transition at the data input causes transitions on all of the memory elements along the data path in the FIFO.

- Normally closed latch control: memory element

  These spurious transitions can be removed by using a **normally-closed** latch control method, as described in section 4.1.1. However, even though this method is used to build a micropipeline FIFO structure, there is still a problem. Every memory element must be toggled to transfer data from the input to the output of the FIFO because this FIFO uses a shift operation to transfer data.

- Common problem: control element

  The final cause of high power consumption is that all control elements are required to make a handshake operation just for a single data transfer from the input to the output of the FIFO. In other words, all of the request signals, REQs, must be toggled from the input to the output to transfer a word of data. All of the acknowledge signals, ACKs, must also be toggled just for a single data transfer. Consequently, the intensive activity of the control elements makes for additional high power consumption.

Performance

After we discussed the operation of the micropipeline FIFO in section 4.1, we can see that there will be two problems which cause performance degradation.

- The first problem with performance in a micropipeline FIFO is the delay from the data input to the output. The total data delay in the FIFO increases as the number of storage elements along the data path increases.

This incremental delay can cause significant performance degradation when the data delay is important to a system.

- The second problem is caused by the handshake mechanism which is needed to transfer the control signal forward and backward. The total delay from *Rin* to *Rout* in figure 4.1 increases as the number of control elements in the FIFO increases. Even though this delay can be reduced using normally open latch control, it is still proportional to the depth of the FIFO.

In the next subsection, a conventional ring buffer FIFO will be compared to the micropipeline FIFO.

4.3.2 Conventional ring buffer FIFO

Power consumption

Unlike a micropipeline FIFO, a conventional ring-buffer FIFO has only one memory element between the data input and the output in the FIFO, as in figure 4.5.

Because the capacitive loads of the *DATAin* and the *DATAout* signals increase as the FIFO depth increases, the total energy consumption for a data transfer increases as the depth increases. However, this incremental energy consumption can be smaller than that of a micropipeline FIFO.

Performance

The incremental capacitive load on each input and output of a memory element in the FIFO also increases the total delay of the data as the FIFO depth increases. However, as we will see in the experimental results in section 6, this incremental delay can be smaller than that of a micropipeline FIFO.

Design

A conventional ring-buffer structure can be used to implement a FIFO [27] both for lower power consumption and for higher performance. However, as Sutherland described in [26], the ring-buffer FIFO is hard to design. This difficulty partly brought about the concept of the **micropipeline** as a substitute for the conventional FIFO.

Consequently, it is useful to develop a new structure which is not only as simple as a micropipeline but also as fast and power-efficient as a conventional ring-buffer.

4.4 A new structure

The new FIFO structure is shown in figure 4.7. The data path structure of the FIFO memory is the same as a conventional ring buffer FIFO.

However, in the new design, multiple local control elements are used for every memory element to control the data path, while a conventional ring buffer FIFO

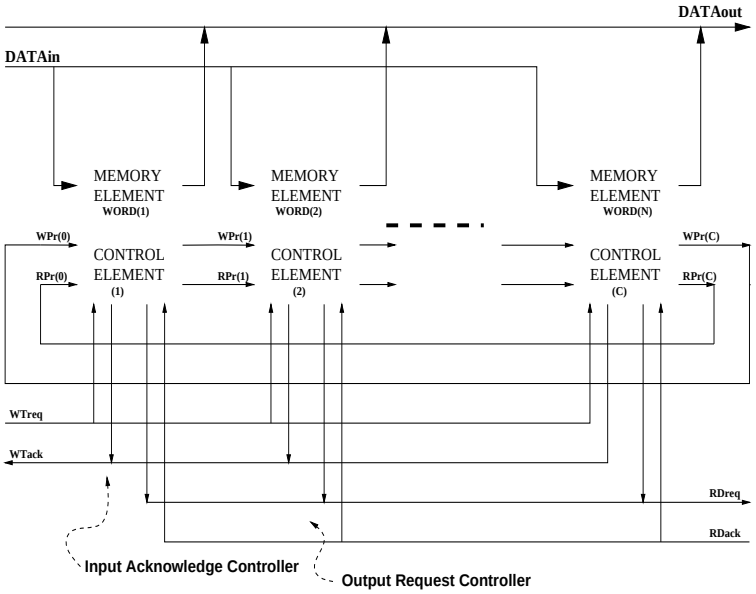


Figure 4.7: Word-Slice FIFO structure

uses a single central controller.

The use of local independent controllers for every memory element allows the FIFO design to scale easily. Each memory element has the same control element as the others, in the same manner as in a micropipeline structure.

There are two data buses, one for input data, the other for output. All of the memory elements share these buses for data input and output signals. Using this structure, input data only need to pass through one memory element to reach to the output. In contrast, the input data to a micropipeline FIFO must pass through all the memory elements, increasing the data delay.

To attain power efficiency, unlike a micropipeline structure, only one of the memory elements and one of the control elements are activated for a data transfer in this new structure. By enabling only one of the write address pointers, *WPr(c)*, in the FIFO (c represents the address of a memory element in the FIFO), only

one memory element is activated for a write operation. By enabling only one of the read address pointer signals, *RPr(c)*, in the FIFO, only one memory element is activated for a read operation.

The input request signal, *WTreq*, is activated to initiate a 4-phase input handshake cycle. The input acknowledge signal, *WTack*, is activated to complete a write operation. The **input acknowledge controller** controls the *WTack* signal by decoding the internal state of all the memory elements.

The output request signal, *RDreq*, is activated to initiate a 4-phase output handshake cycle. The output acknowledge signal, *RDack*, is activated to complete a read operation. The **output request controller** controls the *RDreq* signal by decoding all the internal state of the memory elements.

In the following subsections we will discuss the design and operation of the word-slice FIFO, starting with its interface in section 4.4.1, the word-slice FIFO element in section 4.4.2, the write address pointer in section 4.4.3, the read address pointer in section 4.4.4 and the handshake controller in section 4.4.5.

4.4.1 Interface to a micropipeline

In this work, the 4-phase micropipeline handshake protocol is used to synchronise the input data. The handshake protocol chosen is the **reduced-broad** protocol defined in [5]. To simplify the control and the structure of the design, the concept of delay-matching [5] is applied to this design.

As shown in figure 4.8, decoding the input request signal, *WTreq*, is necessary

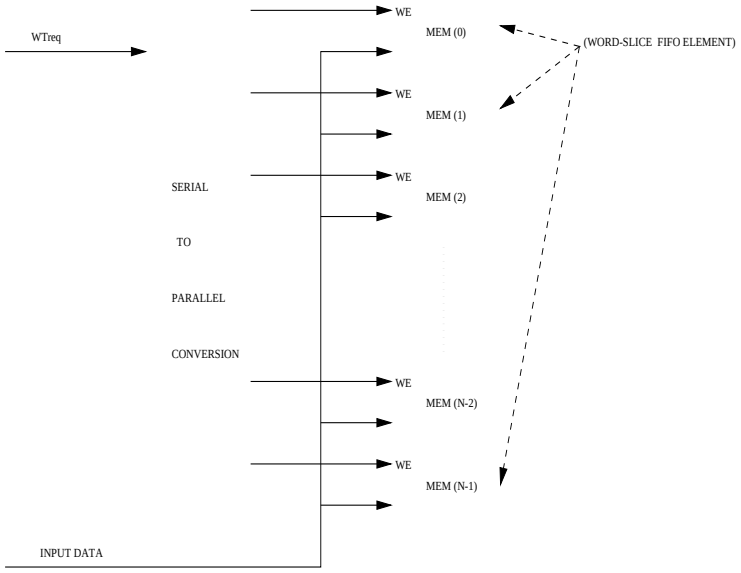


Figure 4.8: Input Request Decoding

to distribute the write control signal to each of the memory elements. A ring counter and a set of comparators are used to build a circular write pointer to the memory. Every cycle of the input handshake increases the value of this ring counter and so moves the write pointer.

The common data input, *INPUT DATA*, will cause a high local power consumption and a long delay as a result of its large capacitive load, when the FIFO element is compared to a micropipeline FIFO stage.

The full signals from each of the FIFO elements are collected to generate a single write acknowledge signal. To match the delay of the bundled full signals from each FIFO element to the acknowledge signal, a bounded delay from the input request signal, *WTreq*, to every full signal is required. If all of the memory elements are in the full state after a write operation, the input acknowledge signal cannot make a transition to low until one of the memory elements is put

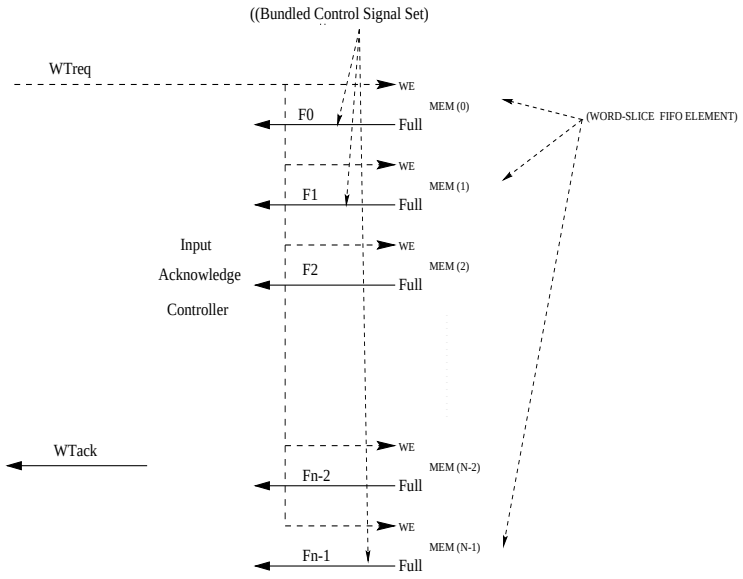


Figure 4.9: Generating the Input Acknowledge Signal

into the empty state by a read operation. By withholding the acknowledge signal transition, the request signal cannot make a transition, thus preventing further write operations.

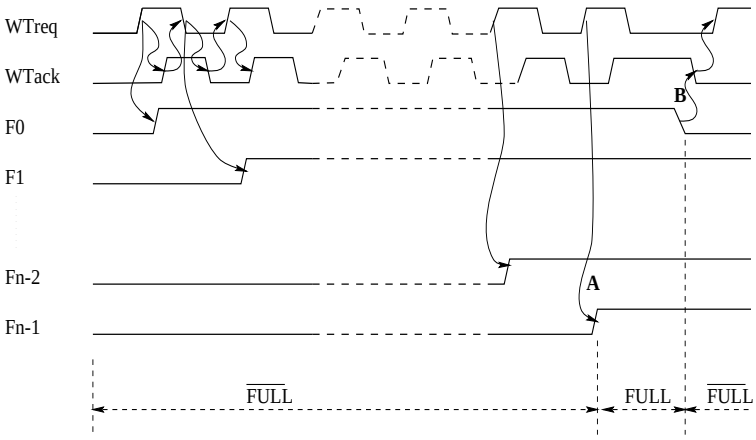


Figure 4.10: Timing diagram for generating the Input Acknowledge Signal

Figure 4.10 presents the timing diagram for the input handshake operation. Initially, the FIFO is empty. When a write is requested by the first rising edge of the *WTreq* signal, the full signal, *F0*, of the first FIFO element is set to high.

The second rising edge of *WTreq* causes the low-to-high transition of the second full signal *F1*. When all the full signals are set to high, transition **A** in the figure, the FIFO is in the **full** state. This full state prevents the *WTack* signal from falling to zero. The 4-phase handshake protocol ensures that the next transition of *WTreq* signal is disabled. Only a read operation to the FIFO can create an empty memory space. After the read operation, one of the FIFO elements is allowed to accept the next input data by returning the full signal, *F0* in this example, to zero: transition **B** in the figure. This allows the *WTack* signal to be set to low. Consequently, the *WTack* signal can be toggled to high again to start the next input handshake cycle.

If more than one of the memory elements are not full (the interval $\overline{\text{FULL}}$ in the figure), there is at least one empty space to be filled with new input data. All of the full signals from each memory element are regarded as a **bundled** set of control signals. In this design, the concept of bundling plays an important role not only for data processing elements but also for the controller design.

4.4.2 Word-slice FIFO element

Figure 4.11 shows the internal structure of the word-slice FIFO element and the external interconnections. Each FIFO element consists of a word of memory and an independent control element. The control element consists of a **read address pointer**, a **write address pointer** and a **handshake controller**.

The data input and output, *Di* and *Do*, are shared with the other FIFO

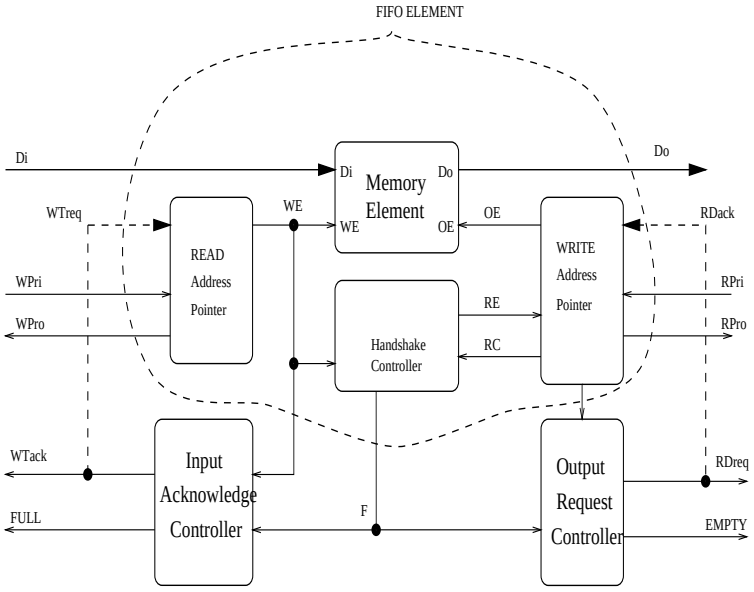


Figure 4.11: Word Slice FIFO Element

elements in a parallel configuration.

There are two ways of controlling the FIFO input. One way is applying the micropipeline control technique. The other way is applying a ring buffer control method.

There are two problems with the first technique as described in section 4.1. A major problem with the micropipeline structure is that every FIFO control unit is dependent on the state of the adjacent control units, this makes the average performance slower. We will show that this unnecessary dependency can be overcome by redesigning a parallel and independent structure.

To remove the dependency on the neighbouring control elements, a handshake controller records the state (full or empty) of a memory element. The state (the signal F in the figure) is provided to the external interface (the input acknowledge controller and the output request controller). The individual states of all the

memory elements in the FIFO are then used to generate the total state of the FIFO: full or empty. The total state is used not only to control the 4-phase handshake protocol, but also to control the interface to the conventional ring-buffer protocol [27].

To avoid the increased power consumption through unnecessary internal switchings, the write address pointer is designed so that there are no simultaneous activations of write control signals *WE*. In addition, the FIFO control element is designed so that only one of the controllers is activated at a time.

4.4.3 Write address pointer

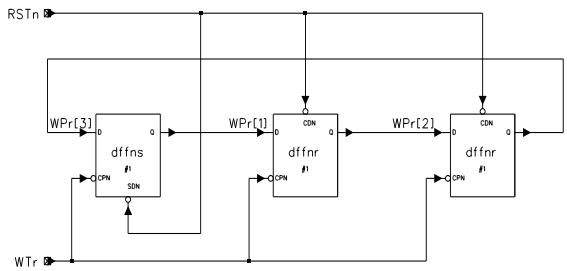


Figure 4.12: A ring counter with three write address pointers

Figure 4.12 shows the ring counter structure used to activate a single write address pointer, *WPr*, at a time. Initially, only *WPr[1]*, among the outputs of the flipflops, is set to high by setting the active low reset signal, *RSTn*, to low. When we set the *RSTn* signal to high, only *WPr[2]* is set to high by the first

falling edge of *WTreq*. Each flipflop in the ring counter can be used as a write address pointer in the FIFO control element, as shown in figure 4.7.

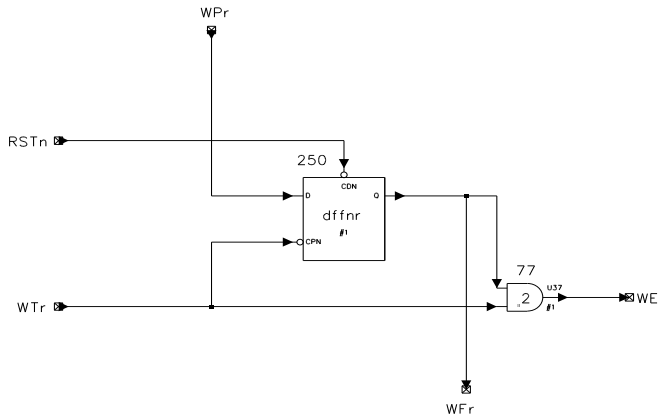


Figure 4.13: Write address pointer

Figure 4.13 shows the circuit diagram of the write address pointer. The circuit function described using STG (Signal Transition Graph) is shown in figure 4.14. The STG was synthesised using the Petrify [16] tool.

In the word-slice FIFO design, the output of the counter, *WFr*, is used as a write address pointer for the current memory element. The input request signal *WTr* can control the write enable signal *WE* only when the address pointer *WFr* is set to high. The input of the counter, *WPr*, is a write address pointer from the previous slot of the FIFO element. The signal *M*, combined with the *WPr* and *WFr* signals, is used to describe a shift operation of the flipflop to move the address pointer along the ring counter.

In response to the input request signal *WTreq*, only one write address pointer, *WFr*, of all the FIFO elements is activated during a cycle of an input handshake.

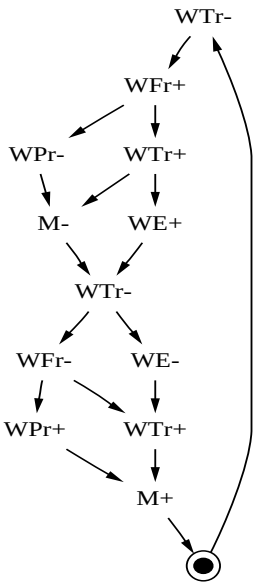


Figure 4.14: STG of the write address pointer

The *WTreq* signal in figure 4.7 is connected to the internal *WTr* signal shown in figure 4.13. The *WTr* signal acts as the clock input to the flipflop. In the figure, the output signal, *WFr*, is connected to the input *WPr* signal of the write address pointer in the next slot of the control element. The *WFr* signal from the final control element, CONTROL ELEMENT (C) in figure 4.7, is connected to the *WPr* signal of the write address pointer in the first control element.

The flipflops in each write address pointer perform a shift operation clocked by the falling edges of the *WTreq* signal. Only one output of the flipflops is set to high at a time because only the *WPr(1)* signal is set to high in the beginning. As a result, a global ring counter is built using the flipflop contained in each individual write address pointer.

We also could use other kinds of counters such as a Gray Coded Counter or a Johnson Counter, to obtain better area efficiency in an implementation. In

these cases, only one output of the counter should be changed at a time to avoid glitches on the decoded output signals.

Alternatively, we can make an independent controller for each FIFO element instead of a global controller for all of the memory elements. In this design, the same memory element and control element can be used in every FIFO element in the FIFO, and so it is easy to vary the depth of the word-slice FIFO.

4.4.4 Read address pointer

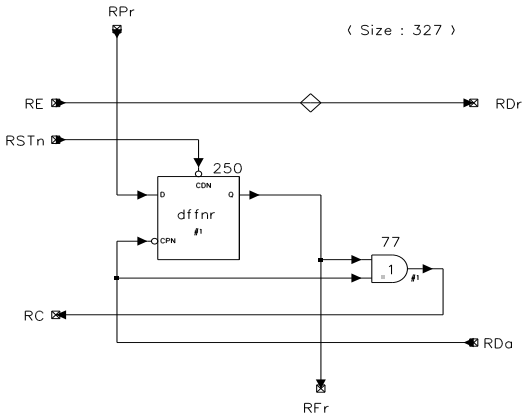


Figure 4.15: Read address pointer

The read address pointer is used to select the memory element from which the data are moved to the output data bus of the FIFO. A shift register is again used to make a simple control structure for the read control. Figure 4.15 shows the circuit for the read address pointer. Figure 4.16 shows the STG for the read address pointer.

RPr, *RFr* and M are the signals used to describe the operation of the flipflop

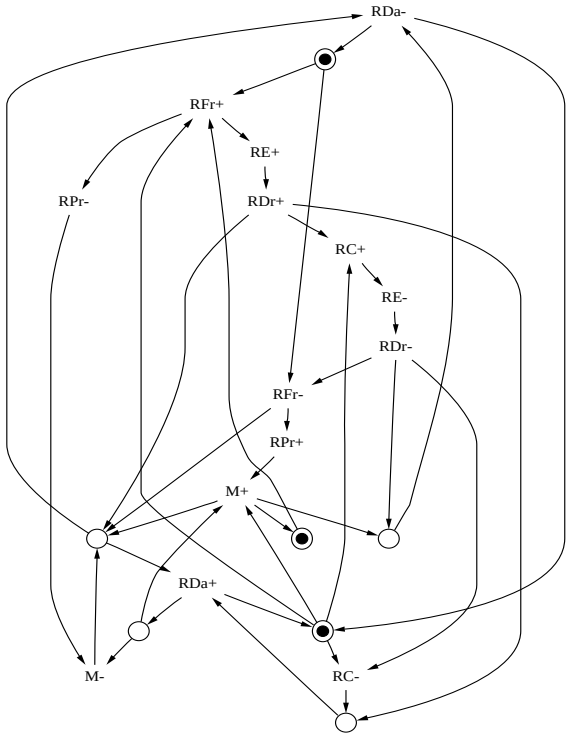


Figure 4.16: STG of read address pointer

which shifts the read address pointer along the ring counter. The input of the flipflop, *RPr*, is a read address pointer from the previous slot of the FIFO element. The input signal, *RFr*, is used to enable a handshake cycle for a read operation from the memory slot which it currently points to.

RE and *RC* are the handshake control signals which are interfaced to the handshake controller shown in the figure 4.11. When the memory is not empty, the *RE* signal is set to high.

RDr and *RDa* are the handshake control signals which are interfaced to the outputs of the FIFO. When the memory is not empty, the *RDr* signal is set to high to request a read operation. After the 4-phase handshake cycle is finished,

the RDa signal returns to zero disabling the current read address pointer signal, *RFr*. The next write operation to the memory is enabled by returning *RC* to zero to indicate that the memory is empty.

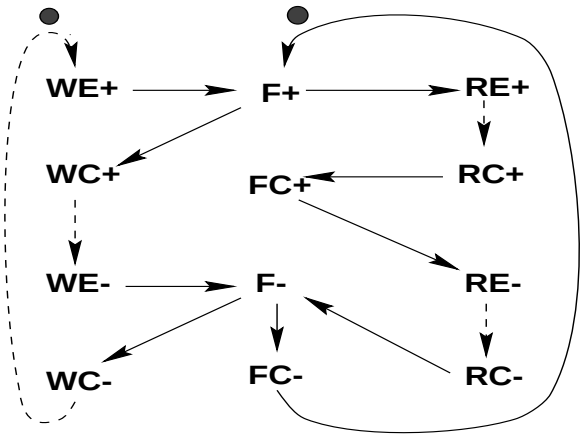
4.4.5 Handshake controller

The handshake controller shown in figure 4.11 has two functions. The first is to control the local handshake between the input and the output. The second is to record the state (full or empty) of the local memory element.

An asynchronous FIFO (in self-timed design) uses a handshake protocol with two pairs of control signals, while a conventional FIFO uses **FULL** and **EMPTY** signals for the interface. Because the handshake controller records the local state of the memory element, the *FULL* signal can be generated simply from all the local state (This is illustrated in figure 4.9: *FULL* = *F0* and *F1* and ... and *Fn-1*).

Figure 4.17 shows the local handshake controller for the asynchronous interface. The upper picture is the signal transition graph of the controller. The lower picture is the resulting circuit synthesised using Petrify [16].

The signals, *WE* (input request) and *WC* (input acknowledge), control the local input handshake control. *RE* (output request) and *RC* (output acknowledge) are used for the local output handshake control. The signal *F* represents the state of the memory element: it is set to high when full, it is set to low when empty. The signal *FC* disables the next write operation until the current read



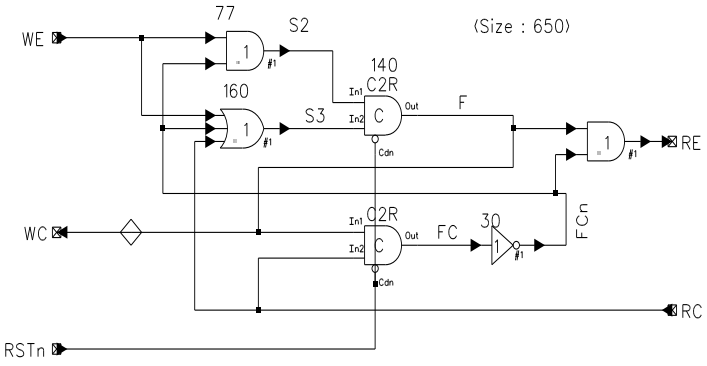


Figure 4.17: Word-Slice FIFO Control for asynchronous interface

operation ends.

We can improve the input cycle time by generating the transition $F+$ directly after the transition $WE+$, as in Figure 4.17. In other words, because the total state of the FIFO, **full**, is generated by combining the individual F, as described in section 4.4.1, the early transition at the signal F reduces the time to generate the total state (**full**). As the full signal generates an acknowledge signal for the

input handshake operation, an improved response time is expected.

Chapter 5

Evaluation 1 : energy consumption

This chapter describes the method and analysis of the energy consumption in the two different FIFO structures.

5.1 Methods of design and verification

5.1.1 Design Flow

VHDL [12] was used to specify the top-level function of the FIFO structures and the test environments. QuickHDL [13] was then used to simulate the VHDL description. All the test vectors for the **PowerMill** simulation were generated using the VHDL simulator.

Signal Transition Graphs (STG) [7, 8] were used to specify the function of the

asynchronous controllers in this design, and **Petrify** [14, 15, 16] was used for the verification of the controllers' function. After the design of the control circuits, they were converted to VHDL to verify the top-level functionality.

All the VHDL descriptions were converted into schematics. All the designs were converted into physical layouts using Compass Chip Compiler [28, 29] and Composition Editor [30]. 0.35 micrometer technology was used for the layout.

5.1.2 Analysis of the energy efficiency

The primary purpose of this analysis is to show the advantage of the new FIFO structure regarding energy efficiency. The second purpose is to derive a reference for the optimal choice of structure for various applications.

A four stage process was employed to obtain comparative measurements.

- Stage one :

  To simplify the analysis, primitive models were defined. These models allow the characteristics of the two different FIFO structures to be represented by simple equations.

- Stage two:

  A simple simulation environment was constructed from which results could easily be obtained. A test control circuit was designed such that it consumed little energy and so had a negligible effect on the overall measurements.

- Stage three:

Basic measurements were performed. Energy consumptions for a data transfer was measured. Data values were obtained from post-layout simulation using PowerMill [31]. The operating voltage was set to 3.3 volts.

- Stage four:

  Energy consumption characteristics were analysed using the resulting data. Several graphs will be presented to simplify the analysis.

Finally, methods of choosing an optimal structure will be described.

5.2 Primitive models

To ease the process of analysing measurements, we need to define a simple but effective model for a FIFO's energy consumption. From this model, simple equations can be used to describe different FIFO structures and different memory sizes.

In CMOS digital logic, energy consumption, E, can be calculated as:

$$\begin{aligned} E &= V \int_0^\infty \frac{Q}{t} dt \\ &= \frac{1}{2} C V^2 \end{aligned} \tag{5.1}$$

C is a load capacitance in CMOS logic. The capacitor is charged with Q coulomb to V volts.

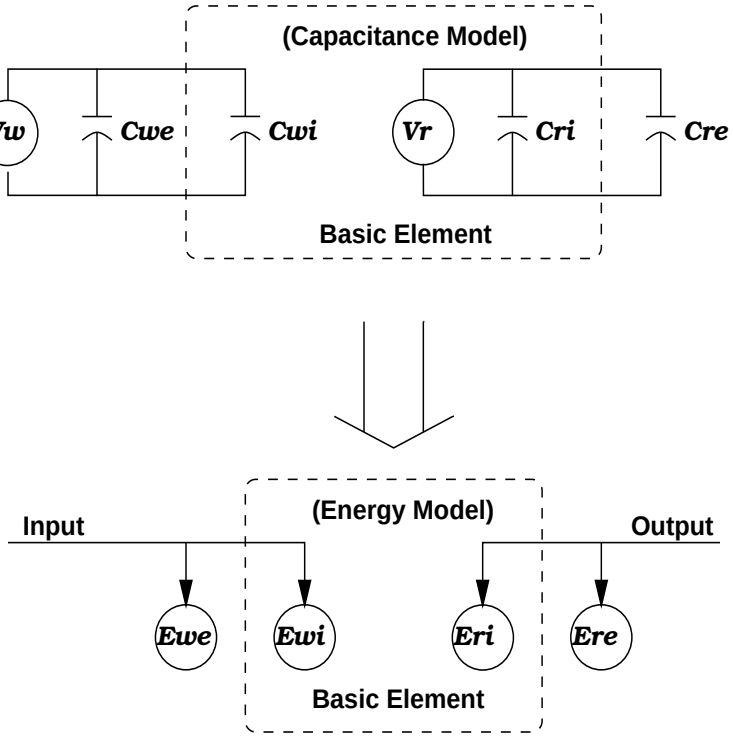


Figure 5.1: Energy model

We can represent an element in a CMOS design using a simple energy consumption model as shown in figure 5.1.

The capacitance model is illustrated in the upper picture.

- Vw: the electrical voltage source for a write operation.
- Cwe: the external equivalent capacitive load at the input side.
- Cwi: the internal equivalent capacitive load at the input side.
- Vr: the electrical voltage source for a read operation.
- Cre: the external equivalent capacitive load at the output side.
- Cri: the internal equivalent capacitive load at the output side.

Using equation (5.1), each capacitive element can be converted into an energy dissipation element, when the voltage source, V, is constant. Therefore the capacitance model can be converted to the energy dissipation model shown in the lower picture in figure 5.1.

Employing the above assumption allows us to remove the need to measure the capacitance, and so simplifies the process of obtaining experimental results.

5.3 Representing different FIFO structures

To calculate the relative power consumptions of different FIFOs operating with the same performance, the energy used to transfer a data item from the input of a FIFO to the output must be measured for each FIFO structure.

If we assume that the power consumption of a FIFO is linear with its depth, we can derive an equation for the power consumption of the FIFO from the measurements at two different FIFO depths.

To enable the analysis of energy consumption for two different structures of FIFOs, four FIFO designs were required. Two of the FIFOs have a micropipeline structure and the other two have the word-slice structure. Two candidate memory sizes were chosen for each structure to enable the derivations of linear equations, by which the characteristics of the FIFOs can be expressed.

A word-slice FIFO will be denoted as WS in the equations, whereas a micropipeline FIFO will be denoted as UP.

A FIFO memory size is defined by two variable parameters: word width, N,

and FIFO depth, C.

The characteristics of input data can have a large impact on the power consumption. For this analysis, we represent the number of toggled data bits for a data transfer by the variable n.

The following symbols are used in the analysis.

- Et : The energy for a data transfer from the input to the output of a FIFO.
- Ew : The energy for writing single data to an empty FIFO.
- Er : The energy for reading data from a FIFO with single valid data.
- Em : The energy consumed by memory elements for a data transfer from the input to the output of a FIFO.
- Ec : The energy consumed by control elements for a data transfer from the input to the output of a FIFO.

These symbols can be used to represent the characteristics of FIFO structures as follows.

- $Et(n, N, C, WS)$: The energy consumption of a FIFO for a data transfer, when n input data are toggled, a memory element consists of N-bit memory, the total number of memory elements is C and the FIFO has **Word-Slice** structure. N, n and C are positive integers. The total number of control elements is the same as the number of memory elements.

- $Ec(n, N, C, UP)$: The energy consumed by control elements in a FIFO, for a data transfer, when n input data are toggled, a memory element consists of N-bit memory, the total number of memory elements is C and the FIFO has **micropipeline** structure. N, n and C are positive integers. The total number of control elements is the same as the number of memory elements.

5.4 Test environments

5.4.1 Scope of the experiments

We determined the energy needed for a data transfer from the input of a FIFO to the output. To simplify the calculation of power consumption, the data transfer rate was kept constant.

In the experiment, the parameters were chosen as follows for the two FIFO structures.

- Memory size

  In this experiment, the number of bits of memory in a memory element word, N, was set to 32. Each FIFO structure was analysed using two different FIFO depths C: 16 and 3.

- Input data

  Each FIFO structure was analysed using the two extreme numbers of toggled input data, n : 16 (half the input data toggled) and 0 (none of the input data toggled).

5.4.2 Test control circuit

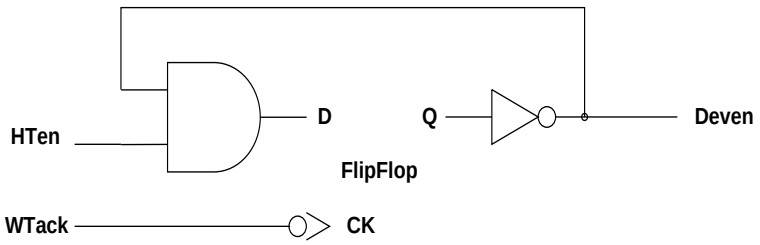


Figure 5.2: Test input pattern generator

The test circuit should not only provide enough functionality to acquire the data values but should also be small enough to have a negligible influence on the experimental results.

In this experiment, the flipflop based circuit shown in figure 5.2 was used to generate toggled patterns of input data. The same test circuit was used for both FIFO structures. When *HTen* signal is set to high, *Deven* signal is toggled by every falling edge of *WTack* signal; when *HTen* is set to low, the *Deven* signal is not toggled. An input handshake control signal, *WTack*, is used as a clock signal for the flipflop. The output *Deven* is connected to every other data input line of a FIFO to toggle half of the data. The other data inputs of the FIFO which are not toggled are set to low.

Two different input data patterns can be generated by this simple pattern generator.

- Zero data toggled input: no data are changed.
- Half the data toggled input: half of the data are changed.

The zero data toggled input can be used to measure the energy consumed by the control elements because there are no activations in the data path composed of memory elements.

The half data toggled can be used to measure the energy consumed by a FIFO when we assume random input data because the probability of the data change is 0.5 for random data.

5.5 Measurements and analysis

5.5.1 Energy consumption by read and write operations

The energy consumption measured for a write and a read operation in each struc-

Table 5.1: Experimental Results

Micropipeline FIFO (UP)		Word-Slice FIFO (WS)	
Test Conditions	DATA (pJ)	Test Conditions	DATA (pJ)
$Ew(16, 32, 16, UP)$	363.	$Ew(16, 32, 16, WS)$	100.
$Er(16, 32, 16, UP)$	25.1	$Er(16, 32, 16, WS)$	83.5
$Ew(0, 32, 16, UP)$	154.	$Ew(0, 32, 16, WS)$	56.4
$Er(0, 32, 16, UP)$	13.5	$Er(0, 32, 16, WS)$	67.0
$Ew(16, 32, 3, UP)$	70.3	$Ew(16, 32, 3, WS)$	70.3
$Er(16, 32, 3, UP)$	21.8	$Er(16, 32, 3, WS)$	29.4
$Ew(0, 32, 3, UP)$	31.7	$Ew(0, 32, 3, WS)$	30.0
$Er(0, 32, 3, UP)$	10.6	$Er(0, 32, 3, WS)$	29.4

ture are presented in table 5.1. The acquired data should be viewed in a relative sense: the measured data values varied by up to 20 percent as the physical layout changed.

5.5.2 Total energy consumption for a single data transfer

A data transfer in a FIFO consists of a write operation to the FIFO and a read operation from it. Therefore the total energy consumed for a data transfer in a FIFO is the sum of energy consumed by a write and a read operation to the FIFO:

$$Et = Ew + Er$$

Using the data in section 5.5.1, the total energy consumptions in the two structures of FIFOs were derived. The results are presented in table 5.2.

Table 5.2: Total energy consumption

Micropipeline FIFO (UP)		Word-Slice FIFO (WS)	
Test Conditions	DATA (pJ)	Test Conditions	DATA (pJ)
$Et(16, 32, 16, UP)$	388.	$Et(16, 32, 16, WS)$	184.
$Et(0, 32, 16, UP)$	168.	$Et(0, 32, 16, WS)$	123.
$Et(16, 32, 3, UP)$	92.1	$Et(16, 32, 3, WS)$	99.7
$Et(0, 32, 3, UP)$	42.3	$Et(0, 32, 3, WS)$	59.4

If we assume that the energy consumption in a FIFO is linear to the depth C, we can derive the equation for $Et(n = 16, N = 32, C, UP)$ from $Et(16, 32, 16, UP)$ and $Et(16, 32, 3, UP)$ labelled as equation 5.2. An equation for $Et(n = 16, N =$

$32, C, WS)$ can also be derived from $Et(16, 32, 16, WS)$ and $Et(16, 32, 3, WS)$ labelled as equation 5.3.

Because n and N are constants,

$$Et(n = 16, N = 32, C, UP) = \frac{388 - 92.1}{16 - 3}C + 92.1 - 3 \times \frac{388 - 92.1}{16 - 3} \quad \text{(pJ)}$$

$$= 22.8C + 23.8 \quad \text{(pJ)} \tag{5.2}$$

$$Et(n = 16, N = 32, C, WS) = 6.45C + 80.4 \quad \text{(pJ)} \tag{5.3}$$

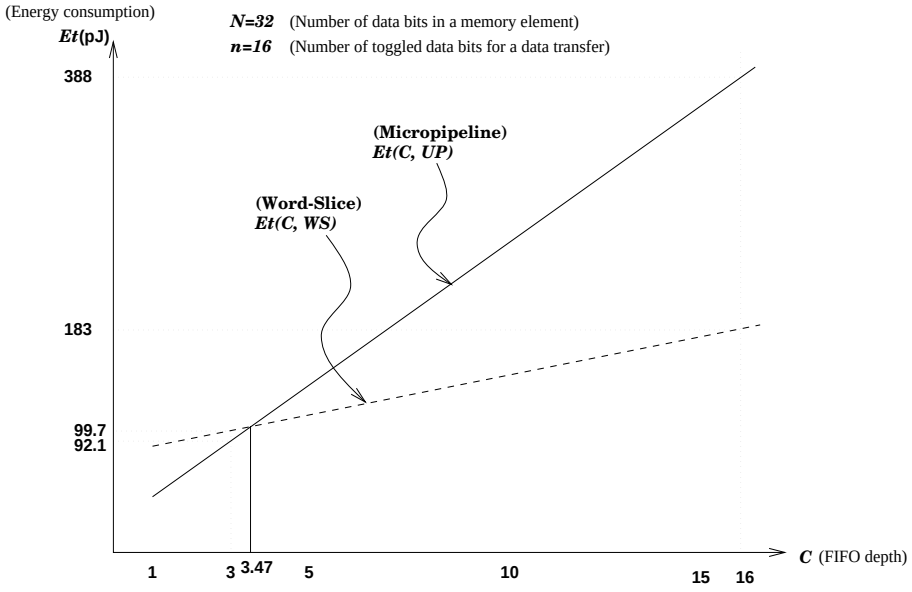


Figure 5.3: Total energy consumption

From equations (5.2) and (5.3), we can predict that the word-slice FIFO consumes less energy than the micropipeline FIFO if

$$C > 3.47$$

.

Figure 5.3 shows the total energy consumption characteristics of two different FIFO structures.

5.5.3 Energy consumed by the control elements

The total energy consumed for a data transfer in a FIFO is the sum of the energies consumed by the control element and the memory element:

$$Et = Em + Ec$$

if the energy consumed by the test circuit is negligible.

If none of the data input bits are toggled, $Em(0, N, C, UP)$ and $Em(0, N, C, WS)$ can be assumed to be zero because none of the memory elements are activated. Consequently, the energy consumed by the control elements is the same as the total energy consumption:

$$Ec(0, N, C, UP) = Et(0, N, C, UP)$$

$$Ec(0, N, C, WS) = Et(0, N, C, WS)$$

From previous measurements in section 5.5.1, the energies consumed by the control elements are as follows.

- Micropipeline structure

  $Ec(0, 32, 16, UP) = 167.6$ pJ

  $Ec(0, 32, 3, UP) = 42.3$ pJ

- Word-slice structure

$Ec(0, 32, 16, WS) = 123.4$ pJ

$Ec(0, 32, 3, WS) = 59.4$ pJ

From the data above, the energies consumed by control elements, when the depth of the FIFOs are varying, can be expressed using the following equations.

$$Ec(n = 0, N = 32, C, UP) = 9.64C + 13.4 \quad \text{(pJ)} \tag{5.4}$$

$$Ec(n = 0, N = 32, C, WS) = 4.92C + 44.6 \quad \text{(pJ)} \tag{5.5}$$

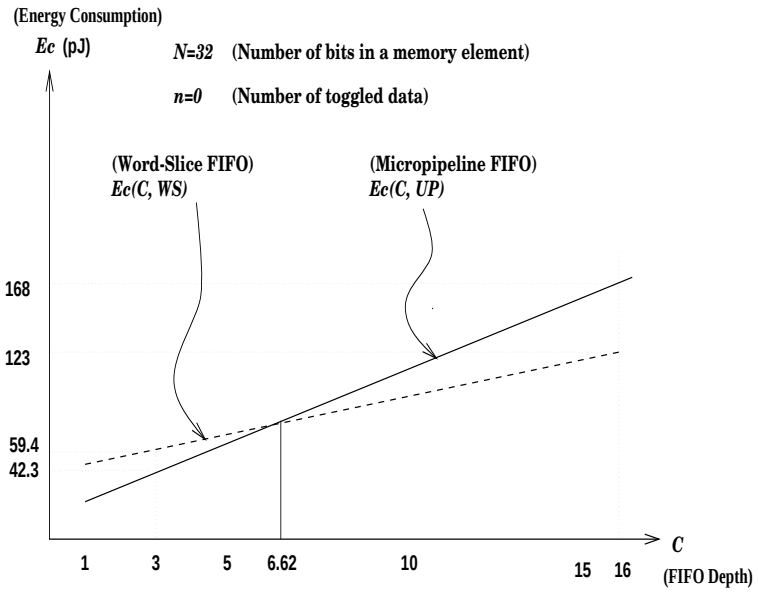


Figure 5.4: Energy consumption by control elements

From equations (5.4) and (5.5), we can predict that the control elements in the word-slice FIFO consume less energy than the control elements in the micropipeline FIFO if

$$C > 6.62$$

Figure 5.4 shows the energy consumption characteristics of the control elements for the two different FIFO structures.

5.5.4 Energy consumption by the memory elements

The total energy consumed for a data transfer in a FIFO is

$$Et = Em + Ec$$

The energy consumed by memory elements is

$$Em = Et \text{ - } Ec$$

All the control element perform the same operation for a data transfer regardless of the change of the input data, and so $Ec(n, N, C, UP)$ and $Ec(n, N, C, WS)$ are constants for all n.

From equations (5.2) and (5.4),

$$Em(n = 16, N = 32, C, UP) \quad = \quad 13.1C + 10.4 \quad \text{(pJ)} \tag{5.6}$$

From equations 5.3 and 5.5,

$$Em(n = 16, N = 32, C, WS) \quad = \quad 1.53C + 35.7 \quad \text{(pJ)} \tag{5.7}$$

From equations (5.6) and (5.7), we can predict that memory elements in the word-slice FIFO consume less energy than control elements in the micropipeline FIFO if

$$C > 2.17$$

Figure 5.5 shows the energy consumption characteristics of memory elements for the two different FIFO structures. The energy consumed by the memory elements in a micropipeline FIFO is dominated by the FIFO depth because the number of data bits toggled in the memory elements is proportional to the depth.

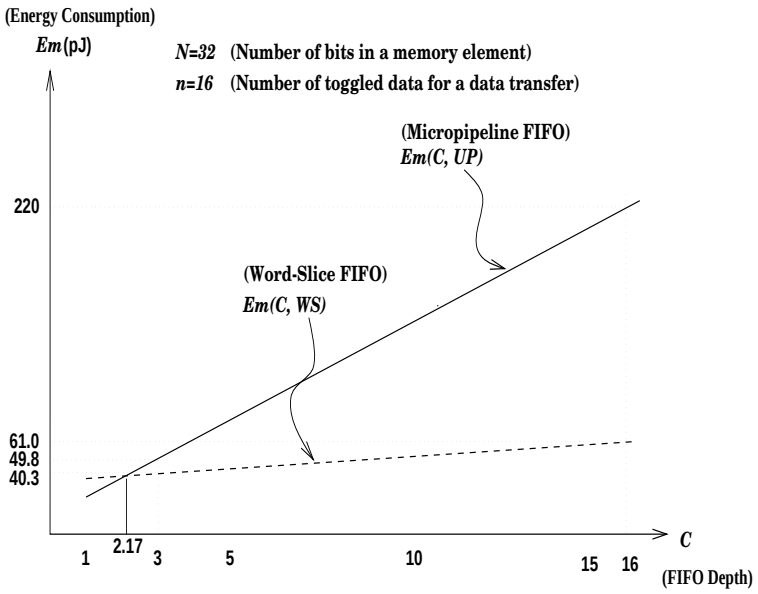


Figure 5.5: Energy consumption by memory elements

In contrast, the energy consumed by the memory elements in the word-slice structure is not affected by the FIFO depth because the number of data bits toggled in the memory elements is not changed by the depth. However, as the depth increases, the sum of the input and the output load capacitances of the memory elements of the word-slice FIFO also increases. The figure shows that these incremental capacitances cause a smaller increment of energy consumption than that caused by the changes of the data in the memory elements.

5.6 Choice of structure according to input data characteristics

It is necessary to analyse the characteristics of the energy consumption not only with the different depth of the memory elements but also with the different characteristics of the input data.

Consider an environment where the input data to a FIFO are frequently changed. In this case, we can set the value n to half of N to represent random data.

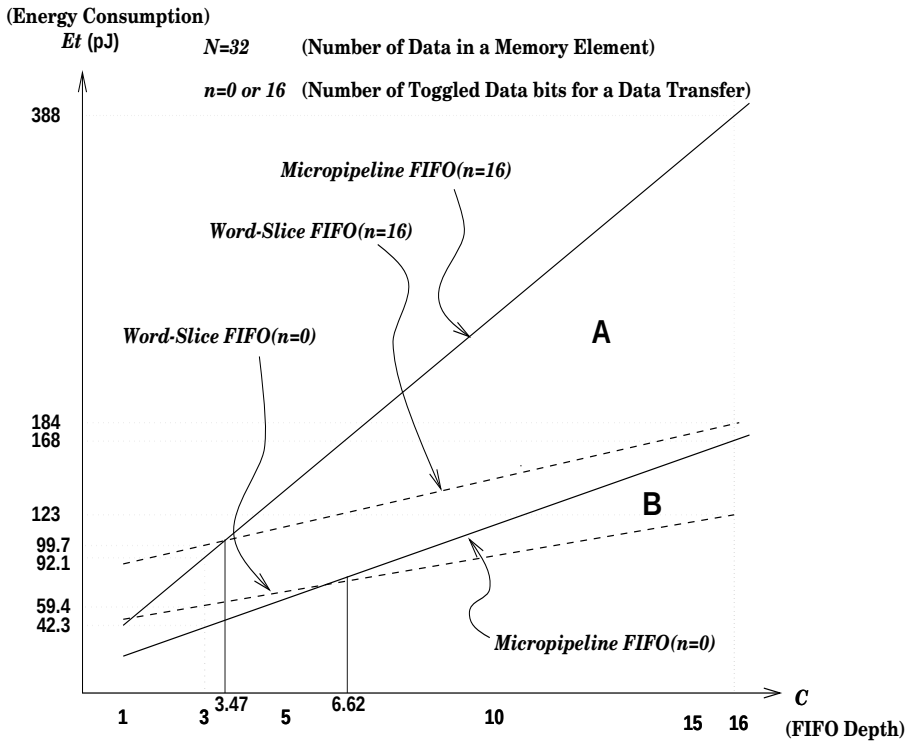


Figure 5.6: Total energy consumption and the characteristics of input data

Consider an environment where the input data to a FIFO are rarely changed. We can set the value n to zero in this case.

From equations (5.2), (5.3), (5.4) and (5.5), we can express the characteristics in the graph shown in figure 5.6. From the figure, it can be seen that the word-slice FIFO has better characteristics in the region **A** when the input data are random. The word-slice FIFO also has better characteristics in the region **B** when there are few changes of data.

5.7 Comparing the energy consumptions for FIFO elements in different structures

Figure 5.7 presents a comparison of the energy consumed by the memory elements and the control elements in the two different FIFO structures. For comparison,

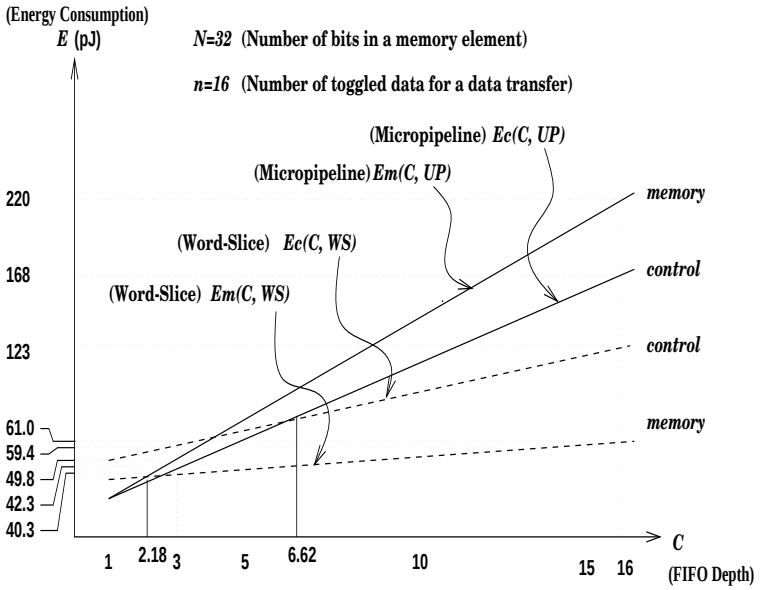


Figure 5.7: Energy consumed by memory and control elements

half of the input data are assumed to be toggled for a data transfer.

In a micropipeline structure, the memory elements consume more power than the control elements. In a word-slice FIFO, the control elements consume more power than the memory elements.

When each FIFO has 16 words of 32-bit memory, the memory elements in the word-slice FIFO consume only 27-percent of the energy needed by the memory elements in the micropipeline FIFO to transfer the data. In this case, the control elements in the word-slice FIFO consume only 49-percent of the energy needed by the control elements in the micropipeline FIFO to transfer the data.

Chapter 6

Evaluation 2 : performance

The performance of the two different FIFO structures are compared in this chapter. Two FIFO depths are chosen as the candidates for each FIFO structure. We then predict the performance variations according to the variance of the FIFO depth with these two sets of measured data.

6.1 Test environments

The performance was measured using two factors:

- FIFO structure: word-slice/micropipeline structure
- FIFO depth: 3/16 words

To control the input and the output handshake operations, the same test circuits were used for both FIFO structures. The input and output operation can be controlled in three modes:

- Fast cycle
- Slow cycle
- Halt

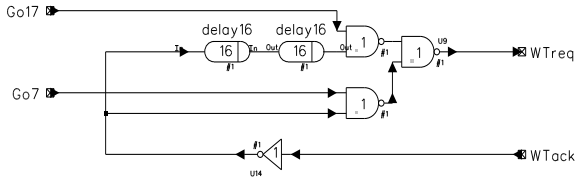


Figure 6.1: Input request controller

The input request control circuit shown in figure 6.1 was used for the input control. The input handshake cycle is in fast mode if *Go7* signal is set to high and if *Go17* signal is set to low. The input handshake cycle is in slow mode if the *Go7* signal is set to low and if the *Go17* signal is set to high. The input handshake operation is disabled if both *Go7* and *Go17* signals are set to low.

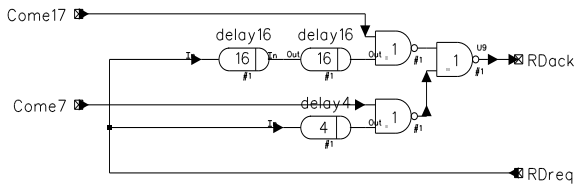


Figure 6.2: Output acknowledge controller

The acknowledge control circuit shown in figure 6.2 was used for the output control. The output handshake cycle is in fast mode if *Come7* signal is set to high and if *Come17* signal is set to low. The output handshake cycle is in slow

mode if the *Come7* signal is set to low and if the *Come17* signal is set to high. The output handshake operation is disabled if both *Come7* and *Come17* signals are set to low.

6.2 Measurements and analysis

The data propagation delay and the data transfer cycle time of the two different FIFO structures will be analysed in this section.

Table 6.1: Performance Measurements

Micropipeline FIFO (UP)		Word-Slice FIFO (WS)	
Test Conditions	DATA (ns)	Test Conditions	DATA (ns)
$Tcyc(C = 3)$	3.5	$Tcyc(C = 3)$	3.7
$Tcyc(C = 16)$	3.5	$Tcyc(C = 16)$	4.2
$Tpd(C = 3)$	2.6	$Tpd(C = 3)$	2.8
$Tpd(C = 16)$	12.4	$Tpd(C = 16)$	3.8

Table 6.1 presents the experimental data. $Tcyc$ is the data transfer cycle time. Tpd is the data propagation delay from the input of a FIFO to the output. C denotes the FIFO depth. UP denotes the micropipeline FIFO. WS denotes the word-slice FIFO.

6.2.1 Data propagation delay

Figure 6.3 shows the difference between the data propagation delays of the two FIFO structures. We can draw a straight line between two different sizes of the

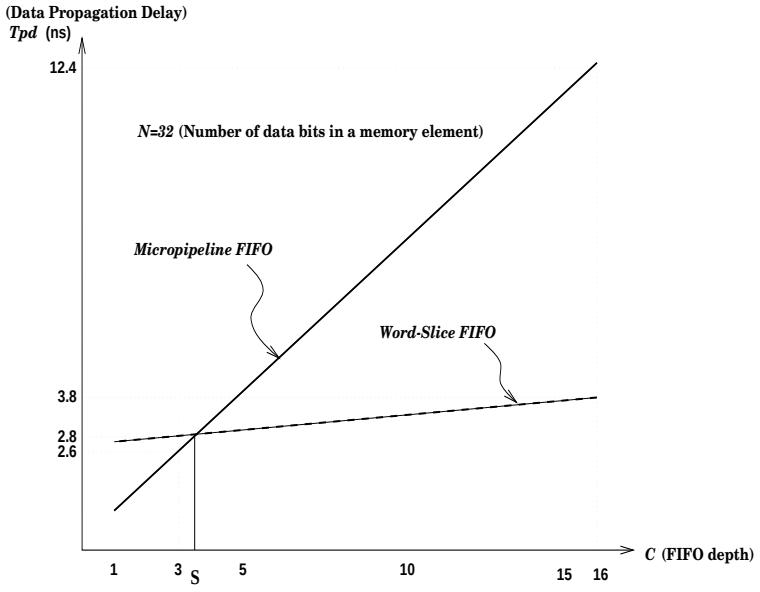


Figure 6.3: Data propagation delay

FIFO to predict the performance for different FIFO sizes. This is based on the assumption that the propagation delay is linear with the FIFO size for the same structure because the FIFOs are scalable so the capacitive loads are proportional to the FIFO sizes. Equations for the propagation delays can be derived from the measured data in 6.1 as follows.

$$Tpd(C, UP) = 0.75C + 0.34 \quad \text{(ns)} \tag{6.1}$$

$$Tpd(C, WS) = 0.077C + 2.6 \quad \text{(ns)} \tag{6.2}$$

From these equations, we can see that the incremental propagation delay (0.75 ns) of the micropipeline FIFO structure is about nine times lager than the incremental delay (0.077 ns) of the word-slice FIFO structure.

From figure 6.3, we can expect a word-slice FIFO to have a better performance

for data propagation when the depth is greater than three. When the FIFO depth is 16, the word-slice FIFO is three times faster than the micropipeline FIFO in the propagation delay.

The serial data path of the micropipeline structure slows down the data propagation significantly while the parallel data path of the word-slice structure leads to a better performance.

6.2.2 Cycle time

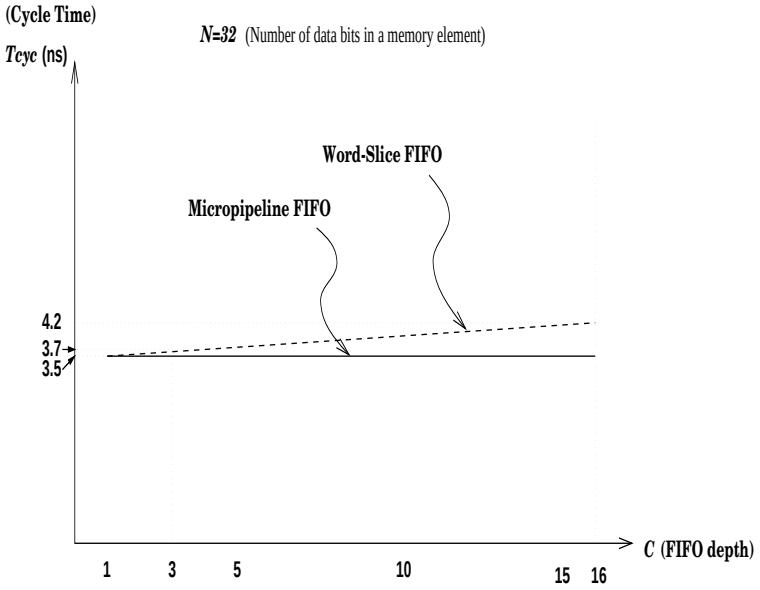


Figure 6.4: Data transfer cycle time

Figure 6.4 shows the difference between the data transfer cycle times of the two FIFO structures. We can draw a straight line between the two different sizes of the FIFO to predict the performance for different FIFO sizes. This is based on the assumption that the cycle time is linear to the FIFO size in the same structure.

From the figure, we can see that the word-slice FIFO is slightly inferior to the micropipeline structure in data transfer cycle time. When the FIFO depth is 16, the micropipeline FIFO is 1.14 times faster than the word-slice FIFO in cycle time while the word-slice FIFO is three times faster than the micropipeline FIFO in propagation delay.

The incremental capacitive load at the common data bus in the word-slice FIFO causes the incremental data transfer cycle time characteristic.

Chapter 7

Conclusion

A new asynchronous FIFO structure has been developed. Three problems with current FIFO designs have been addressed: power consumption, performance and simple structure for VLSI design.

To simplify the VLSI design process, the new FIFO structure is scalable in its depth: it uses a **word-slice FIFO** structure, and it overcomes the difficulty of FIFO design which was described by Sutherland [26].

Low power consumption was achieved using two methods. The first method is to control the memory so that only the memory elements selected by the read or write pointers are activated. The second method is to minimise the activity of the control elements.

High performance was achieved by arranging all memory words in the FIFO in parallel. This structure requires the input data to pass through only one memory element to reach the output.

7.1 Comparison

The power consumed by the word-slice FIFO structures tested in this thesis was less than that of conventional micropipeline FIFO structures. However, the best relative results were obtained if the input data is random. In this case, the power consumed by the word-slice FIFO is only 48 percent of that of micropipeline FIFO. Even when the data are correlated, the power consumed by the word-slice FIFO is only 73 percent of that of the micropipeline FIFO.

From these observations, we can observe that the power consumed by the control element in a word-slice FIFO is smaller than that of a micropipeline FIFO. Consequently, we need to minimise the activity not only of the memory elements but also of the control elements in micropipelines.

The conventional micropipeline FIFO has a slightly better data transfer cycle time than the word-slice FIFO. The micropipeline FIFO is 1.14 times faster than that of the word-slice FIFO when the FIFO depth is 16.

However, the word-slice FIFO has a significantly improved data propagation delay compared to the micropipeline FIFO. The word-slice FIFO is 3.26 times faster than the micropipeline FIFO when the FIFO depth is 16.

In circumstances where a slight difference in data transfer rate causes critical disadvantages, a micropipeline FIFO is recommended. A micropipeline FIFO is also better when the buffer depth is less than three. In all other cases, a word-slice FIFO has benefits in power consumption and performance when the FIFO

depth is greater than three.

We can expect that a conventional ring-buffer FIFO has similar performance and power consumption characteristics to the word-slice FIFO because of the similarity of the data path. However, due to the difference in control structures and complexity, the word-slice FIFO has clear advantages. The simple structure of the word-slice FIFO eases the VLSI design process.

The power consumed by the control elements in a micropipeline has been shown to be significant when it is compared to the power consumed by the data path without processing elements. When the data are correlated, we can expect that most of the power consumption is caused by the control elements in an asynchronous system.

7.2 Future Work

Further work is needed to take maximum advantage of the word-slice FIFO structure.

The first step is to build an automatic process from a FIFO specification to a physical layout. This automatic process can be used to build a data-path library so that a designer can use a FIFO by simply specifying the width and the depth.

The second step is to optimise the handshake controller for improved energy efficiency. There can be two ways that this can be achieved. The one is to

minimise the activity of the CMOS gates in the handshake controller, the other is to optimise the transistor size.

Bibliography

[1] S.B. Furber, J.D. Garside, S. Temple, P. Day, N.C. Paver, J.V. Woods, "AMULET1 : An Asynchronous ARM Microprocessor", IEEE Transactions on Computers, vol. 46, No. 4, pp. 385-398, April 1997.

[2] Amulet Group, "Amulet2e" data sheet from the *University of Manchester*, 1996

[3] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, N.C. Paver, "AMULET2e : An Asynchronous Embeded Controller", Proceedings Async '97 pp. 290-299 IEEE Computer Society Press, April 1997.

[4] S.B. Furber, "Asynchronous Logic", IberChip, Sao Paulo, Brazil, 12 Feb 1996, http://www.cs.man.ac.uk/amulet/publications/papers.html.

[5] S.B. Furber, "A small Compendium of 4-Phase Micropipeline Latch Control Circuits", University of Manchester, Nov 1997.

[6] A.J. Martin, "The limitations to delay-insensitivity in asynchronous circuits", Sixth MIT Conference on Advanced Research in VLSI, Cambridge, Massachusetts, 1990

[7] Chu T.A., "On the models for designing VLSI asynchronous digital systems", North-Holland, INTEGRATION, THE VLSI JOURNAL VOL. 4, 1986, PP. 99-113

[8] Chu T.A., Leung C.K.C. and Wanuga T.S., "A design methodology for concurrent VLSI systems", Proceedings of ICCD, 1985.

[9] C.J. Lin, S.M. Reddy, "On delay fault testing in logic circuits", IEEE Transactions on Computer Aided Design, vol. 6, September 1987, pp.694-703.

[10] D.A. Huffman, "The synthesis of sequential switching circuits", J. Franklin Institute, vol. 257, March/April 1954, pp.161-190.

[11] D.E. Muller, W.C. Bartky, "A theory of asynchronous circuits", Annals of Computing Laboratory of Harvard University, 1959, pp.204-243.

[12] IEEE Computer Society Publication Department, "VHDL Language Reference Manual, IEEE-STD-1076", 1987.

[13] Mentor Graphics "QuickHDL User's and Reference Manual", Software Version 8.5_4.

[14] J.L. Perterson "PETRI NET THEORY AND THE MODELING OF SYSTEMS", 1981.

[15] J. Cortadella, M. Kishinervsky, A. Kondratyev, L. Lavagno, E. Pastor and A. Yakovlev, "Synthesizing Petrinets from State-Based Models", June 12, 1997.

[16] J. Cortadella, M. Kishinervsky, A. Kondratyev, L. Lavagno, E. Pastor and A. Yakovlev, "Petrify User's manual", June 12, 1997.

[17] Liu J., "The design of asynchronous multiplier", MSc thesis, University of Manchester, June 1995 pp.27-35.

[18] Liu J., "The design of asynchronous multiplier", PhD thesis, University of Manchester, June 1997, pp.97-120.

[19] M.B.Josephs, "Speed-Independent Design of a Toggle", EXACT Workshop on Asynchronous Controllers and Interfacing Leuven, Belgium, September, 1992.

[20] Mark R.G. and Lenneth S., "Bubbles Can Make Self-Timed Pipelines Fast", Journal of VLSI Signal Processing, 2, 1990, pp. 139-148.

[21] Michael K., Luciano L. and Peter V., "The Systematic Design of Asynchronous Circuits", ICCAD '95 tutorial, pp. 22-25, pp.75-91.

[22] Molnar C.E. et al, "Synthesis of delay-insensitive modules", Proceedings of the 1985 Chapel Hill Conference on VLSI, (Computer Science Press, Rockville, MD, 1985).

[23] N. C. Paver, "The Design and Implementation of an Asynchronous Microprocessor", PhD thesis, University of Manchester, 1994, pp. 48-53.

[24] S. B. Furber, P. Day, "Four-Phase Micropipeline Latch Control Circuits", IEEE Transaction on VLSI Systems, vol. 4, June 1996, pp. 247-249.

[25] Scott Hauck, "Asynchronous Design Methodologies : An Overview", Proceedings of IEEE, vol. 83, No. 1, pp. 69-93, January 1995.

[26] Sutherland I.E., "MICRO-PIPELINES", The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, June 1988, pp. 720-738.

[27] TI, "The TTL Data Book ", Vol. 2, 1987, pp. 2-225 - 2-264.

[28] COMPASS, "COMPASS Design Automation: LOGIC ASSISTANT", V8R3, 1991.

[29] COMPASS, "COMPASS Design Automation: CHIP COMPILER", V8R3, 1991.

[30] COMPASS, "COMPASS Design Automation: COMPOSITION EDITOR", V8R3, 1991.

[31] EPIC, "EPIC PowerMill User Manual", Release 3.4, 1996.

Appendix A

Word-slice FIFO

A.1 VHDL

A.2 Test environment

A.3 Top level FIFO structure

A.4 FIFO element

A.5 Physical layout

```
------------------------------------------------------------
-- Asynchronous to Synchronous interfaceing FIFO
-- WS_FIFO.vhd
-- KyoungKeun Yi, 4/JUN/1998

-- Word-Slice Archetecture
------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use work.WS_FIFO_package.all;

-- ================================================================

entity WS_FIFO is
      generic (WIDTH : integer ;
            WTreq2ack : time;     -- WTreq to WTack cycle time
            DEPTH : integer );
      port (
            -- Data Input
            Di : in  std_logic_vector ((WIDTH - 1) downto 0);
            -- Data Output
            Do : out std_logic_vector ((WIDTH - 1) downto 0);

            -- Asynchronous Req. input
            WTreq : in std_logic;
            -- Asynchronous Ack. Output
            WTack : out std_logic;

            RDreq : out std_logic;
            RDack : in std_logic;

            FULL  : out std_logic;
            EMPTY : out std_logic;

            RSTn : in std_logic
            );
end WS_FIFO;


--
```



```
------------------------------------------------------------
architecture WORD_SLICE_L of WS_FIFO is
      signal  DWTr, iWTa, iRDr : std_logic;
      signal  WTr, WTa, RDr, RDa,
            WPr, WPa, WFr, WFa,
            RPr, RPa, RFr, RFa, F, E
            : std_logic_vector ((DEPTH - 1) downto 0);
      signal Fi, Ei, Eni, Ai
            : std_logic_vector ((DEPTH - 1) downto 0);
      subtype WORD_TYPE is std_logic_vector ((WIDTH - 1) downto 0);
      type WORD_ARRAY_TYPE is array ((DEPTH - 1) downto 0) of WORD_TYPE;
      signal Q : WORD_ARRAY_TYPE;

begin
      OUTSEL : process (RDr, Q) begin
            for I in 0 to (DEPTH - 1) loop
                  if RPa(I) = '1' then
                        Do <= Q(I);
                  end if;
            end loop;
            end process;

      WS_0 : WS_ELEMENT_0
                  generic map (WIDTH => WIDTH)
                  port map (
                        Di => Di,
                        Do => Q(0),
                        WTr => WTreq,
                        WTa => WTa(0),
                        RDr => RDr(0),
                        RDa => RDack,
                        WPr => WPr(0),
                        WPa => WPa(0),
                        WFr => WFr(0),
                        WFa => WFa(0),
                        RPr => RPr(0),
                        RPa => RPa(0),
                        RFr => RFr(0),
                        RFa => RFa(0),
                        FULL => F(0),
                        EMPTY => E(0),
                        RSTn => RSTn
                  );
```

```
WS_E : WS_ELEMENT_E
            generic map (WIDTH => WIDTH)
            port map (
                Di => Di,
                Do => Q(DEPTH - 1),
                WTr => WTreq,
                WTa => WTa(DEPTH - 1),
                RDr => RDr(DEPTH - 1),
                RDa => RDack,
                WPr => WPr(DEPTH - 1),
                WPa => WPa(DEPTH - 1),
                WFr => WFr(DEPTH - 1),
                WFa => WFa(DEPTH - 1),
                RPr => RPr(DEPTH - 1),
                RPa => RPa(DEPTH - 1),
                RFr => RFr(DEPTH - 1),
                RFa => RFa(DEPTH - 1),
                FULL => F(DEPTH - 1),
                EMPTY => E(DEPTH - 1),
                RSTn => RSTn);
WS_ARRAY :
    for I in 1 to (DEPTH - 2) generate
        WS : WS_ELEMENT
            generic map (WIDTH => WIDTH)
            port map (
                Di => Di,
                Do => Q(I),
                WTr => WTreq,
                WTa => WTa(I),
                RDr => RDr(I),
                RDa => RDack,
                WPr => WPr(I),
                WPa => WPa(I),
                WFr => WFr(I),
                WFa => WFa(I),
                RPr => RPr(I),
                RPa => RPa(I),
                RFr => RFr(I),
                RFa => RFa(I),
                FULL => F(I),
                EMPTY => E(I),
                RSTn => RSTn);
    end generate;
REQCON :
```



```
    process (WFr, RFr) begin
        for I in 0 to (DEPTH - 1) loop
            if (I = 0) then
                WPr(I) <= WFr(DEPTH - 1) ;
                RPr(I) <= RFr(DEPTH - 1) ;
            else
                WPr(I) <= WFr(I - 1);
                RPr(I) <= RFr(I - 1);
            end if;
        end loop;
    end process;

ACKCON :
    process (WPa, RPa, RSTn) begin
        for I in 0 to (DEPTH - 1) loop
            if (I = (DEPTH - 1)) then
                WFa(I) <= WPa(0);
                RFa(I) <= RPa(0);
            else
                WFa(I) <= WPa(I + 1);
                RFa(I) <= RPa(I + 1);
            end if;
        end loop;
    end process;

                --< GLOBAL >---------------------------
                -- WTack  <= DWTr or iWTa;
                RDreq  <= iRDr;

                FULL <= Fi(DEPTH - 1);
                Fi(0) <= F(0);

                EMPTY <= Ei(DEPTH - 1);
                Ei(0) <= E(0);

                --< Local >----------------------------
                WTack  <= iWTa;

                iWTa <= Ai(DEPTH - 1);
                iRDr <= Eni(DEPTH - 1);


                Ai(0) <= WTa(0);
                Eni(0) <= RDr(0);
```

```
        FEg :
          for I in 1 to (DEPTH - 1) generate
                --< GLOBAL >---------------------------
                Fi(I) <= Fi(I-1) and F(I);
                Ei(I) <= Ei(I-1) and E(I);
                --< Local >----------------------------
                Ai(I) <= Ai(I-1) or WTa(I);
                Eni(I) <= Eni(I-1) or RDr(I);
          end generate;


    WACK_Time : process (WTreq) begin  -- Speed up ack cycle
            if WTreq = '1' then
                DWTr <= WTreq after 1 ns;
            else
                DWTr <= WTreq after WTreq2ack;
            end if;
    end process;
    ------------------------------------------------
end WORD_SLICE_L;
```

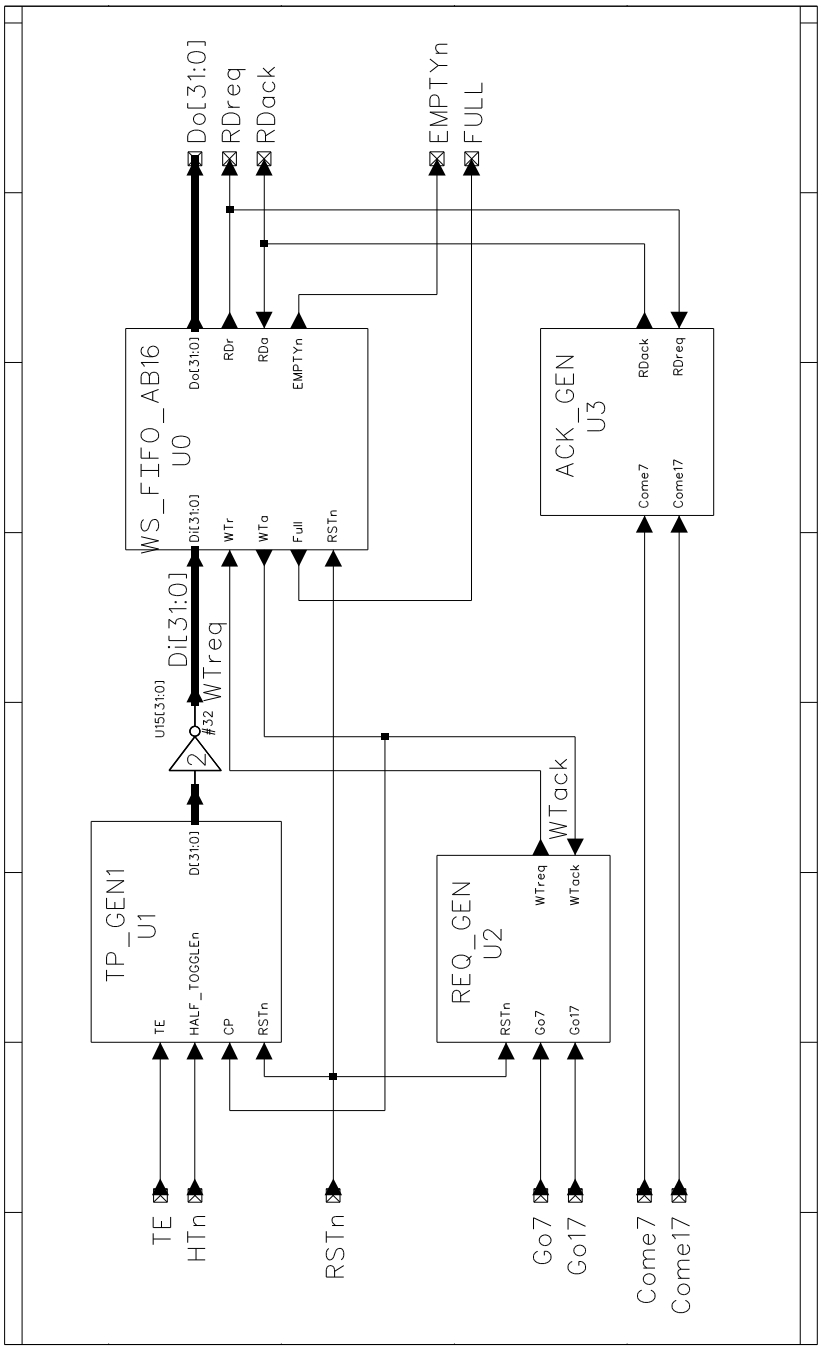


Figure A.1: Test environment of the word-slice FIFO

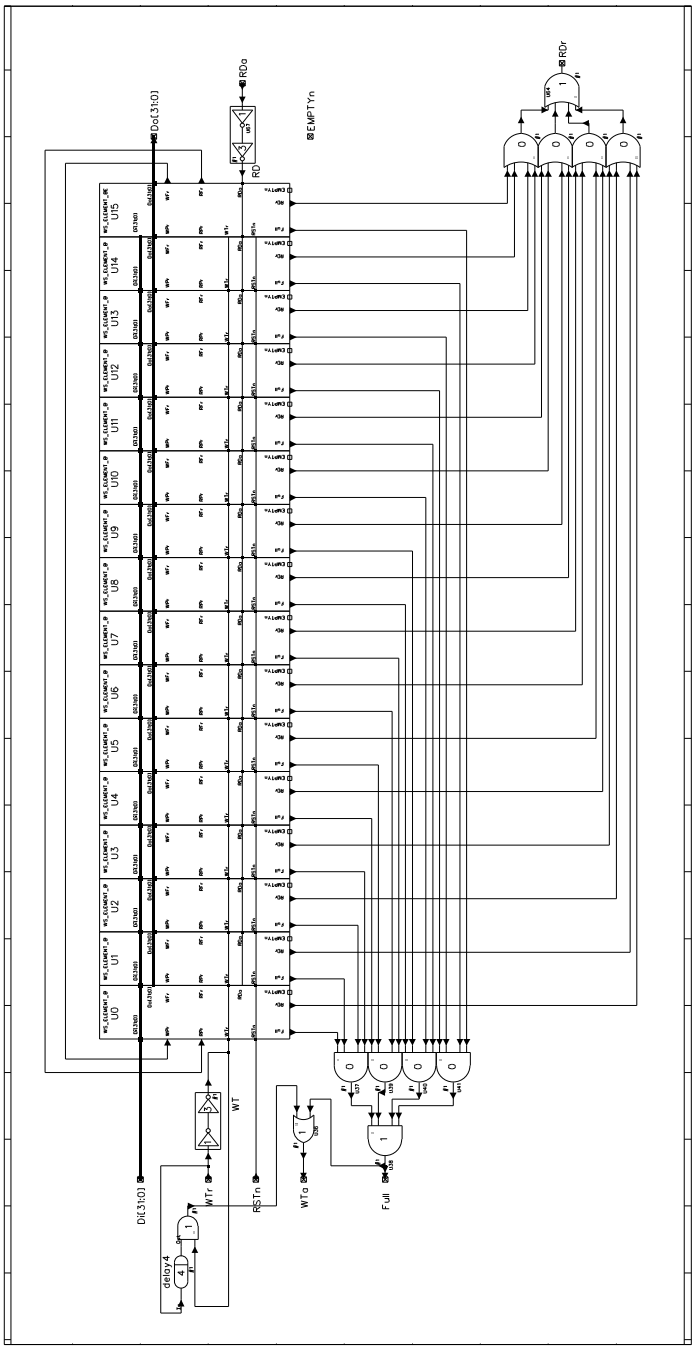


Figure A.2: The word-slice FIFO

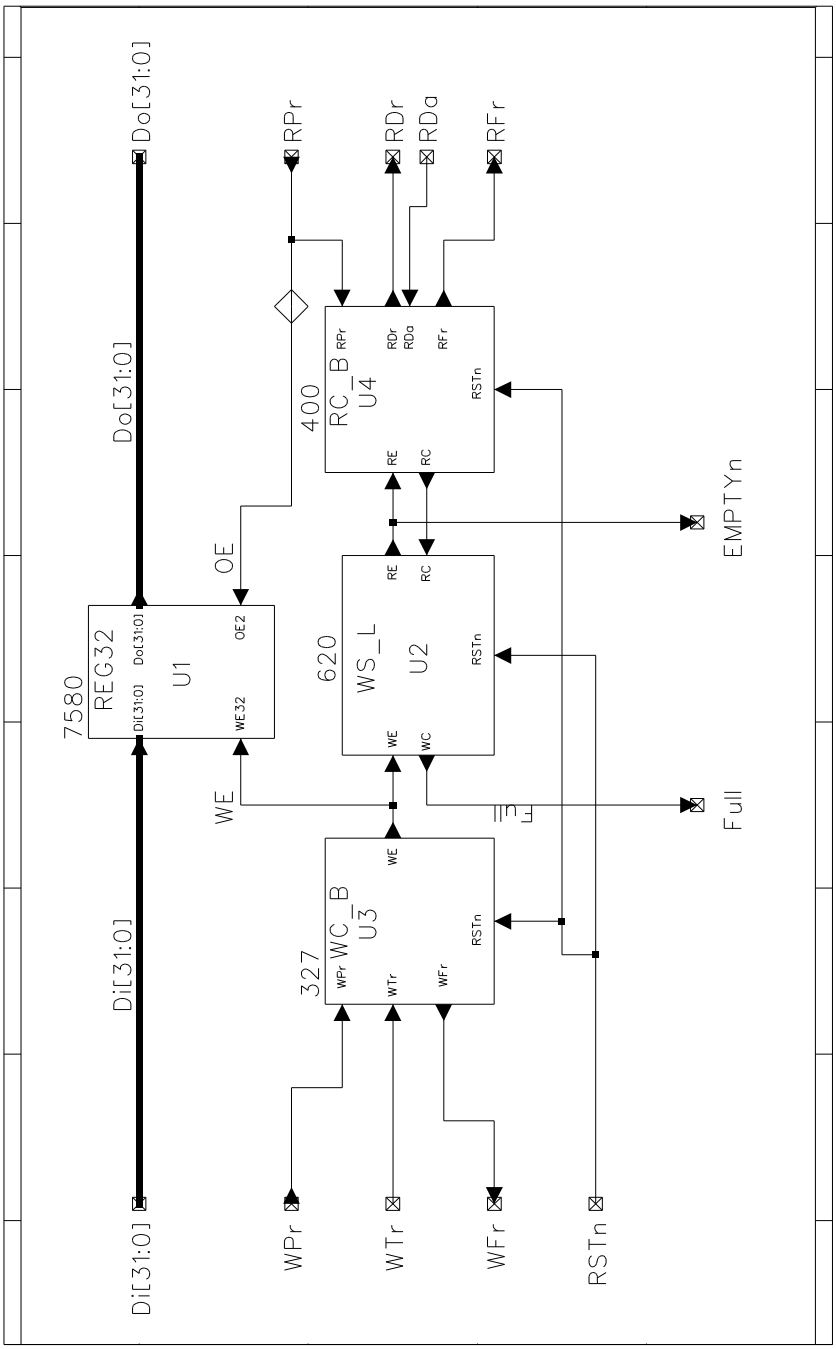


Figure A.3: The word-slice FIFO element

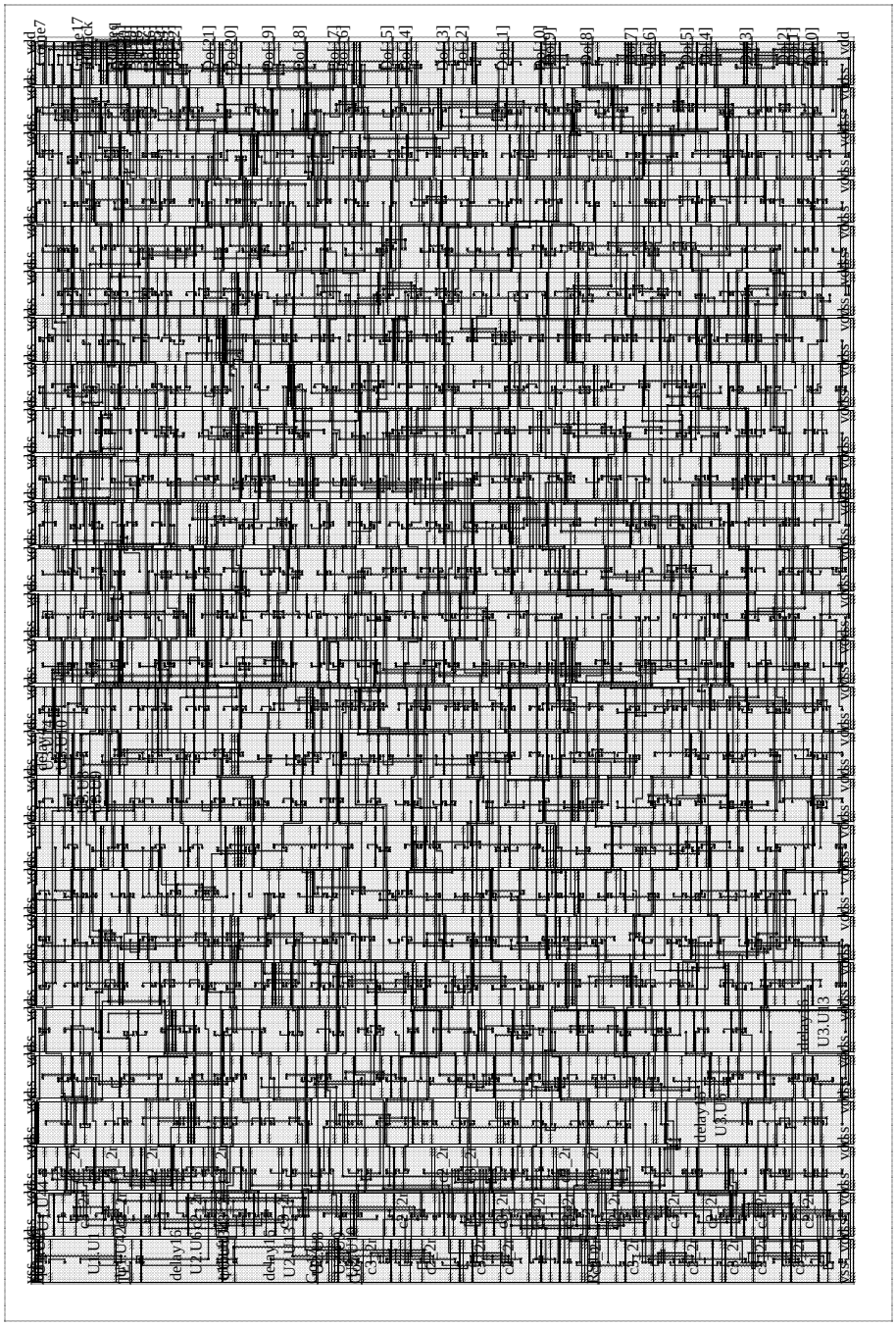


Figure A.4: Physical layout of the word-slice FIFO

Appendix B

Micropipeline FIFO

B.1 Test environment

B.2 Top level FIFO structure

B.3 Physical layout

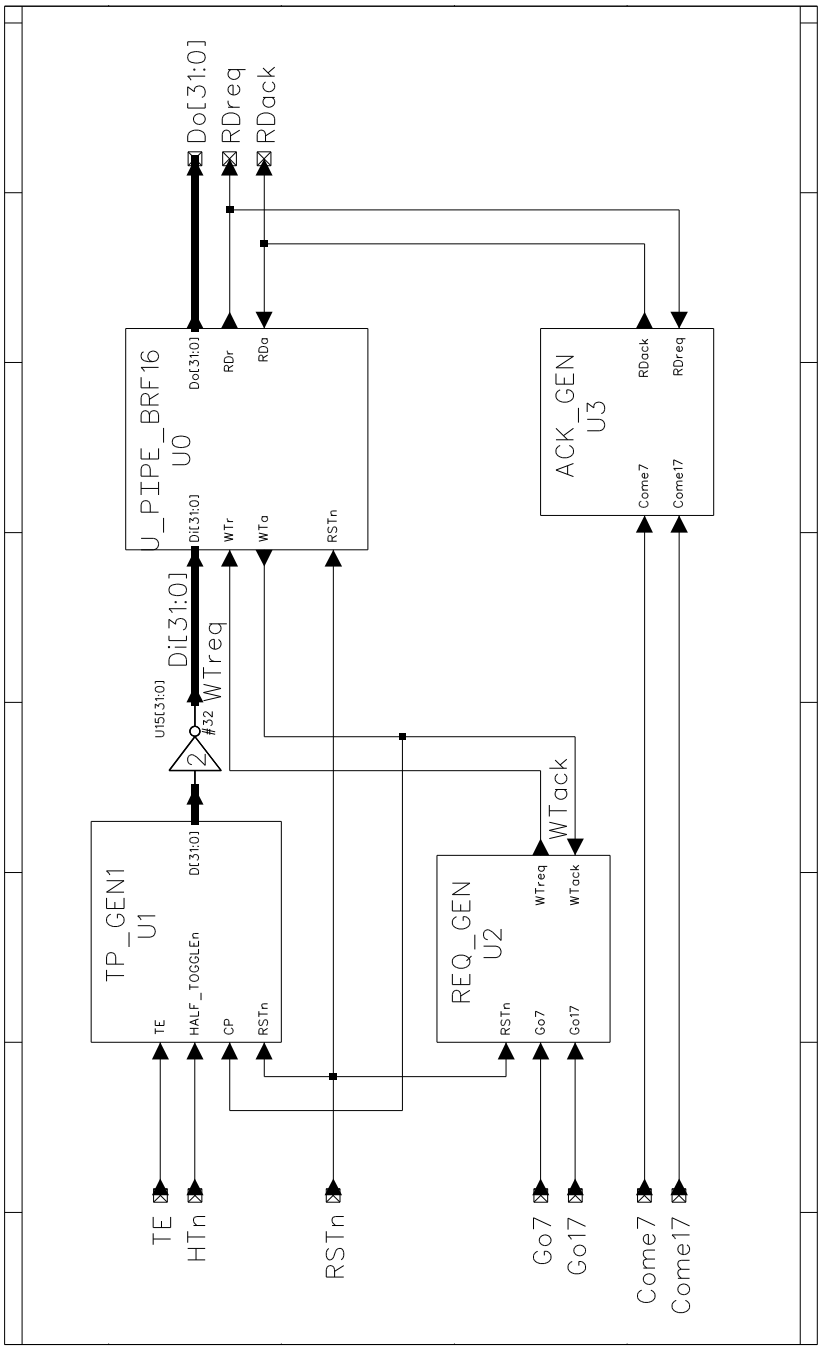


Figure B.1: Test environment of the micropipeline FIFO

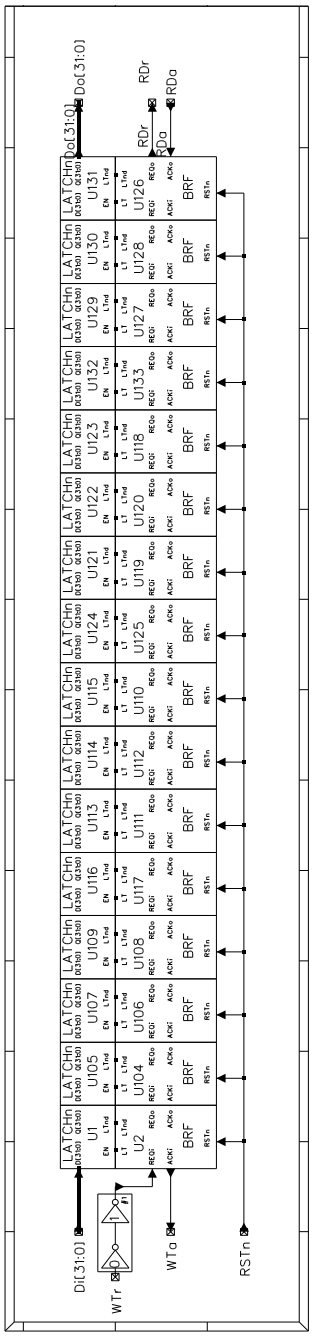


Figure B.2: The micropipeline FIFO

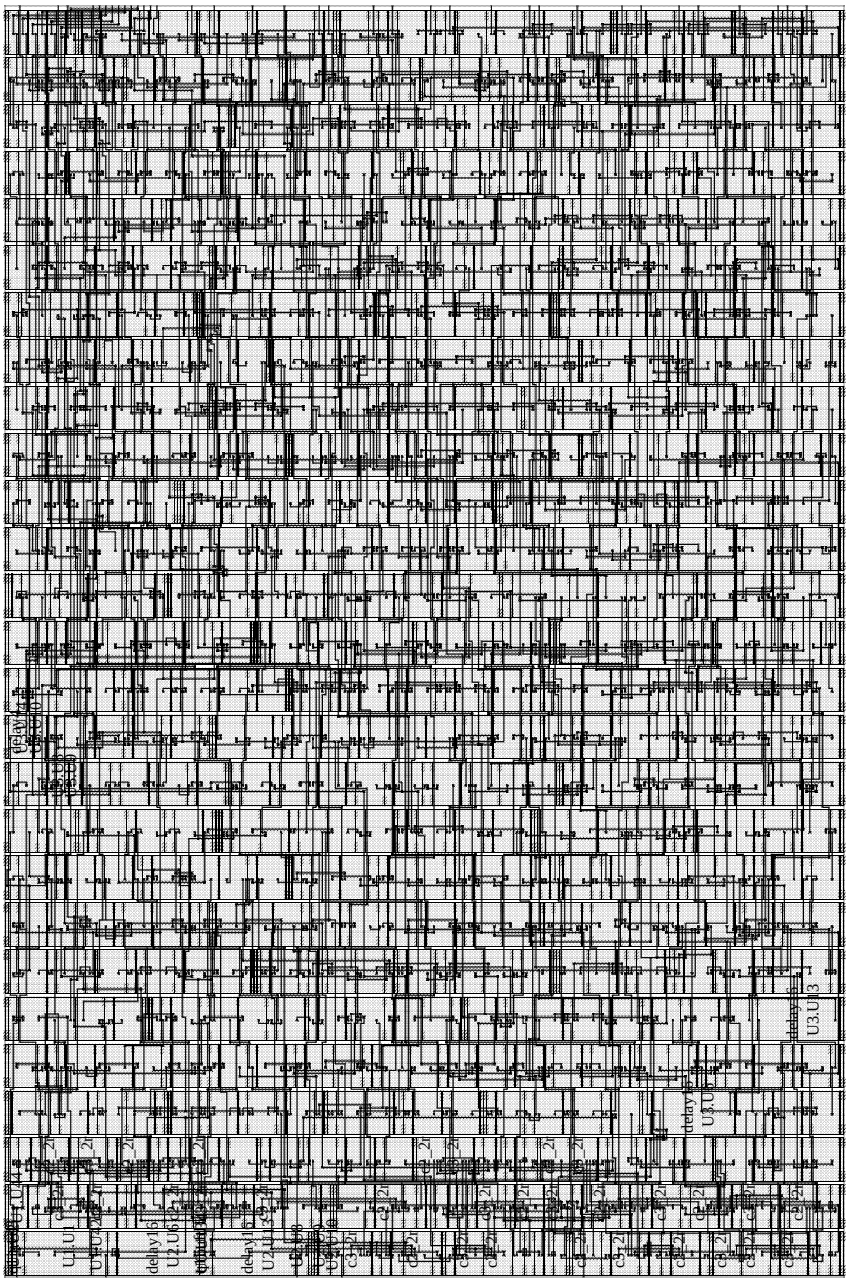


Figure B.3: Physical layout of the micropipeline FIFO

Appendix C

Evaluation

C.1 Measurement of current for a data transfer

The upper plot of i(GND) is the current wave of micropipeline FIFO with 16 words of 32-bit memory. The lower plot of i(GND) is the current wave of corresponding word-slice FIFO. In the micropipeline FIFO, we can see the consecutive peaks after a write operation. It shows that a series of memory and control elements are activated for a write operation. In contrast, there are only three peaks in word-slice FIFO: a middle peak is for the change of stored data, the other peaks are for driving the control signals.

Figure C.1: Measuring currents for a data transfer