A DUAL-CLOCK FIFO FOR THE RELIABLE TRANSFER OF HIGH-THROUGHPUT DATA BETWEEN UNRELATED CLOCK DOMAINS

By

RYAN WILLIAM APPERSON
B.S.E.E. (University of Washington) March 2002

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan Baas

Member, Dr. Rajeevan Amirtharajah

Member, Dr. Venkatesh Akella

Committee in charge
2004

Abstract

First-In First-Out (FIFO) memory structures are widely used to buffer the transfer of data between processing blocks. High performance and high complexity digital systems increasingly are required to transfer data between modules of differing and even unrelated clock frequencies. This thesis presents an encompassing description of the motivation and design decisions for a robust and scalable dual-clock FIFO architecture. It also investigates the hardware design issues involved in this architecture through the custom CMOS circuit design of the dual-clock FIFO architecture. The proposed design utilizes an efficient and low-latency memory array structure and can operate in applications where multiple clock cycles of latency exist between the data producer, FIFO, and the data consumer. This feature is increasingly relevant in high-speed designs where multiple clock cycles are not uncommonly needed to transmit data between major processing blocks. It also includes a configurable synchronization circuit that robustly synchronizes asynchronous signals within the FIFO.

Acknowledgments

This thesis is dedicated to all the people who have helped me along way to achieve this accomplishment. I would like to thank my friends and family for their guidance and support. I would especially like to thank my parents and Michelle for all their love and support through all the challenges of graduate school. Thank you to all the members of VCL, especially Mike Lai, Omar Sattari and Mike Meeuwsen for their encouragement, advice, support and all the laughs. I would also like to thank Dr. Amirtharajah and Dr. Akella for taking the time to be on my committee and all of their feedback and suggestions. Additionally, thank you to Intel for their support of VCL and providing the computers that performed much of the work for this thesis. Finally, I would like to thank my advisor, Dr. Baas, for his support, advice, optimism and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Synchronous systems have traditionally been the predominant method of implementation in digital electronics design. Nearly all present day systems utilize synchronous design techniques [1]. However, in order for this to be accomplished, a global clock reference must be accurately supplied to all areas of the circuit at almost precisely the same time. Transistor sizes have been continually shrinking, while both clock frequencies and design automation tool capabilities have been increasing. This is leading to larger and more complex, high-speed system implementations. Unfortunately, global interconnect scaling has not been able to maintain the same performance increases [2], causing distribution of a global clock signal to become a major concern in system design. This has resulted in clock distribution requiring more hardware and increased design time and effort.

One solution for coping with this problem is to utilize *self-timed* or *asynchronous* circuits, which lack a global timing reference. However, several obstacles prevent a major transition to this design style, including the lack of mature design tools for asynchronous design and the unwillingness of the industry to incur the cost and risk of moving away from a design style that has been so successful in the past [3]. An alternative approach is to create systems that mix asynchronous and synchronous design techniques. This design style is referred to as a Globally Asynchronous Locally Synchronous (GALS) design [4]. In this design paradigm local blocks are built using traditional synchronous design techniques, but these synchronous blocks do not share global timing information and are asynchronous with respect to each other.

Unfortunately, while it is often convenient to divide a system into multiple sub-components, it is unlikely that these components will be completely autonomous. Accordingly, data transfer is generally required between local synchronous blocks. Accomplishing this task reliably (*i.e.,* without

data corruption or loss) and efficiently (*i.e.,* minimizing area and energy dissipation) is one of the primary challenges in GALS designs.

One structure that is particularly well-suited for this task is the *First-In First-Out* (FIFO) memory structure. As indicated by its name, data items flow through the structure in the prescribed fashion, wherein the first data item that enters the structure will be the first data item to leave the structure. A basic FIFO architecture can be modified to accommodate two independent clock inputs. Data passing through the FIFO module will enter with reference to one clock and exit with reference to the other clock. In this way, data can be safely passed between independent clock domains.

1.1 Project Goals

There are three primary goals for this project. The first is to research synchronization strategies, asynchronous design and FIFO design, focusing on how this information can be applied to the development of an effective means of transporting data between two unrelated clock domains. Secondly, this knowledge will be used to design a module capable of reliably transporting high-throughput data between independently clocked processing units. This module will then be integrated into two different RTL level models of the working GALS multi-processor system. The first is a behavioral model and the second is a more complex, cycle-accurate structural model. The final goal is to design and implement the circuits and full custom layout for the FIFO module for fabrication in a deep sub-micron CMOS technology.

1.1.1 Target System

To help gain perspective on the design choices and requirements that motivated the design and implementation of the dual-clock FIFOs presented in this thesis, it is useful to give an overview of the target application. The FIFO was designed for implementation in the *AsAP (Asynchronous Array of Simple Processors)* system chip [5]. The AsAP system is a parallel, reconfigurable two-dimensional array of processors. Each processor has a 16-bit fixed point datapath, which is divided into a 9-stage pipeline. The instruction set architecture follows a RISC type design style. Each processor has its own small memory space that is divided into separate modules for data, instructions and configuration values. The simplicity of the architecture, along with the small local memories, facilitates several of the primary design goals including high-speed, low area, good energy-efficiency and ease of programmability. Because the architecture is targeted to digital signal processing tasks,

several special purpose modules are included, such as address generators, repeat logic and a multiply-accumulate unit.

Clock generation is done independently at the processor level. This is accomplished by using a programmable ring oscillator, which can be configured to operate over a range of frequencies. The clock frequency can be controlled with special configuration instructions. Accordingly, the clock frequency can change dynamically during run time. Additionally, if a processor becomes idle, the clock can also be stopped by the processor control hardware until it can again perform useful work. In general, the clock pauses based on either the availability of the data items required to execute the processor's current instruction or the downstream processor's ability to accept data. While the clock is stopped when the processor becomes idle, unlike some applications, the clock is not paused or modified for synchronization purposes (see Chap. 3-5 for details). Because the processor clocks are generated and controlled locally, each processor operates asynchronously to the other processors in the array.

The processor array connections are software configurable. Each processor in the array has two input ports and one output port. Each input port can be connected to the output port of a neighbor processor. There can be no more than two input sources for a single processor. However, one processor can source its output data to as many as four processors.

The dual-clock FIFO modules described in this thesis are the mechanism for communication between the independent processor cores and the off-chip I/O devices. Because each processor has its own independent and dynamic clock, the FIFO must both buffer and reliably synchronize the data being transmitted between processors. Ideally, this is a high-throughput operation with as little latency as possible. The FIFO also has to supply a wake-up signal to any neighboring processors that are halted waiting on the FIFO. As mentioned the local clocks will pause when the processor becomes idle. When the clock is paused, all activity in that processor is stopped and all register values are frozen. Accordingly, a combinational signal must be supplied to restart the processor clock so that it can wake-up and perform the task that it was waiting to complete. Details of this asynchronous wake-up signal are covered in section 5.8.

1.2 Overview

This thesis addresses the design of dual-clock FIFOs and the necessary background information for understanding the problems this design addresses. We are particularly interested in solutions that enable the transfer of data between modules from completely unrelated clock domains.

Chapter 2 introduces key structures and parameters for all styles of FIFO buffers by analyzing the single-clock case. Chapter 3 discusses metastability and synchronization background and issues that need to be considered when working with multiple clock domain systems. Chapter 4 describes extensions to single-clock FIFOs that enable operation in dual-clock settings and presents an architecture for the design of robust and efficient solutions for implementing a dual-clock FIFO. Chapter 5 describes the hardware implementation of the dual-clock FIFO. Chapter 6 summarizes the work presented and suggests areas for future work.

Chapter 2

Single-Clock FIFOs

First-In First-Out (FIFO) buffer structures find widespread use in many digital system applications [6]. In general, the use of FIFOs falls into one of two categories. The first is for data rate matching between modules producing and consuming data at different rates. Since data flow into and out of an interface must be equal over long periods of time, rate matching implies a producer and consumer with different patterns of data bursting over time. A second category, which is becoming increasingly important with high clock rate systems, includes applications that use FIFOs to transfer data between blocks in clock domains that are unsynchronized—meaning the phase and frequency of the two clocks are not matched. This mismatch may be intentional (*e.g.,* an SoC with multiple clock domains) or unintentional (*e.g.,* in a large clock distribution network which has significant jitter and skew). Examples of these can be seen in Figure 2.1.

A particularly useful example of a FIFO in the second category is a *dual-clock* or *mixed-clock FIFO.* These FIFOs are also sometimes called *asynchronous FIFOs,* however, since the term *asynchronous* implies a lack of a clock, the term is likely better reserved for circuits without clocks. Dual-clock FIFOs operate with two clocks, with varying levels of timing similarity. Dual-clock FIFOs are covered in Chapter 4.

To understand the fundamentals behind dual-clock FIFOs it is useful to consider the case of a single-clock synchronous FIFO, because many of the same concepts apply. This chapter covers these fundamental principles of FIFOs.

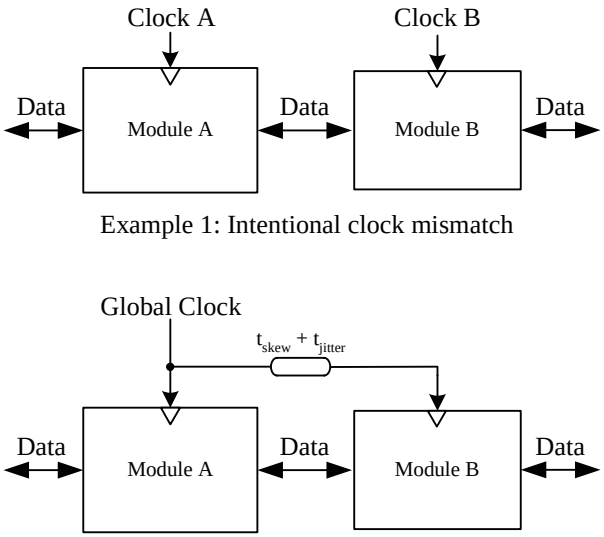

Figure 2.1: Target systems for dual-clock FIFOs

When comparing FIFO designs, a number of design parameters are important; they include:

- Robustness—data can not be lost, corrupted, or duplicated;
- High-throughput;
- Energy-efficiency;
- Scalability—the most widely-applicable designs will allow buffer sizes to scale over large ranges;
- Low-latency for latency-sensitive applications; and
- Support high clock rates for high performance.

2.1 Linear FIFOs

The simplest FIFO structure consists of a linear chain of latches or flip-flops connected serially as a shift register. Data is shifted into one end of the chain and propagates through every memory element until it reaches the end as shown in Figure 2.2. This FIFO is synchronous since all movement of data requires a clock edge.

Alternatively, an elastic FIFO can be constructed that uses control signal handshakes to propagate data from location to location. Unlike the synchronous case, a datum can propagate

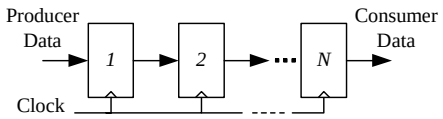

Figure 2.2: Linear shift-register FIFO block diagram

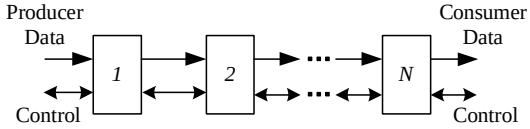

Figure 2.3: Linear elastic FIFO block diagram

through the FIFO without any new items entering. This results in the FIFO being at various degrees of fullness, hence, the name elastic. FIFOs of this nature fit nicely into asynchronous designs and many examples of these can be found in the literature [7] [8] [9]. An example of this type of FIFO can be seen in Figure 2.3.

Drawbacks of these approaches include high latency, low power efficiency and low memory density—which make scaling more difficult. High latency and power dissipation arise from the fact that each datum must flow through every element of the FIFO memory. Additionally, in the synchronous case every memory element requires a clock signal that impedes scalability and increases power consumption. Low memory density is caused by the high area per bit of latches and flip-flops.

Many extensions of this basic FIFO structure have been proposed with the key differences being the path by which data travels through the structure. These extensions provide worst case paths that are shorter than the total number of memory locations in the FIFO, resulting in lower latencies and improved energy efficiency. Examples of these variant structures are square FIFOs, parallel FIFOs, tree FIFOs and folded FIFOs [8].

2.2 Circular FIFOs

A more efficient method to construct a FIFO is to create a circular buffer using an array of memory elements designed so that data can be written to and read from arbitrary locations in the memory array. These structures are also called *parallel FIFOs* [10] and are often built using common SRAM or DRAM memories. The random access nature of memory arrays enables low minimum latencies, and high energy efficiencies compared to linear FIFOs. Scalability is also dramatically

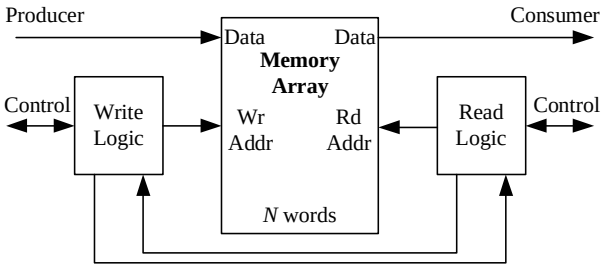

Figure 2.4: Circular FIFO block diagram

improved due to the fact that clock distribution is not directly affected by the FIFO size and the memory density is higher. In the event of a buffer size change, generally only very minor control logic changes are required. Figure 2.4 shows a high-level block diagram of a circular FIFO. Note that the read and write control blocks in single-clock FIFOs share a common clock so their separation represents a functional division, not necessarily a physical one. The write circuitry controls data being written to the FIFO and tests whether the memory is full. The read circuity controls the flow of data out of the FIFO and is concerned with determining when the memory becomes empty.

Control circuits for circular FIFOs are more complex than control circuits for linear FIFOs due to the need to manage non-regular write and read accesses with special cases when the array is full or empty. In particular, common failure modes for these FIFOs include the following cases:

- Underflow—invalid or duplicate data are transmitted by the FIFO,
- Overflow—valid data in the memory are overwritten,
- Deadlock—an interfacing condition that permanently prevents operation of the FIFO, and
- Stuck data—valid data remain in the FIFO, but are not read out even though read requests are made.

All parallel FIFOs must keep track of three mutually-exclusive states: 1) empty (also the initial state), 2) full, and 3) data occupancy between empty and full.

Tracking valid data within a FIFO is typically accomplished with one of two approaches. The first method involves using an N-bit register where each bit represents the validity of data in the memory and N is the number of words in the memory. A second method is to maintain read and write (or head and tail) address pointers which indicate the beginning and end of the valid data range in the memory. Figure 2.5 shows a scheme where the write address pointer (wr_ptr) indicates

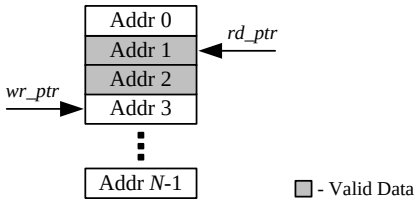

Figure 2.5: Typical write and read address pointer scheme for a circular FIFO

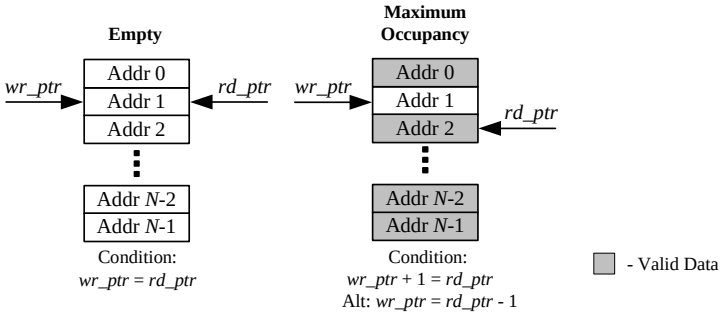

Figure 2.6: Circular FIFO with *empty* defined as when *wr_ptr* == *rd_ptr*

the memory location for the next data write, and the read address pointer (*rd_ptr*) indicates the location of the next memory read. Other schemes are possible, the most common of which offset their pointers by one memory location from the example shown here.

Given the use of read and write pointers, there are two primary methods to define the empty and full conditions. As shown in Figure 2.6, the first possibility is to define the empty condition as occurring when *wr_ptr* is equal to *rd_ptr*. The "maximum occupancy" (full – 1) condition is indicated when *wr_ptr* + 1 = *rd_ptr* or alternatively, when *wr_ptr* = *rd_ptr* – 1. The second case is shown in Figure 2.7 where the full condition is indicated by equality of *rd_ptr* and *wr_ptr* and the "minimum occupancy" (one datum) condition is indicated by *rd_ptr* = *wr_ptr* – 1.

Clearly, these schemes present problems in representing all possible states because the case where *rd_ptr* = *wr_ptr* becomes ambiguous without either keeping track of the pointer history or preventing the pointers from reaching this state in either the full or empty condition as mentioned above. This is because there are $N + 1$ levels of occupancy (0 to N words) but only $\log_2(N)$ bits in the address pointers—representing only N possible values. (There are actually many more total states, but given one address, say *rd_ptr*, there are $N+1$ possible states for *wr_ptr* and all such cases are equivalent.)

For the case shown in Figure 2.6, the N possible representations are used for memory

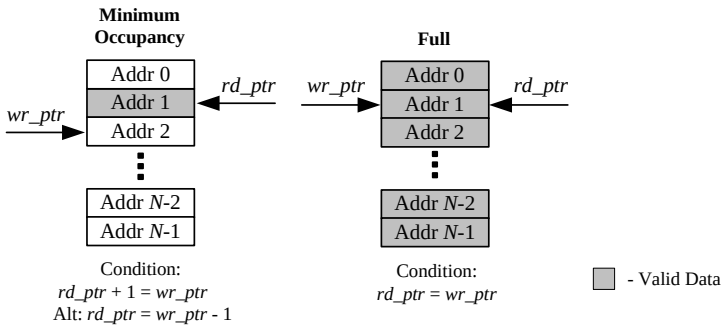

Figure 2.7: Circular FIFO with *full* defined as when *rd_ptr* == *wr_ptr*

occupancies of 0 to $N-1$ words. This results in a straightforward initial state (empty) and logic that must prevent the Nth memory location from being used.

For the case shown in Figure 2.7, the N possible values of the pointers are used for occupancies of 1 to N words. Since no empty state is possible, an appropriate initial state requires additional logic and the usefulness of this scheme is limited.

2.2.1 Methods for using all N words

Adding a state bit

There are two basic methods for using all $N+1$ memory occupancy states. The first involves adding a state bit to handle previously unrepresentable cases and keep track of the pointer history. For example, in the situation shown in Figure 2.6, the extra state bit could be used to indicate when the memory was full to distinguish the two cases when both address pointers were equal. Similarly, the scheme depicted in Figure 2.7 could use the additional state bit to indicate the empty condition.

Increasing the size of the address pointers

The second method is to increase the size of the address pointers by one bit. This additional bit increases the range of the pointers from modulo N to modulo $2N$. Read and write pointers that are equal modulo $2N$ unequivocally indicate an empty FIFO. On the other hand, read and write pointers that differ by N modulo $2N$ indicate a full FIFO since the *wr_ptr* must be N ahead of *rd_ptr*. This method will normally be simpler to implement than the method of adding a state bit. Empty detection is straightforward and full detection is accomplished by an equivalence test of the lower $\log_2(N)$ bits and an XOR of the address pointers' MSBs. For correct operation, the following

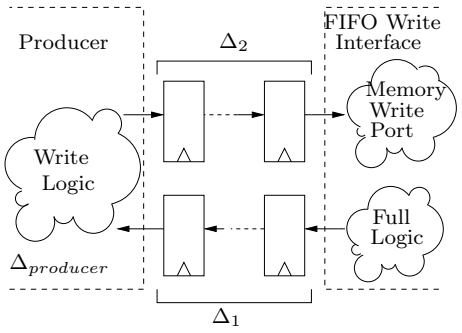

Figure 2.8: FIFO to producer write path showing multi-clock latency from *full* signal to halt in write data stream

inequalities must hold at all times.

$$rd\_ptr \leq wr\_ptr \leq rd\_ptr + N \tag{2.1}$$

2.2.2 Reserve space

Some applications require multi-cycle delays between a FIFO and the interfacing data producer or consumer. When multiple clock cycles separate the FIFO from the consumer, there is a possibility that data requests will be made to the FIFO that it will not be able to fulfill. This normally does not present a problem since a *valid* signal accompanying read data prevents the misinterpretation of unfulfilled read requests. However, when multiple clock cycles separate the FIFO from the data producer, the possibility of overflow exists if special precautions are not taken. Figure 2.8 illustrates this case showing the critical path from the *full* detection logic through the data producer's logic, and back to the FIFO's write port. The total delay in clock cycles is $\Delta_{total} = \Delta_1 + \Delta_{producer} + \Delta_2$. Prevention of overflow can be accomplished in one of two ways: either by the addition of a secondary FIFO of at least Δ_{total} words, or by the FIFO signaling the data producer to stop writing when its occupancy reaches $N - \Delta_{total} = Reserve$ words—which reduces the effective size of the memory under most conditions. Because of its simpler implementation, the second method is generally preferred.

2.2.3 Arithmetic hardware

For the case of address pointers of length $\log_2(N) + 1$ bits, and a non-zero *Reserve* space, we desire simple logic to determine when to signal the data producer to stop sending data (*wr_hold*).

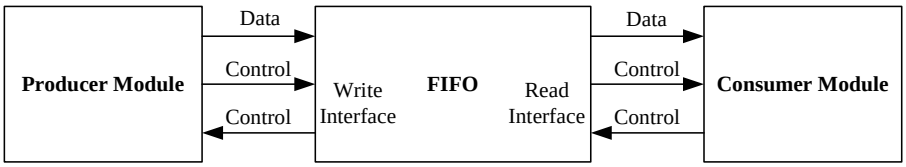

Figure 2.9: System architecture using a FIFO module for communication

Since $wr\_ptr - rd\_ptr$ is the occupancy of the memory, the signal threshold is then,

$$wr\_ptr - rd\_ptr \geq N - Reserve \tag{2.2}$$

$$wr\_ptr - rd\_ptr + Reserve \geq N \tag{2.3}$$

We prefer the form of Equation 2.3 because we can easily test whether a value is greater than or equal to N when using $\log_2(N) + 1$ bit words by checking the MSB of the sum—the inequality is true if the MSB = 1. The left hand side of Equation 2.3 is calculated with a three-input, $\log_2(N)+1$ bit adder. While converting values to signed 2's complement form will certainly work, it is not required and simple unsigned values will work properly for all cases since modulo $2N$ arithmetic will effectively map all negative values to their value plus $2N$.

Other required arithmetic logic for the read and write sides is roughly equivalent. Binary incrementers are needed for address generation and comparators are needed for empty (and full if reserve space is not required) detection. In some cases, a value indicating the occupancy of the memory is desired. Hardware to calculate this difference can be shared with the previously-mentioned adder.

2.2.4 Communication protocols

When considering the interface protocol, a *producer* module refers to a module which is transmits data and a *consumer* module refers to a module which is receives data. Accordingly, a FIFO module lies in between a producer and a consumer module, with the FIFO write circuitry interfacing to the producer module and the FIFO read circuitry interfacing to the consumer module. This is shown in Figure 2.9.

A *channel* refers to a communication connection between two modules, and the points of connection are called *ports* [11]. The FIFO shown in Figure 2.9 has two channels. The write channel refers to the channel connecting a producer to the input (write) port of the FIFO. The read channel refers to the channel connecting a consumer to the output (read) port of the FIFO. Each channel

has one control line going in each direction.

A communication channel should always have one *active port* and one *passive port* [11]. One concern in the interface design is deciding which port will be passive and which port will be active. For the FIFO, each channel is independent and the choice in port assignments is arbitrary, so long as each channel has one active and one passive port. For example, looking at the write channel there are two choices for the signalling convention. One choice is that the control line leaving the FIFO is active and the control line leaving the producer is passive. The terms *request* and *valid*, respectively, fit well with this convention. Since the FIFO is the active port, any time it wants data it will indicate this on the request line. It will then wait for the producer to send it data. The producer indicates this event with its *valid* line. In the second case, the FIFO write interface is the passive port and the producer is the active port. In this situation the producer will indicate that it wants to send data using the *valid* line. When the FIFO is ready to accept the data it will acknowledge receipt of this data with its control line.

The same situation exists on the read channel, with the only difference being that the FIFO read interface is producing data and the consumer is receiving data. The formation of the read port signalling convention can be done in a manner analogous to the write side case.

From the FIFO design standpoint, there is no inherent advantage to one signalling approach over another, so the target application will likely determine the choice. Table 2.1 summarizes all the possible states of one interface channel. The data valid and data request columns indicate the control line status in each of the cases. The consumer is assumed to have the active port in this case. The final two columns are mutually exclusive and represent the equivalent state of a FIFO if the interface state in the far left column were mapped onto the write channel or read channel, respectively. For example, the second row of Table 2.1 shows the case where the producer has no data and the consumer wants data. The write side of the FIFO is always a consumer, so if this state were mapped onto the write channel it means that the producer (the upstream module) has no data, but the FIFO write interface can accept data, so the FIFO is consequently not full. This is noted in the column showing the equivalent FIFO state if the interface were mapped onto the write channel. Alternatively, if this state were mapped onto the read interface of the FIFO, the FIFO state would be empty since the read side of the FIFO is always a producer and the producer has no data. As noted, these interfaces are not mapped onto the read and write channels at the same time, even though they are listed in the same row.

Having a reserve space increases the number of possible conditions within the write interface

Interface State	Data Valid	Data Request	FIFO state if interface is mapped to write channel	FIFO state if interface is mapped to read channel
1) Producer has no data, Consumer doesn't want data	No	No	Full	Empty
2) Producer has no data, Consumer wants data	No	Yes	Not full	Empty
3) Producer has data, Consumer doesn't want data	Yes	No	Full (not receiving)	Not Empty (not transferring)
4) Producer has data, Consumer wants data	Yes	Yes	Not Full (receiving)	Not Empty (transferring)

Table 2.1: Four primary states of a general data transferring interface

Interface State	Data Valid	Data Request	FIFO State
1) Producer has no data, FIFO doesn't want data	No	No	Not Receiving data (FIFO Full or in reserve)
2) Producer has no data, FIFO wants data	No	Yes	Not Receiving data (Not full and not in reserve)
3) Producer has data, FIFO doesn't want data	Yes	No	Receiving data into reserve space
4) Producer has data, FIFO wants data	Yes	Yes	Receiving data into non-reserve space

Table 2.2: States of a data transferring interface with equivalent FIFO state when reserve space is utilized

logic of the FIFO. Table 2.2 enumerates these changes. Since the reserve conditions generally only apply to the write side of the FIFO and the write side is always a consumer, the consumer if now call the FIFO in the leftmost column. To avoid overflow conditions, no data can be requested until there are at least *reserve* empty locations in the FIFO. Accordingly, the effective size of a FIFO with reserve space is reduced to the number of FIFO memory locations minus the number of reserve locations.

Chapter 3

Synchronization

One of the fundamental problems in systems lacking a single global timing reference is correctly ordering events. This process is generally referred to as synchronization. Some systems operate with no sense of global timing, and instead operate in direct response to signal transitions. This is referred to as asynchronous circuit design [11]. In these cases, synchronization is often unnecessary [12]. However, an alternative method for circuit design employs a global clock signal, which carries timing information for all signals within a specific area. A *clock domain* is specified as an area where all the signals utilize the same clock signal for their timing reference [12]. Circuits that operate in reference to a specific clock are referred to as synchronous circuits.

Systems designs of the past and present day are typically built to be synchronous. However, as mentioned in Chapter 1, maintaining this trend is becoming difficult and expensive, so future designs will likely need alternative approaches to solve the global timing issue. One option is to move to fully asynchronous design, but, as mentioned, a complete transition to this style is unlikely in the near future. To overcome these obstacles a more moderate design paradigm is to break the system up into multiple clock domains, and not maintain a specific timing relationship between clock domains. Systems of this type are also called *Globally-Asynchronous, Locally-Synchronous (GALS)*. Chapiro [4] is often cited as one of the first in-depth investigators of GALS architectures.

In general, the timing relationship between a signal and a clock can be cast into one of five categories [12] [1]: 1) Synchronous, where the signal matches the clock in both frequency and phase; 2) Mesochronous, where the signal is the same frequency as the clock, but has a constant phase difference; 3) Plesiochronous where the signal is at a frequency close, but non-identical to the clock frequency, which implies a varying phase difference; 4) Periodic, where the signal has an

unknown relationship to the clock, but is periodic in nature; and 5) Asynchronous, where the signal is completely unrelated to the clock and signal transitions are arbitrary.

3.1 Metastability

Metastability is a fundamental problem present when interfacing asynchronous blocks [12] [13]. Synchronous systems require signals on the inputs of registers to be stable around the active edge of the clock signal. We refer to this time period as the *critical window*, and its boundaries are set by the setup and hold time requirements of registers. If this requirement is not met, the value may not be accurately sampled by the system and the output behavior of the sampling element is unknown. This results in the output of the sampling element resolving to one of three states: 1) the input value before the transition, 2) the input value after the transition, or 3) an intermediate value known as the metastable point. This intermediate state exists in all sampling elements with bistable components [14]. This state is not stable due to the high gain of the feedback loop, but the response time of the flip-flop leaving this state is non-deterministic [15]. If a sampling circuit with no regenerative element is used and timing is violated, intermediate values that are sampled (*i.e.,* the input is sampled somewhere along the transitioning data edge) will be held internally and not resolve to a logical value until a new value is sampled.

Synchronization is used to avoid, or reduce the probability of, metastability. The strategy utilized depends on the timing relationship of the clock and data signal(s), as detailed in the five categories above. Many examples of this can be found in the literature [12] [16] [17]. In general, the more that is known about the relationship between two signals, the easier it is to synchronize those signals.

Metastable output values are generally in the logically undefined region and can result in error propagation through a synchronous system. Due to its random nature, these errors are infrequent, difficult to characterize and hard to detect [15]. Accordingly, special care must be taken to avoid metastability when sampling asynchronous inputs. It should also be noted that metastable conditions can also occur due to pulse width violations of the clock signals [15] [18].

3.1.1 Background

Some of the earliest discussions of system failures due to metastability were published in the mid 1960's [19]. Since then this topic has been extensively published making a comprehensive survey of metastability literature challenging. Accordingly, this is outside the scope of this thesis,

and instead a brief overview of some important literature in this area will be given. Much of the pioneering work to analyze metastability was done in the 1970's. Molnar and Chaney recorded and discussed anomalous flip-flop responses to logically undefined input conditions [20]. Pěchouček [15] attempted to address the issue of predicting and modeling a device's metastability characteristics and also discussed techniques for reducing the probability of synchronization failure. Hurtado and Elliot [14] addressed the metastability of bistable devices from a theoretical standpoint and showed that bistable devices must have a metastable state, and that such devices can be driven into this region and remain there for an unbounded amount of time. Marino [21] developed a generalized model for metastability in digital systems and demonstrated the unavoidability of metastable operation when using fully asynchronous inputs to bistable devices. Hohl *et al.* [22] looked into predicting the probabilities of synchronization failure based on simulation and theoretical analysis. Flannagan [23] investigated how to optimize CMOS circuits for synchronization performance. Horstmann *et al.* [18] researched synchronization issues in custom digital CMOS ASIC designs, including proposing ways to improve reliability of such cells. Portmann and Meng [24], considered the effects of buffering, technology scaling, and power supply on metastability characteristics. They also show simulation methods for approximating metastability device parameters. Jex and Dike [25] developed a high-performance, BiNMOS latch with good metastability resolution characteristics. They also give an overview of architecture techniques for reducing the probability of failure, and a straightforward simulation technique for determining a synchronizer's τ parameter. Dike and Burton [26] built upon these findings and also considered the effect of Miller capacitance and thermal noise on metastability characteristics. Ko and Balsara [27] simulated, measured and compared flip-flop metastability parameters of six flip-flop architectures using the method of Portmann and Meng [24]. Kinniment *et al.* [28], develop a more complex model for estimating MTBF and further investigate the effects of thermal noise on metastable behavior. Semiat and Ginosar [29] implemented several synchronizer architectures on a programmable logic device, took measurements to determine metastability characteristics and compared the various architectures.

Solutions to remove the metastability problem have been published, but many have limitations [30] and can result in circuit failures if not utilized properly [12] [13].

3.1.2 Basic failure model

In theory, a device can remain in an intermediate metastable state for an infinite amount of time. However, in practice the circuit resolves to a valid state after some non-deterministic period.

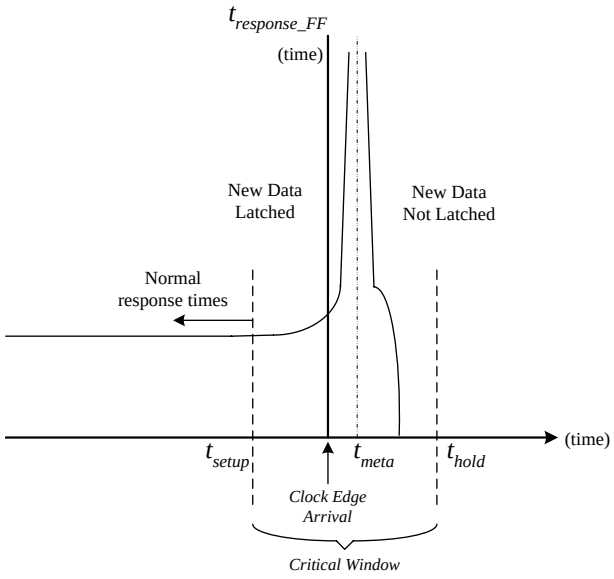

Figure 3.1: Generalized flip-flop response time versus data arrival time with reference to a clock edge arrival [11] [24]

Figure 3.1 shows the approximate relationship between input event transitions and the resulting flip-flop resolution time. A useful and prevalent approximation found in the literature for modeling the average failure rate due to metastability is shown in Equation 3.1 [11].

$$(Mean\ Time\ Between\ Failures)\ MTBF = \frac{e^{t_r/\tau}}{T_0\ f_c\ f_i} \tag{3.1}$$

The variables in Equation 3.1 are defined as follows: f_c is the clock frequency, f_i is the input data event frequency, and t_r is the allowed settling time before sampling. The remaining two parameters τ and T_0 are device dependent, and need to be experimentally determined [11]. The parameter τ is the exponential time constant of the metastability decay rate, and is sometimes called the metastability time constant [25]. The T_0 parameter is the asymptotic width of the time aperture in which the device can enter the metastable state, normalized to a response time of zero [25]. Since no real device has a zero response time, the T_0 parameter has no practical meaning, and is simply a mathematical characterization of the device's susceptibility to entering the metastable state.

3.1.3 Final remarks

Metastability is difficult to model and measure. This is primarily caused by the fact that true metastable events are probabilistic in nature, making them hard to quantize and capture. Because of this, metastability does not exist in RTL or logic simulations, and it is difficult to model in numerical circuit simulators because metastable events are normally extremely rare and extremely

small time steps are needed, requiring high numerical precision. In order to measure metastability parameters, special hardware and simulator test circuits must be constructed [26]. In addition, recent results suggest that device specific metastability parameters can vary over operating conditions [28] making it more elusive and difficult to predict. Even when hardware is available, metastability is difficult to test and measure because of its intermittent and rare occurrence in normally-designed circuits.

3.2 Synchronization Strategies for Asynchronous Inputs

In the case of a fully asynchronous input, no information is available regarding timing of signals, making synchronization the most difficult. Solutions to synchronize asynchronous signals fall into one of two categories: increasing time for resolution and pausing the clock [11].

3.2.1 Increasing metastability resolution time

The most prevalent model for the probability of a failure due to metastability in a bistable element is an exponentially decaying function [11].

$$P(t_s > t_r) = \frac{T_0}{T} \cdot e^{-\frac{t_r}{\tau}} \tag{3.2}$$

This model is shown in Equation 3.2, where t_r is the bounded time window allotted for resolving an intermediate output value and t_s is the actual time required for the signal to resolve. The two device parameters, t_0 and τ, are introduced in section 3.1.2. Accordingly, increasing the resolution window decreases the probability of synchronization failure exponentially. This equation can be extended to estimate the Mean Time Between Failures as given in Equation 3.1. Synchronization failure generally means that a synchronizer cannot produce a stable logical output by the time the system uses its output. The key result from both of these equations is that increasing the resolution time can greatly improve the ability of a synchronizer to avoid failures.

The most basic form of synchronizer in this category is the two flip-flop synchronizer [12]. Extensions of this idea are numerous, including pipeline synchronization [31], where more memory elements are placed in series to further increase the resolution window. Taking the most basic case of one synchronizing element, the MTBF formula can be calculated from Equation 3.1. The form of the equation remains the same, the only variable directly affected is the allowed resolution time (t_r). For the single element synchronizer, t_r can be determined by Equation 3.3, where T_{clk} is the clock period, t_{ff} is the propagation delay plus setup time of the sampling element, and t_{logic} is the

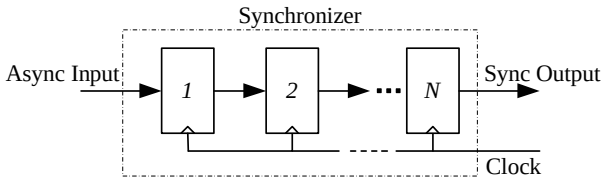

Figure 3.2: An N element chain synchronizer circuit

delay of any logic before the next memory device.

$$t_r = T_{clk} - t_{ff} - t_{logic} \tag{3.3}$$

This equation can be generalized to calculate t_r for a pipelined synchronizer of N elements as shown in in Figure 3.2. From Equation 3.3, a generalized equation for determining the resolution time for this setup can be developed. In this case, the memory elements will be assumed to be flip-flops since they are commonly used as the sampling elements in synchronizers. If N flip-flops are chained together, the synchronizer will now have roughly N clock periods to resolve possible metastable events. With the assumption that the flip-flops are directly tied together, t_{logic} will equal zero for all but the last memory element. Therefore, only one t_{logic} must be subtracted. The propagation delay and setup time for each of the N flip-flops must also be subtracted from the resolution time. The result is shown in Equation 3.4. In most systems, the output of the synchronizer is generally assumed to be available after only a t_{ff} delay, so the remainder of the cycle can be fully utilized for other logic. In this form, $t_{logic} + t_{ff}$ is equivalent to one clock cycle (T_{clk}), thereby reducing Equation 3.4 to the one shown in 3.5. The primary disadvantage to this setup is that $N-1$ additional cycles of latency are added even when no metastability occurs.

$$t_r(N) = N \cdot (T_{clk} - t_{ff}) - t_{logic} \tag{3.4}$$

$$t_r(N) = (N-1) \cdot (T_{clk} - t_{ff}) \tag{3.5}$$

There many possible methods for increasing the resolution time. An alternative to pipelining flip-flops in series is to uses a divided clock, which also reduces f_c and f_d. This method gives a better t_r for the same amount of delay, because only one t_{ff} is subtracted instead of $N \cdot t_{ff}$. However, it reduces throughput in addition to the increased latency. Another method, called a parallel synchronizer [25], uses alternately enabled parallel flip-flops. Each flip-flop is clocked at a slower rate, but on any given clock cycle only one of the flip-flops is supplying the output and only one is sampling the current input. The remaining flip-flops are resolving any metastability

that occurred at their respective sampling time. The output is multiplexed to choose the currently enabled output. This results in a larger t_r than pipelining without the reduced throughput of the divided clock method. However, it requires more hardware and is more complex because it requires properly timed enable signals.

The general trade-off for these schemes is increased area and/or latency in exchange for a lower mean time between failures. It should also be noted that these schemes do not eliminate the probability of a metastable event; they only reduce it.

Metastable immune synchronizers

In the above examples, a fixed window is given for the metastability to resolve before the data is sampled. An alternative approach is to isolate the metastability within the synchronization module and then allow as much time as necessary for the metastability to resolve [32]. This prevents indeterminate values from propagating past the synchronization module. This type of synchronizer does not provide the new stable output any sooner, nor does it provide a better defined binary value than a non-immune synchronizer [25]. In fact, due to the additional circuitry, these synchronizers have extra loading and delay, so it usually takes a longer period of time to resolve than a simple non-immune synchronizer [25]. An additional concern is the non-deterministic delay that is introduced into the system.

3.2.2 Pausing the clock

The second category of solutions uses *pausible clocks* clocks–also called *stoppable clocks* or *stretchable clocks*–to avoid metastability. In one variation of this technique, an arbiter circuit decides (usually with a mutually exclusive element) the precedence of a local clock edge or data event occurring in the *critical window*. The technique does not prevent metastability, but isolates metastability to the arbiter circuit. During the arbitration period, the sampling clock is paused until the metastability resolves. Pěchouček [15] and Chapiro [4] and were some of the earlier presenters of this technique, and many have since extended this idea. Yun and Dooply [33], developed pausible clock control circuits, which controlled asynchronous wrapper logic and used arbitration to avoid simultaneous data and clock events. Muttersbach *et al.* [34] built upon this idea, but aimed for a more scalable and portable design methodology by utilizing a different implementation. Moore *et al.* [35] developed a similar implementation, but used arbitration to avoid unnecessary pausing in order to increase communication bandwidth by reducing latency. These techniques use passive I/O

ports, which allow the asynchronous data module to control the flow of data. In these schemes I/O operations must be mutually exclusive to avoid conflicts.

An alternative is to use a direct communication scheme. This removes the need for arbitration in the clock oscillator, by only allowing the synchronous module to control communications. Bormann and Chueng [36] wrapped synchronous modules with asynchronous interface modules utilizing stretchable clocks with a direct communication scheme. Myer [37] used a similar scheme in interfacing asynchronous logic into a synchronous pipeline. Recently, Kessels *et al.* [38], proposed a clock synchronization style that opened up the ring oscillators and directly synchronized the clocks. The main advantage of direct communication is that it does not require arbitration in the clock oscillator. However, arbitration may still be required when there are multiple I/O ports. Arbitration free synchronization has the limitation that when the synchronous block requires data it must pause and remain paused until the request is granted. This can lead to non-deterministic delays in complex systems. Additionally, no computation can be performed during this period since the clock is paused, potentially reducing system performance.

While the exact implementations and clock pausing details vary, the main benefit of using pausible clocks over the previous schemes is that it can reduce the probability of synchronization failure to zero. However, these solutions also require that each module have a locally generated clock that can be locally controlled. Also, in cases where arbiters are used to resolve asynchronous conflicts, the system must be able to tolerate non-deterministic delays, since the time taken for arbitration is unpredictable and unbounded. In general, when the clock is paused the entire system is frozen and must wait for arbitration to complete before any work can be done. This becomes even more of a factor when interfacing to multiple asynchronous signals. Not only are more advanced arbitration schemes required, but each signal can cause a conflict. This results in increased arbitration time as the number of inputs grows. Care also must be taken to avoid system deadlock when modules are pausing their clocks. An additional concern is the difficulty in pausing the clock on the correct cycle in applications that have large synchronous blocks with complex clock buffering networks [39] [40].

3.3 Synchronizing Multi-Bit Words

There are potential problems sampling a multi-bit word using the synchronization methods that have fixed resolution time windows (sec. 3.2.1). This is true even without considering intermediate metastable output values. This results from real systems having varying amounts of delay per wire and the unpredictability of the logical result when timing constraints are not met on the

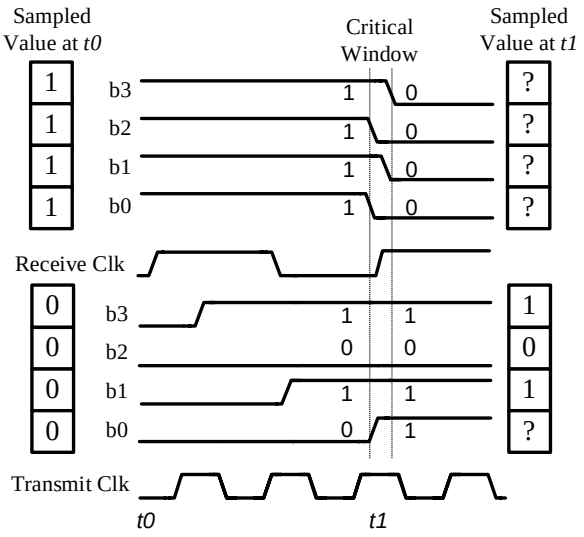

Figure 3.3: Sampling of a multi-bit transition word and a single-bit transition word

sampling device. The consequence is that it will be possible to sample different signals at different times. This case is illustrated in Figure 3.3 where two asynchronous words are sampled by a clocked module.

The top example shows the worst case where all bits transition within the critical window and every bit could independently take on the old value or the new value—after a sufficient metastability resolution period. In the bottom example, only one bit of the word transitions in the critical window and thus only one bit in the word has an unpredictable resolution value. Note that one resolution value (0) results in the previous word and the other resolution value (1) results in the new word—no erroneous words are possible!

It is important to further note that this property holds regardless of the relative frequencies of the read and write clocks. In the lower example, the "old" value of the receive register is 0000 and while the values change more than once before being read a second time, only one bit changes during the critical window and regardless of how the unstable bit resolves, a valid word will always be read. Details of how this property is integrated within a dual-clock FIFO design will be covered in Chapter 4.

It should be noted that when the timing relationship between the two communicating units is not asynchronous, alternative techniques can be employed to synchronize multi-bit vectors [12]. Oftentimes, such as in the mesochronous case, these techniques can result in the probability of metastability going to zero.

Chapter 4

Dual-Clock FIFO Architecture

Interfaces that require rate matching between their read and write sides and are clocked by unrelated clocks, require the use of a dual-clock FIFO. In some cases, the burst patterns of both producer and consumer are well characterized and bounded and the special techniques described in this section are not required, but this is not the general case. The focus of this section is on a flexible dual-clock FIFO architecture which supports data transfer across two clock domains with completely arbitrary phase and frequency relationships. Additionally, both clocks can change dynamically including dynamic frequency scaling and indefinite pausing. More specific implementation details for the physical hardware design of a dual-clock FIFO are discussed in Chapter 5.

4.1 Background

FIFO architectures find frequent use in situations where data must be passed between unsynchronized clock domains. Dally and Poulton [12] and Balch [6] present high-level views of the structure, but details of dual-clock FIFO design are lacking in the literature. Siezovic [31] presents a linear FIFO architecture for data synchronization known as pipeline synchronization. Because the architecture is linear, the limitations presented in section 2.1 regarding linear FIFOs apply.

Fully asynchronous FIFOs are common in the literature, but these designs do not utilize clocks, and therefore are difficult to apply in the case of synchronizing data between clock domains. Examples include fully asynchronous linear [41] and square [42] FIFOs, and investigations into improving the asynchronous control [9] of these FIFOs. An alternative FIFO architecture for use in limited dual-clock applications is presented by Chelcea and Nowick [43]. While the architecture robustly transfers data across unequal clock domains, the rate difference must be known *a priori*

to guarantee proper operation. Accordingly, dynamic frequency scaling is not supported. The architecture also presents scalability challenges due to the high fan-out of internal control signals. Additionally, it should be noted that the two flip-flop control synchronizer in the design does not guarantee complete avoidance of failure, especially at high-frequencies. A final reference that is a more informal description of an approach to designing a FIFO similar to the one presented here is given by Cummings [44]. This paper outlines one approach to dual-clock FIFO design with an emphasis on writing synthesizable HDL for this type of FIFO. Many background details on FIFOs and synchronization are not covered and the concepts of reserve space, pipelining and other techniques required for integration into real hardware are not covered.

4.2 Architecture Details

A block diagram of the proposed circular, dual-clock FIFO implementation is shown in Figure 4.1. The given dual-clock FIFO synchronizes data through its memory core and write and read circuitry operate in different clock domains. The synchronization task then involves passing control information between domains. This means that no data manipulation is required and bits within each data word are guaranteed to be uniformly synchronized. The primary challenge in designing a dual-clock FIFO compared to a single-clock FIFO is deciding what information to transmit between clock domains and how to synchronize that data. Otherwise, the architecture is very similar to its single-clock counter-part. The fact that each side of the FIFO is its own synchronously designed module makes the dual-clock architecture attractive because its logical design and verification is more straightforward than one involving fully asynchronous circuits.

4.2.1 Address and control information

Clearly, the key issue with a dual-clock FIFO is determining what control information to pass and how to pass it between unrelated clock domains. The method chosen here is to pass read and write address pointers. The pointers are increased to $\log_2(N) + 1$ bits to allow straightforward use of all N memory words. This still does not solve the problem of safely transferring those vectors across clock domains. As discussed in section 3.2, there are two general categories of synchronization methods, either pausing the clock or increasing the metastability resolution time. Many applications of interest do not allow local clock pausing, so we exclude pausible clock solutions as a solution in this case. This does not present a problem as we can reduce the probability of metastability arbitrarily low by extending the time resolution window as discussed in section 3.2.1. However, as noted in

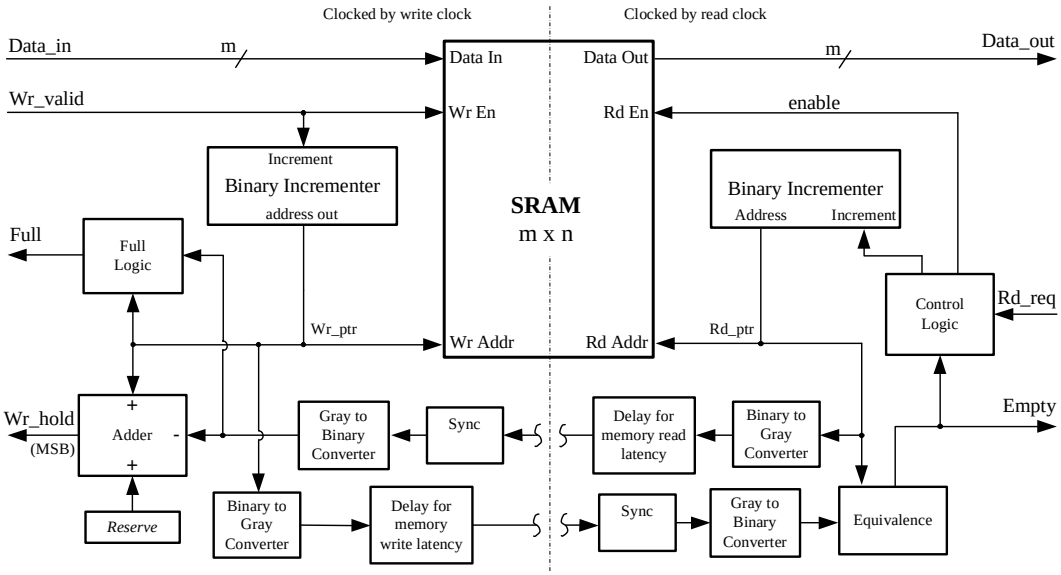

Figure 4.1: Detailed diagram of dual-clock FIFO architecture

section 3.3, multi-bit data must allow at most one bit transition between data words for reliable transmission. It is most convenient and natural to perform arithmetic manipulations on binary data. Unfortunately, during normal operation of the FIFO, the address pointers will frequently increment, causing a binary encoded value to have an arbitrary—often greater than one—number of bit transitions. For this reason, addresses are transformed to Gray code *(details in sec. 5.4)* before being passed across the clock boundary to prevent multi-bit transition failures. The addresses are converted back to binary for further processing once they are synchronized into the other domain.

As discussed in section 3.3, even though Gray coded values are used and the probability of synchronizaton failures (*i.e.,* the propagation of logically undefined values past the synchronizer module) are made extremely low, the absolute value of the sampled signal still cannot be predicted with certainty. As shown in Figure 3.3, one of the bit values is still potentially unknown. However, it is certain that the the vector will either represent the old value or the new value. The dual-clock FIFO architecture naturally solves any conflict that this may normally present. In this architecture, the control vector represents the location of the address pointer and is used by the other side of the FIFO. If the sampled address resolves to the new value there is obviously no problem. The case where the old value is retained also does not present a problem because it will be interpreted by the side receiving the address as the case where the pointer has remained in its old location and no actions (*i.e.,* reads or writes) have occurred. While this potentially adds latency to the system, it will not cause any data to be overwritten or incorrectly read. Eventually—likely on the next

sampling edge—an updated value will be received. Accordingly, barring a synchronization failure this method of passing data between domains is extremely robust.

Reducing the probability of synchronization failure to an acceptable level needs to be addressed within the synchronizer. The implementation of the address synchronization circuit is flexible and the exact design depends on the requirements for a specific application. For the case of clocks with arbitrary phase and frequency, a robust synchronization technique is required. The simplest sufficient synchronizer uses multiple series registers. If reasonably metastable-resistant registers are used, the probability of failure can be made extremely small with 2 or 3 series registers [11]. For increased flexibility, this architecture allows several parameters to be made configurable, including the choice of synchronization circuit—which determines the metastability resolution time. This allows the latency to be modified independent of the frequency at which each clock domain is being operated, which in turn enables an optimal balance between latency and synchronization failure without redesigning the circuit. Further details of the synchronizer used here are discussed in section 5.3.

An additional concern when sending address data to the read side is any latency associated with the memory core. This prevents the read side from reading data out of a memory location before it has been written. Likewise it prevents the write side from writing data before it has been read. Any necessary delay can be inserted by adding registers just before the data passes into the synchronizer circuit.

4.2.2 Reserve space

Reserve space is also utilized in this design to account for the transmit delay in both directions and the write logic of the data producer. A difference module is used to determine the current space left in the FIFO by determining the distance between the two pointers. This difference is then compared with the *reserve* value. If the difference is less than or equal to the reserve amount, it is clear that the FIFO must signal the producer to stop sending data. Any data left in the pipeline between the producer and the FIFO is safely written to the reserve space preventing overflow. As discussed in section 2.2.2, including reserve space reduces the effective size of the FIFO. The reserve space may not actually contain valid data even if the FIFO is not requesting data. Additional logic can be added to intelligently request small bursts of data to utilize some of the reserve space when it is unused. However, this increases the complexity of the FIFO in terms of design time and area and also increases the potential for an overflow condition if not designed correctly.

4.2.3 Communication interface

Details on the interface design choices for circular FIFOs and background on two-line signalling are presented in section 2.2.4. As discussed above, the dual-clock FIFO presented here utilizes reserve space on the write side of the FIFO. To increase producer efficiency, the write channel should adhere to the convention where the FIFO write interface control signal is the active port. This is a consequence of the latency that exists between the producer and the consumer. In general, an active port waits for a response from the passive port to determine its next step. In this case, the upstream producer is multiple clock cycles away from the FIFO write logic. The write logic can directly determine when the FIFO cannot accept any more data. If the FIFO's port is active, it can immediately transmit this information on its control signal (*wr_hold*) and the latency it takes the producer to receive this indication—and any data already in transit to the FIFO—is stored in the reserve space. If the producer were chosen to be the active port, every time it wanted to write to the FIFO it would have to stall. The stall length would be equal to *"reserve"* cycles, because a write would require the write control signal (*wr_valid*) to be transmitted to the FIFO, the FIFO to process the write attempt and then send back a reply. As discussed in section 2.2.2, each of these can potentially take several cycles. In some applications, this stall delay may be acceptable, but in general, processing modules want to be as active as possible. Accordingly, in this design some FIFO space is forfeited for higher activity in the producer.

On the read side, choosing one active port over the other has no clear advantage. Therefore, it will be determined by the target application. In one scenario, the FIFO read control output signal (*empty*) is the active port. In this case, it will always indicate if the FIFO is empty and the consumer should not request data on its control line (*rd_request*) until the FIFO has indicated that it is not empty. Alternatively, if the consumer is the active port, it can request data at anytime and the FIFO will respond by signalling whether the request was granted using its control line (*empty*).

Chapter 5

Hardware Implementation

This chapter covers the circuit level design and layout of a dual-clock FIFO module using the architecture introduced in Chapter 4. The primary hardware units that are required for the design are a memory array, a synchronization module, a binary incrementer, Gray/binary converters, an adder for determining the "In Reserve" condition, a comparator, and interface/control logic. These modules are elaborated upon and discussed below. The target application (see Sec. 1.1.1) drove most of the final implementation choices detailed in this chapter.

5.1 Design and Verification Process

The design process for the FIFO consisted of first creating a behavioral model in Verilog that was integrated into a simplified, single-cycle model of the AsAP system. NC-Verilog [45] was used for logical verification of the FIFO and the target system. After logical verification, a cycle accurate, pipelined version of the FIFO was created. This was integrated into the second version of the AsAP system, which was also pipelined and cycle accurate. Once the logical operation of the pipelined FIFO was verified and tested within the system, the final hardware design was performed. Structurally accurate models of the hardware modules were created in Verilog and tested to ensure proper logical operation. At this point, circuit design began with a 0.18 μm standard CMOS process as the target technology. Accordingly, all area estimates presented in this chapter assume a 0.18 μm process. The circuits were then layed out in MAGIC [46] and extracted for testing, with a parasitic capacitance threshold of 0.1 fF. HSPICE [47] simulations were run on the extracted layout to verify proper circuit operation over supply and temperature and also estimate performance numbers. IRSIM [48] was also used on the extracted layout to verify proper logical operation of the

modules.

The standard test conditions for reported performance numbers in this Section are, unless otherwise specified, a 1.8 Volt supply, typical NMOS and PMOS devices and an operating temperature of 40°C. To verify low-voltage operation, the circuits were tested with a supply voltage of 1 V. Output loading for testing individual modules was set to a default of four minimum sized inverter input capacitances. This was chosen because this represents a *fan-out of four (FO4)* [2] load for a minimum sized driver. We sized the PMOS to achieve a balanced rise and fall time for our minimum inverters, which results in an NMOS width of 5 λ and a PMOS width of 13 λ. The default load capacitance is then approximately 15 fF for a 0.18 μm technology. In some cases, such as in the memory, more detailed capacitive loading estimates were made to more accurately predict performance numbers and verify proper timing operation.

As mentioned, the overall design goal is to be able to operate at a high-speed. The rough speed target for the system under typical conditions is in the high hundreds of megahertz. The baseline goal is to guarantee operation with a clock period of less than 2 ns (frequency $>$ 500 Mhz). However, ideally the module will operate at higher speeds, so options to increase performance were generally taken, so long as the design time impact was not too large. The total flip-flop time—clock to output time plus setup time—for the D Flip-flop used in this design (Fig. 5.10) is approximately 250 ps under typical conditions and a minimum F04 load. This leaves the remaining time for all other logic in one pipeline stage.

5.2 Memory Design

As discussed in Section 2.2 the FIFO requires a memory core to buffer the elements as they pass through the module. From the architecture standpoint this is a drop-in module, so any type of memory array can be utilized. For this design a 16-bit by 32 entry memory array is required. To balance density complexity and robustness, a *Static Random Access Memory (SRAM)* type memory was chosen. SRAM architectures and design techniques are extensive, including optimizations for area, power and speed [49]. For this design, the key requirement was to make the memory robust and, secondly, try to balance the remaining parameters. It should also be noted that the SRAM presented here was designed such that the sub-cells and general architecture could be re-used for other memories within AsAP processors. Accordingly, not all the design choices were made entirely with the FIFO SRAM specifics in mind.

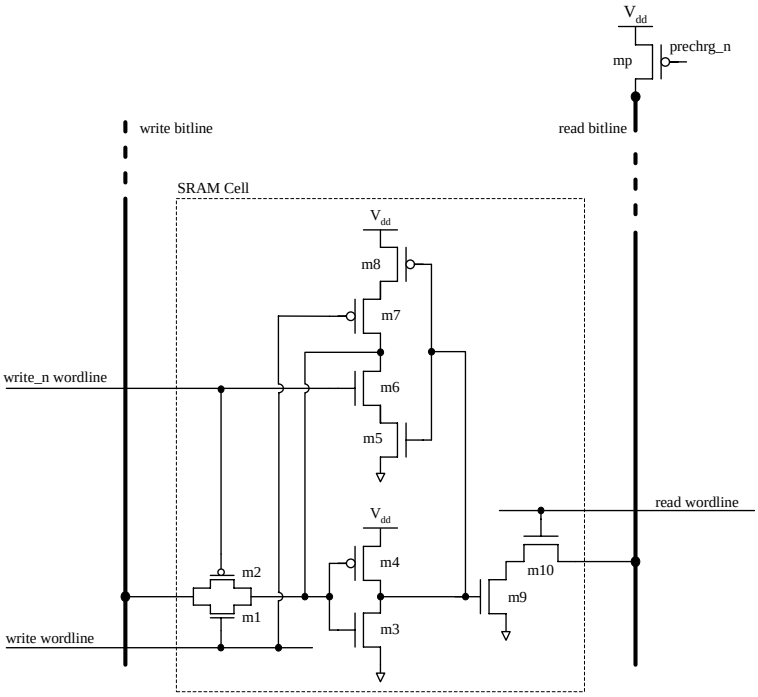

Figure 5.1: Basic ten transistor SRAM cell

5.2.1 SRAM cell

This SRAM core utilizes a ten transistor SRAM cell. A transistor level schematic can be seen in Figure 5.1. Sizing for the transistors is shown in Table 5.1. The full transmission gate (m1 and m2) controls writes into the cell and increases the speed and robustness of the circuit. Additionally, using a tri-state inverter (m5–m8) for the feedback in the cell makes writes faster and more reliable. Device m9 is included to reduce the capacitive coupling of the read control line into one of the primary storage nodes (drains of m3 and m4) and also to isolate that node to prevent current feeding back into the cell. Both of these events can potentially cause the cell to flip its value during a read. It was determined from circuit simulations that the amount of voltage drop occurring in the cell without the isolation device was too high, so the extra device (m9) was added. Overall, these modifications increase the cell's robustness, which is a top design goal. The trade-off is a larger cell size.

In the first pass of the memory design, the read bitline was static and an inverter within the cell drove it in both high and low through a full transmission gate. After simulating the critical path of this architecture, it was determined that without making the PMOS devices very large they were too slow to statically pull up the bitline during a read. To alleviate the problem, the read

Device	Transistor widths (λ)
m1	5
m2	5
m3	6
m4	12
m5	5
m6	5
m7	5
m8	5
m9	20
m10	13
mp	24

Table 5.1: Device sizing for SRAM cell of Figure 5.1 ($\lambda = 0.09$ μm for a 0.18 μm process)

bitline was made dynamic, so it is pulled-up by device mp during the pre-charge phase. Details of the pre-charge signal generation are shown later. This allows the removal of two large PMOS devices from the cell reducing cell area. It also removed 32 large P-diffusions from the each read bitline and replaced them with only one from the pull-up device (mp). This cut the total diffusion capacitance on the bitline by roughly one half. Assuming a fifty percent bitline toggle rate, the power consumption would not be negatively impacted by using a pre-charged bitline, and should even improve since the overall bitline capacitance was reduced. Additionally, the change achieves the desired result of increasing the overall read access speed of the SRAM.

Figure 5.2 shows the layout for the SRAM cell and also how the cell fits next to its neighbor cells when the cells are tiled out to form the final memory. Each SRAM cell occupies 32.6 μm^2 (5.04 μm tall by 6.48 μm wide).

5.2.2 Architecture

Write circuitry

Once the SRAM cell is designed, the control logic for generating the write wordline, read wordline and pre-charge signals needs to be designed. This logic is shown in Figure 5.3 For the write logic, the first stage consists of a set of predecoders to better distribute loading within the logic network and thereby speed up control signal generation. Even with the logic network optimized and broken up into stages with high current drive capabilities, it was still difficult to generate a transition on the actual wordlines within a small propagation delay of the inputs changing. To increase the overall robustness, the clock signal was used to gate the two fastest predecode signals. This causes the wordlines to unassert more closely in time—in terms of gate delays—to the active clock edge.

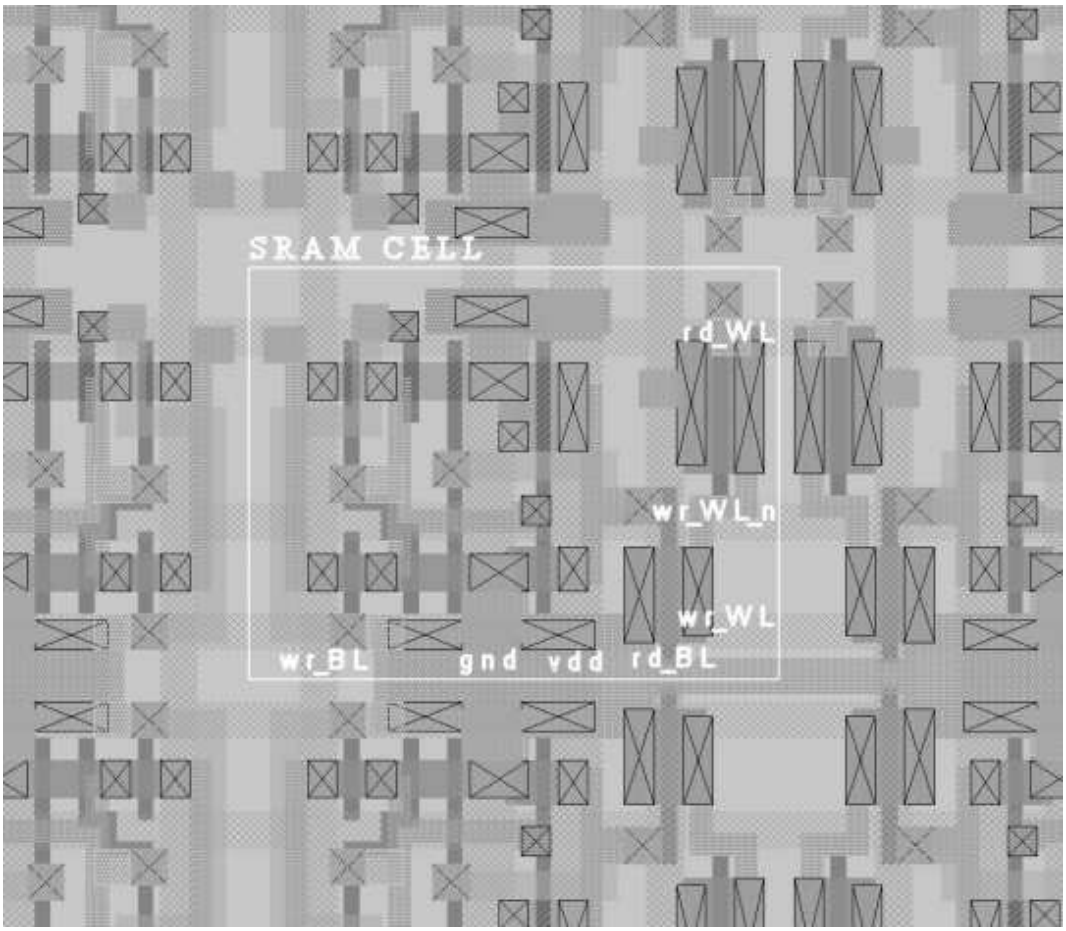

Figure 5.2: SRAM cell layout (BL = bitline , WL = wordline)

This keeps events ordered properly and helps prevent glitching on the wordlines. Event ordering is an issue, because the write bitline can toggle soon after a clock edge, whereas it takes longer for the last write wordline to unassert and the next one to assert. If the bitlines begin to toggle too early, the new value can overwrite the value that was just written to the previously active row of SRAM cells. To further assist with ordering events, the write data is re-registered by a positive edge-triggered flip-flop that is placed after the standard negative edge-triggered flip-flip. This prevents early transitions on the bitlines, and allows enough time for the wordlines to properly settle to their new values. The waveforms for a writing a “0” and writing a “1” into an SRAM cell are shown in Figures 5.4 and 5.5 respectively.

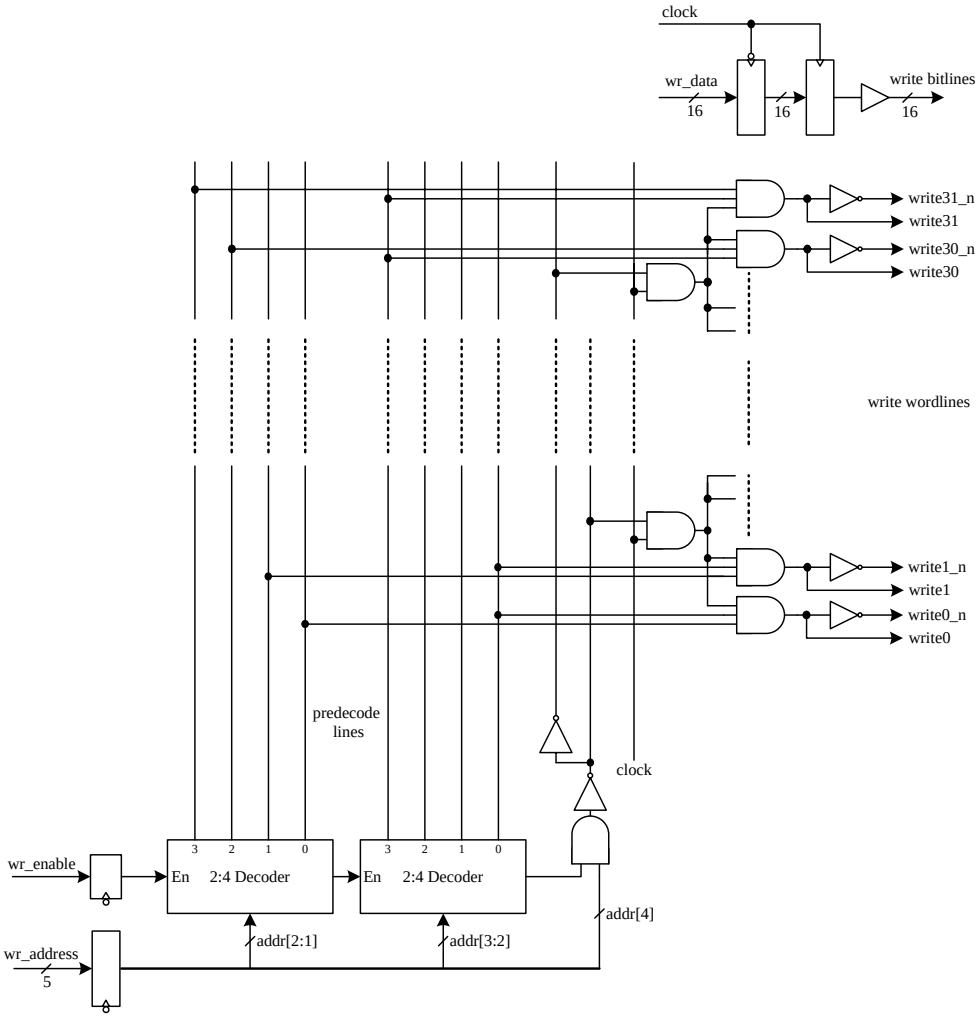

Figure 5.3: SRAM write control signal generation

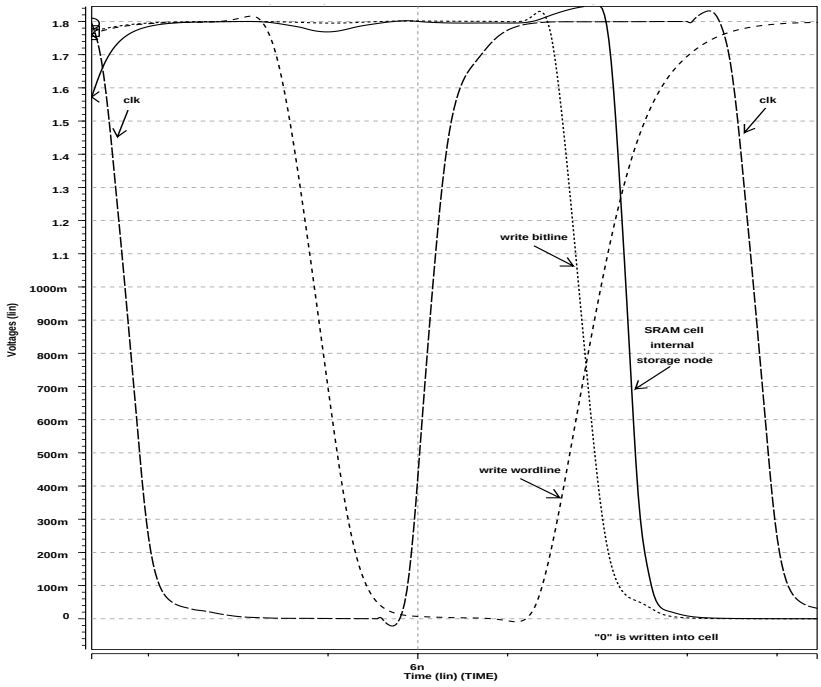

Figure 5.4: Simulation waveforms for writing a “0” to an SRAM cell

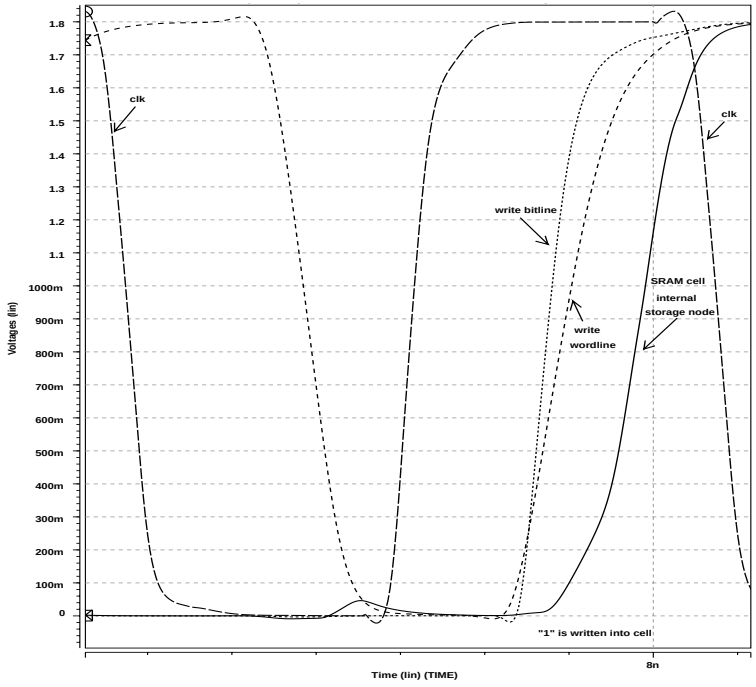

Figure 5.5: Simulation waveforms for writing a “1” to an SRAM cell

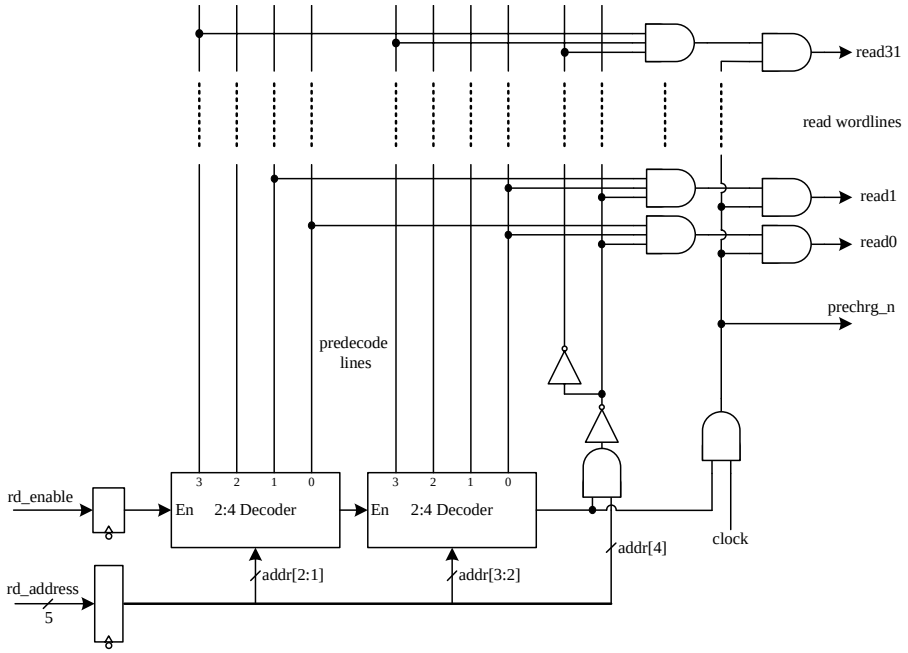

Figure 5.6: SRAM read control signal generation

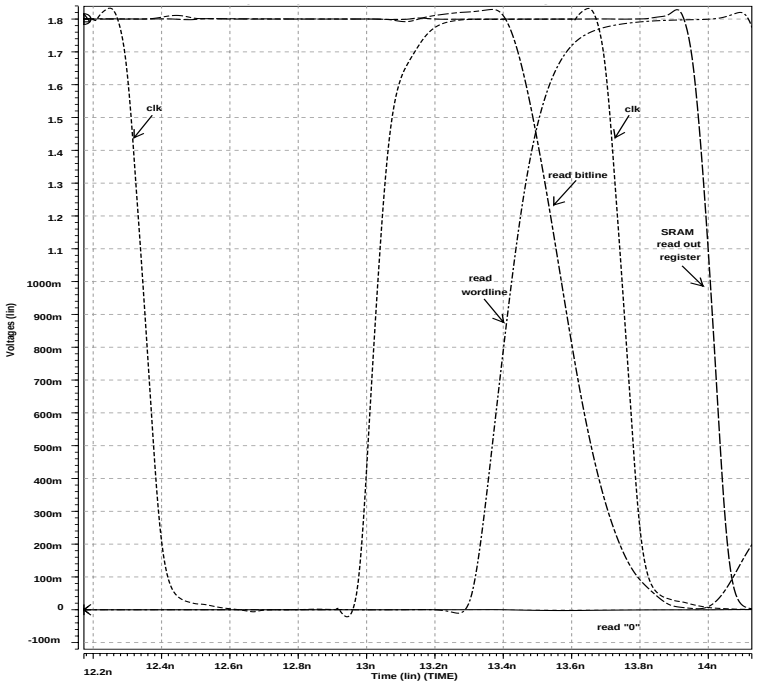

Figure 5.7: Simulation waveforms for reading a “0” from an SRAM cell

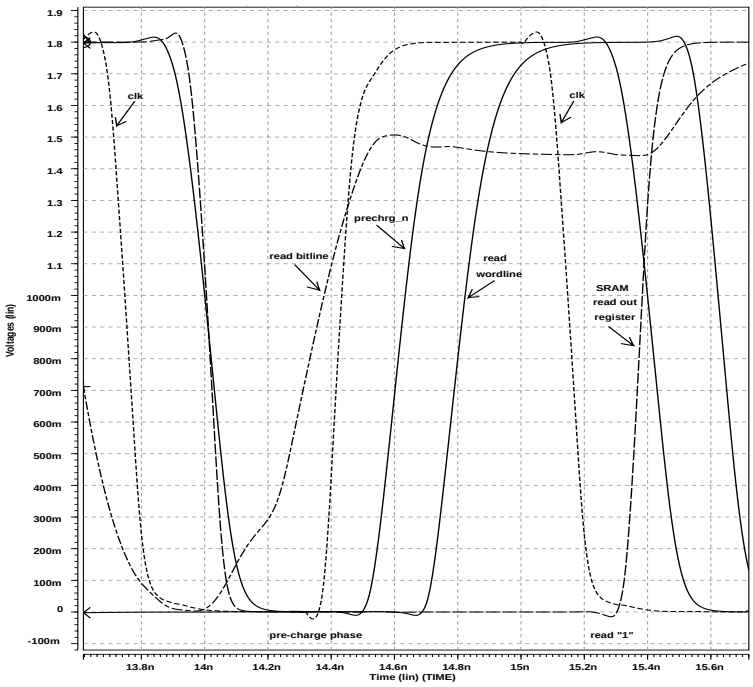

Figure 5.8: Simulation waveforms for reading a "1" from an SRAM cell

Read circuitry

The read control logic is shown in Figure 5.6 and is similar to the write control logic. Predecoders are used in the first stage of logic followed by wordline driving logic. The primary difference on the read logic is that the clock gate signal enters directly into the wordline drivers. This allows the read wordlines to toggle even more closely in relation to the clock. It also generates a pre-charge signal that has a more optimal timing relationship to the read wordline signals than the clock signal itself. It is desirable for power savings to avoid the situation where both the pull-up transistor and a pull-down transistor (in the cell) are "on" at the same time. If the clock signal is used directly for the pre-charge signal, the time period where both devices are "on" is on the order of 400 ps. With the given topology there is a still a small period time—approximately 150 ps—where the two signals do overlap. There are likely possibilities to further minimize this conflict, however, due to design time constraints this was deemed to be an acceptable amount of overlap. The waveforms for a reading a "0" and reading a "1" from an SRAM cell are shown in Figures 5.7 and 5.8 respectively. Weak PMOS keepers that can be turned "on" and "off" with a configuration bit are included in the design. These can be activated to compensate for the droop in the read bitline when the pull-up device is "off" and no cells are pulling the bitline down. This is particularly

useful for very slow speed operation or if the PMOS mobility is low after fabrication.

Performance

To more robustly test the final layout of the SRAM, additional capacitive loading was added into the extracted HSPICE circuits to better represent the large wire loads presented in the memory. Additionally, low-supply voltage operation was verified by running simulations at 1 V. The circuit operates correctly with this low-voltage supply at speeds of up to 300 Mhz. The final SRAM memory core has approximately 30,500 μm^2 of active area and the smallest rectangle it occupies is approximately 35,000 μm^2 (202 μm tall by 175 μm wide). Extracted simulation results indicate a maximum frequency of 865 MHz with a 1.8 V power supply.

5.3 Synchronizer Design

In FIFO designs utilizing unrelated clocks, asynchronous inputs must be properly synchronized. Generally, these inputs are multi-bit vectors, however, as shown in Chapter 4, this problem can be reduced to a single bit synchronization problem. As discussed in Chapter 3, the fundamental choice in synchronization strategy is either to use pausible clocks, or to allow enough resolution time to reduce the probability of synchronization failure to an acceptable level. The application will likely govern this decision. For this FIFO design, using pausible clocks at the interface was deemed to be unacceptable, due to complexity issues, the non-deterministic delays it would cause in the system, and the fact that each domain had several incoming and outgoing interfaces, making proper arbitration difficult. Once this was decided we chose a simple pipeline synchronizer (described in Sec. 3.2.1) for synchronizing the incoming address pointers. In order to have reliability control for characterization purposes, the number of stages in the synchronizer is configurable. Additionally, since the target system can run at various frequencies, a configurable length synchronizer decouples the resolution time from the clock frequency. This way at high frequencies, several stages can be used to ensure reliability, but at lower frequencies the number of stages can be reduced, while the mean failure rate and total latency remain the same. The architecture for the synchronizer circuit is shown in Figure 5.9. The unsynchronized path (Config= 0) was included for metastability testing purposes.

Once the architecture is chosen the actual synchronizing circuit needs to be designed. Traditionally, a memory element—usually a flip-flop or latch—is used for this task. The target application for this project also requires that the design be synthesizable. Accordingly, the amount

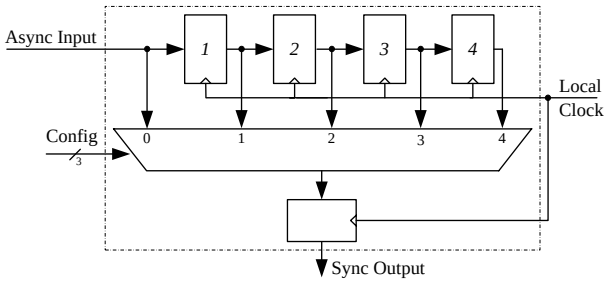

Figure 5.9: Synchronizer element with configurable metastability resolution time

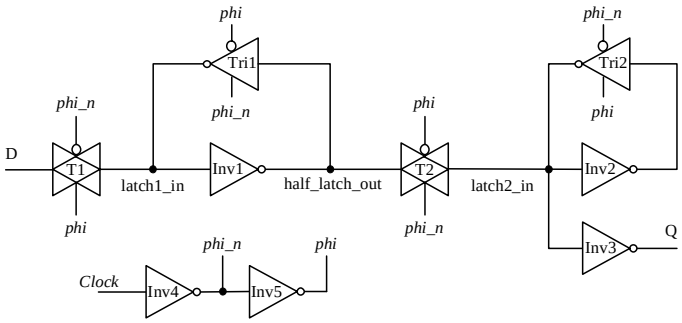

Figure 5.10: Schematic of negative edge triggered synchronizer flip-flop with local clock buffers

of non-standard library elements must be minimized. For this reason, and simplicity, we used a standard D flip-flop for the synchronizer, similar to the flip-flop used in the remainder of the target system. For the custom layout target we were able to make sizing adjustments and measurements to the circuit in order to optimize and characterize its metastability performance. The flip-flop synchronizer circuit can be seen in Figure 5.10 along with its device sizes in Table 5.2.

When designing a synchronizer there are two device parameters that need to be considered. As described in Chapter 3 these are the metastability time constant(τ) and the normalized aperture in which metastability can occur (T_0). There has been substantial research into the area of improving synchronizer performance, although there still appears to be some disagreement about the best way to design an optimized synchronizing element. It is generally agreed upon that when designing a synchronization flip-flop it is important to keep the time constant of the flip-flop small. This results in a faster resolution times, and, therefore, a higher propensity for avoiding resolution failures. In order to have a small time constant, loading on the feedback loop of the bistable element needs to be minimized. Additionally, some types of flip-flops, such as dynamic flip-flops, have infinite time constants due to the lack of feedback, making them poor synchronizers. The metastability

	NMOS width (λ)	PMOS width (λ)
Inv1	10	19
Inv2	5	5
Inv3	10	26
Inv4	5	10
Inv5	8	14
Tri1	5/5	5/5
Tri2	5/5	5/5
T1	5	8
T2	5	12

Table 5.2: Device Sizing for Flip-flop Synchronizer of Figure 5.10. (λ = 0.09 μm for a 0.18 μm process). For tri-state devices there are two devices in series for both the pull-up and pull-down.

time constant τ, introduced in Section 3.1, can be minimized in the same manner. It has been shown that the parameter τ is related to the gain-bandwidth product of the first stage in the flip-flop [18] [24]. The relationship has been established with τ equal to the inverse of the gain-bandwidth product of the synchronizer [25]. This means that ideal synchronizers have high current drive (*i.e.,* large transconductance (g_m) values) and small node capacitances. Synchronizers using bistable inverters have been shown to yield better results than those constructed out of more complex gates [28]. Based on the gain-bandwidth requirement, this makes sense, since inverters present less loading and have higher gain than other types of logic gates. Given a synchronizer topology utilizing bistable inverters, as shown in Figure 5.10, it has been suggested that the optimum device width ratio is 1:1(PMOS:NMOS) for maximizing the gain-bandwidth product of the synchronizer [23] [50]. Unfortunately, when considering the MTBF equation for a pipelined synchronizer (Eq. 3.1 and 3.4), both τ and t_{ff} appear in the exponent. While the specified ratio can improve gain-bandwidth, it can increase the propagation time of the flip-flop, especially in technologies with low PMOS device mobilities. Accordingly, optimizing entirely for gain-bandwidth is not necessarily the best approach. Additionally, there has been debate about the accuracy of the relationship between gain-bandwidth and τ [51]. It has also been shown that in two-stage flip-flops—master-slave—the synchronization task is left up to the first stage of the flip-flop and often any metastability is isolated to the internal nodes and only a delayed flip-flop output response is seen [18]. It is still possible to sample a metastable value at the output because the transition time is non-deterministic, making it asynchronous to the next device. However, it is less likely than if the output of the first flip-flop stage were used because that stage's output will sit at a logically undefined state while the flip-flop resolves any conflicts.

When hardware is not available, it is useful to use circuit simulations to estimate metastability performance. Methods for using simulations to approximate metastability characteristics have

been published [22] [24] [25], but simulations can be hard to control [50]. Additionally, time steps on the order of 0.1 ps or less are required to make these types of measurements [28]. However, these techniques have been utilized to achieve reasonable approximations to the final hardware values [26]. Accordingly, it is still useful to simulate these values to get a general idea of failure rates prior to committing a design to hardware.

The layout for the flip-flop design in Figure 5.10 was extracted to include all parasitic capacitances greater than 0.1 fF. Numerical circuit simulations in HSPICE were used to estimate metastability performance. Figure 5.11 shows the simulation results from a setup time violation of this flip-flop. This was accomplished by iteratively running simulations and shifting the D (*data_in*) input transition until values were found that caused the flip-flop to behave incorrectly. In this case, the input change occurs about 40 ps after the clock input (*phi_in*) transitions. The first stage is forced into a metastable state with both internal nodes (*latch1_in* and *half_latch_out*) sitting in the logically undefined region near $V_{DD}/2$. As noted above, while the first stage of the flip-flop is resolving the input conflict the output of the second stage is being held, and is not forced into the metastable state. This simulation was performed at 100°C and a V_{DD} of 1.62 volts with a resolution time of 0.01 ps. The simulation conditions were chosen because increasing the temperature and lowering the supply voltage increases a devices susceptibility to metastability [50] [24].

We chose the simulation methods described by Jex and Dike [25] for estimating the metastability time constant τ, and we chose Portmann and Meng's [24] method for estimating T_0. Determining T_0 by simulation is more difficult and less accurate than τ [51]. However, since τ appears in the exponent of the MTBF equation (3.1), and T_0 does not, variations or inaccuracies in τ have a much greater effect on the estimate than variations in T_0.

The method for determining τ by simulation consists of using a voltage controlled switch to force the internal nodes (*latch1_in* and *half_latch_out* in Fig. 5.10) into the metastable state. This is accomplished by first shorting these nodes together with a low resistance (0.001 Ω) switch. The *clock* input is statically tied low to enable gates $T2$ and $Tri1$ and disable $T1$ and $Tri2$. Even though some of the devices will always be disabled in this simulation, they were left in to accurately model the internal loading conditions. The switch was left closed for 10 ns and then opened with a control voltage pulse. The "off" resistance of the switch was set to 100 MΩ. At this point numerical simulator noise will likely cause the nodes to drift apart, but a small power supply, on the order of 1 μV, can be added in series with the switch to force the nodes apart [25]. The time constant at which the nodes drift apart gives the desired approximation for τ. By plotting the difference between the

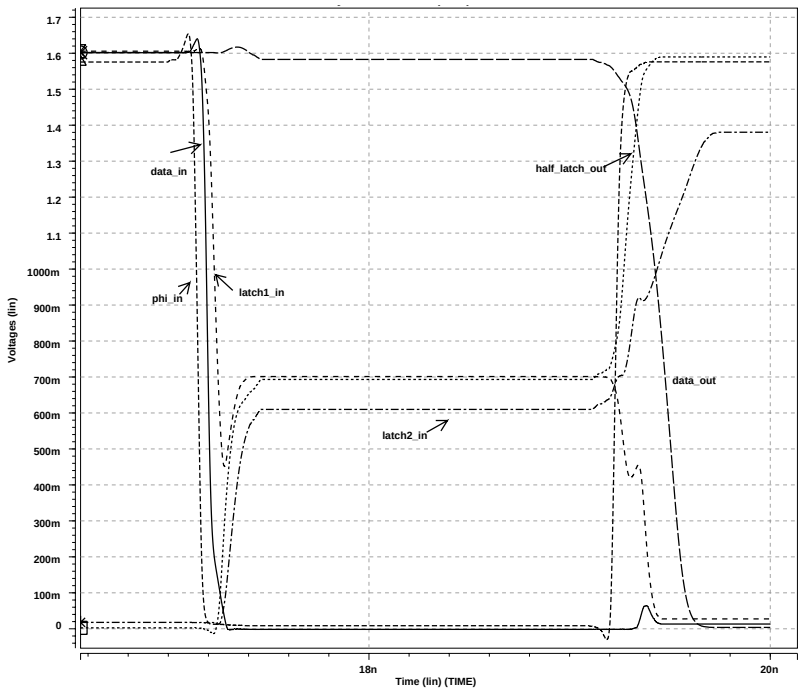

Figure 5.11: Metastable event of D flip-flop in HSPICE circuit simulator

Test Condition	T_0 (ps)	τ (ps)
Data transition high to low	600	-
Data transition low to high	720	-
Forced to metastable point and released	-	30.5

Table 5.3: Metastability parameters from HSPICE numerical simulator (test conditions = 100°C and V_{DD} = 1.62V)

two nodes on a semilog plot the time constant can be found using Equation 5.1 [25].

$$\tau = \frac{t_1 - t_2}{\log_e(V_2/V_1)} \tag{5.1}$$

An example waveform set from an HSPICE simulation of this test can be seen in Figure 5.12. The estimated worst case τ measurement for the synchronizer design shown in Figure 5.10 is listed in Table 5.3.

In order to estimate T_0, iterative simulations were run on the synchronizer flip-flop. Two sets of simulations were run, one for a data transition high to low and one for a data transition low to high. The process consists of moving the data transition edge through the critical window of the flip-flop. The two pieces of information to be collected from the simulation are, 1) the time difference between the active clock edge and the data transition (D) and, 2) the time difference between the active edge of the clock and the output (Q) transition. After a first pass through the critical window

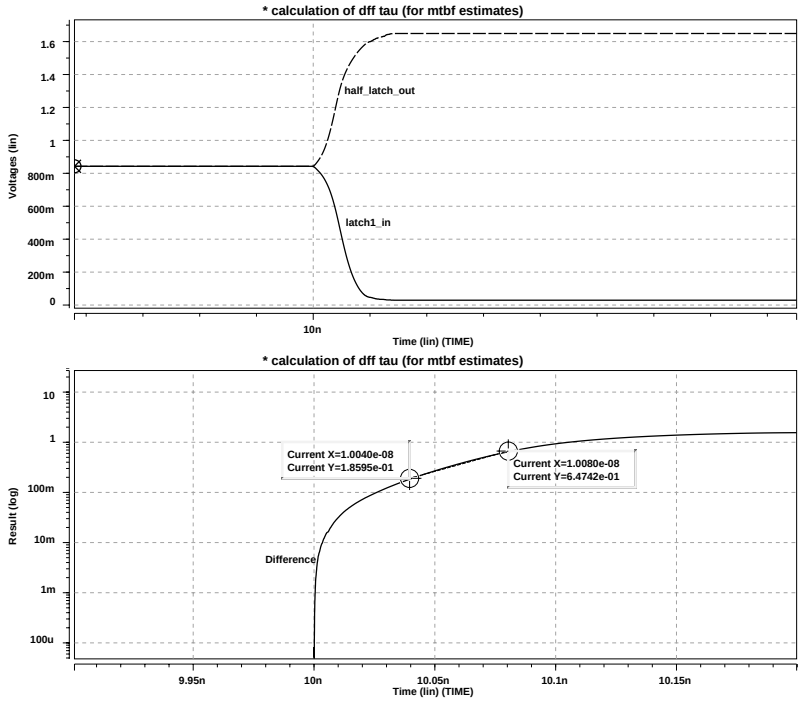

Figure 5.12: Measuring the τ parameter for synchronizer (T= 100°C and V_{DD}= 1.6 V)

with a large increment size *(∼100 ps)*, smaller windows are used with a smaller increment size. The focal point of the exercise is the window where deep metastability occurs. This is defined as the region where the clock to data output becomes non-deterministic, and is not simply a delayed version of the correct output [25] (*i.e.,* larger than normal clock–to–Q times). In HSPICE this region is the area where the flip-flop output resolution direction is governed by numerical noise in the simulator, and no longer by the input data value. An example of this simulation state is shown in Figure 5.11. The data transition delay was set to 17.04444ns. If the simulation is re-run with a ± 0.1ps change in the delay (*i.e.,* shifting the data transition by that amount) the resulting simulation output changes. This indicates that this transition is occurring within the deep metastability region. The beginning of this region is indicated by an extremely steep increase in the clock–to–Q time of the flip-flop and also an inconsistency in the output results when small shifts are made in the data transition location. A plot of the data collected from one of these simulation sets is shown in Figure 5.13. Failed transitions—ones that did not cause an output transition—are not shown for readability. The collected data matches nicely to the generalized approximation for a flip-flop's output response shown in Figure 3.1. In this plot, deep metastability begins at approximately–50 ps on the x-axis. As mentioned in Section 3.1, the T_0 parameter is the asymptotic width of the time aperture that

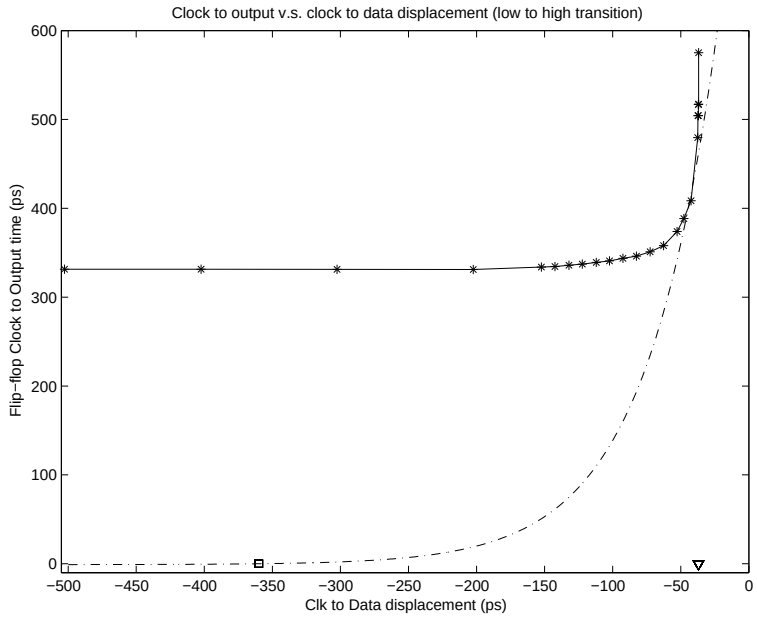

Figure 5.13: Plot of HSPICE data used for determining T_0 parameter (dashed line is fitted exponential approximation, "□"= zero crossing and "∇"= t_{meta})

the device enters metastability. Targeting just the deep, or true, metastability region provides more accurate failure results than if the region is included where the flip-flop output response is just delayed [26]. The shifted and fitted exponential function shown in Figure 5.13 estimates this exponential aperture width. The "∇" symbol near–40 ps represents the t_{meta} point as shown in Figure 3.1. The "□" represents the zero crossing of the exponential. The magnitude of the time difference on the x-axis between the exponential zero crossing and t_{meta} is equal to $T_0/2$ [24]. The estimated worst case T_0 measurements for the synchronizer design shown in Figure 5.10 are listed in Table 5.3.

Based on prior work in the area of synchronizers, the results shown in Table 5.3 seem reasonable. A jamb latch architecture has been shown to have very good τ values [26]. In a 0.25 μm CMOS process, simulations estimated τ and T_0 for a jamb latch to be 20 ps and 15 ps respectively [26]. This was then validated on silicon. Simulated estimates of τ and T_0 for several flip-flop architectures in a 0.6 μm CMOS process were determined to be in the range of 56 ps–110 ps and 7 ns–73 ns respectively [27]. The dramatically larger values of T_0 compared with the other estimate and this work, are attributed to the larger process dimensions and the fact that they did not target deep metastability.

Table 5.4 shows the estimated mean time between failures of our synchronizer architecture

MTBF	Number of Synchronizing Flip-flops (N)			
Clock Frequency	2	3	4	5
1000 MHz	25.81 sec	39×10^3 years	1.86×10^{15} years	8.91×10^{25} years
750 MHz	33.5 days	2.44×10^{14} years	6.41×10^{29} years	1.69×10^{45} years
500 MHz	5.64×10^8 years	4.69×10^{33} years	3.86×10^{58} years	3.20×10^{83} years

Table 5.4: Estimated mean time between failures using synchronizer from Fig. 5.9 and worst case simulated device parameters (T_0 = 720 ps , τ = 30.5 ps, t_{ff} = 300 ps and t_{mux} = 50 ps)

(Fig. 5.9) utilizing the synchronizing flip-flops shown in Figure 5.10. To make the estimates, the worst case parameter measurements were used in conjunction with Equations 3.1 and 3.5. In this case, there will also be a $t_{mux} = 50$ ps value subtracted from the resolution time, to account for the selection mux clock to data delay. This mux delay can be made fairly small, since the configuration values generally do not change during run-time. Given that the results for a two flip-flop synchronizer in the frequency range of the target application show MTBFs of greater than 1 billion years, the failure rates are deemed to be acceptably low.

5.4 Gray/Binary Converters

5.4.1 Background

A *Gray code* is a digital code where only one bit changes between successive code words [13]. Constructing a Gray code sequence of an arbitrary bit length can be done recursively. Two possible methods for generating a Gray code sequence are outlined by Wakerly [13]. As discussed in Sections 3.3 and 4.2.1 , this is the type of encoding required when synchronizing a multi-bit vector with flip-flops. However, as shown in Chapter 2 and mentioned in Section 4.2.1, it is more convenient to perform mathematical manipulations with a binary encoded vector. In the FIFO design this results in a scheme where vectors are mapped into a Gray code when being transferred across the clock boundary and are then mapped back into binary for calculations once they are synchronized into the clock domain.

Converting binary values to Gray code is straightforward. Given an n-bit binary vector $(b_{n-1}, b_{n-2}, \ \ldots\ , b_1, b_0)$, the Equations in 5.2 can be used to convert to an n-bit Gray coded vector

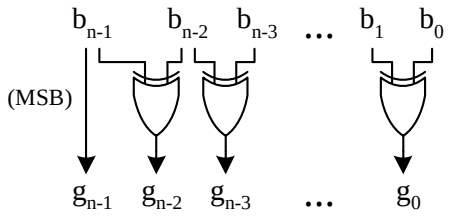

Figure 5.14: Binary to Gray code conversion circuit

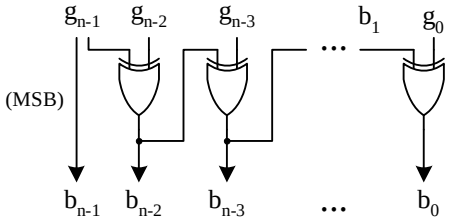

Figure 5.15: Generalized N-bit Gray to binary converter architecture

$(g_{n-1}, g_{n-2}, \ \ldots\ , g_1, g_0)$, where " $+$ " indicates the sum ignoring the carry.

$$\begin{aligned} \text{(MSB)}\ g_{n-1} &= b_{n-1} \\ g_{n-2} &= b_{n-1} + b_{n-2} \\ g_{n-3} &= b_{n-2} + b_{n-3} \\ &\ldots \\ g_0 &= b_1 + b_0 \end{aligned} \tag{5.2}$$

This can be accomplished using the XOR function and a general circuit architecture for performing this conversion is shown in Figure 5.14. Since each bit can be calculated in parallel, the worst case gate delay for this circuit for any n-bit vector is one XOR gate. The calculation requires $n-1$ XOR gates.

The reverse conversion from Gray to binary is similar to the above case. Given an n-bit Gray coded vector $(g_{n-1}, g_{n-2}, \ \ldots\ , g_1, g_0)$, the Equations in 5.3 can be used to convert to an n-bit

binary vector $(b_{n-1}, b_{n-2}, \ ... \ , b_1, b_0)$, where " $+$ " indicates the sum ignoring the carry.

$$\begin{aligned} \text{(MSB)} \ b_{n-1} &= g_{n-1} \\ b_{n-2} &= b_{n-1} \ + \ g_{n-2} \\ b_{n-3} &= b_{n-2} \ + \ g_{n-3} \\ &\cdots \\ b_0 &= b_1 \ + \ g_0 \end{aligned} \tag{5.3}$$

Unlike the previous case, in this calculation all the bits—aside from the MSB—require the next most significant binary bit output as an input. This results in a worst case gate delay of $n-1$ XOR gates for an n-bit vector and also requires a total of $n-1$ XOR gates. A general circuit architecture for this conversion can be seen in Figure 5.15.

5.4.2 Circuit design

The 32-entry FIFO needed for the target application requires a 6-bit address field to be converted to Gray code for transmission between clock domains. For a 6-bit input vector, the binary to Gray converter shown in Figure 5.14, will have a worst case delay of one XOR gate. A fast XOR gate can be constructed from 6 transistors. The delay of this gate was measured to be roughly 150 ps. However, this circuit has a path that consists of only two pass transistors. To increase the robustness and the output drive capabilities of the gate, a similar topology with an inverter output buffer can be used. However, this increases the propagation delay by approximately 50 ps, in the worst case. An example of this circuit can be seen in the sum logic of Figure 5.19. Circuit simulations on the extracted converter circuit indicate that the worst case propagation delay for the module was 200 ps with a default load capacitance.

If the architecture in Figure 5.15 were used for the Gray to binary conversion, the worst case delay would be 5 XOR gates in series, yielding a propagation delay of approximately 1 ns based on the above results. This logic block is in series with several other pieces of slow logic, which ends up creating the worst case delay path in the first stage of the FIFO. Therefore, reducing the delay here is beneficial to increasing the overall speed of the module.

The long propagation time is due to the ripple effect in the circuit—*i.e.,* less significant bits require information from higher bit calculations before they can calculate their results. A similar problem is found in ripple-carry adders [1]. A *carry-select* architecture is a straightforward way to reduce the worst case propagation delay of such adders [1]. A similar technique can be used for

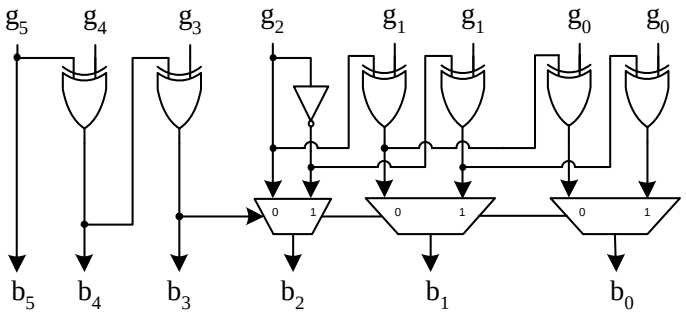

Figure 5.16: Six-bit Gray to binary conversion circuit

the Gray to binary converter circuit. A 6-bit implementation utilizing this technique can be seen in Figure 5.16. The main idea is to break apart the ripple path. Since this circuit has a ripple path of 5 gates, breaking it more than once would result in significantly larger area with little speed-up, accordingly it is broken only once. The circuit hardware is then duplicated and one set of gates is used to calculate the result if the ripple bit incoming to the breakpoint was a "1" and the other calculates the result if the incoming ripple bit was a "0". The two values are then multiplexed and the true ripple value is used to pick the correct result. For this architecture, the critical delay path is reduced from 5 XOR gates, to either 2 XOR gates plus a mux control input to output, or an inverter plus 2 XOR gates plus a mux data input to output. The extracted layout circuit simulations showed a worst case delay of 520 ps. This achieves a 48% reduction in delay from the normal architecture. The cost in circuit area is about an 81% increase from 185 μm^2 to 336 μm^2. However, the actual physical area increase is small compared to the total design size and is a worthwhile cost for reducing the delay by nearly one half.

5.5 Binary Incrementers

In order to keep track of the address pointer locations, an unsigned binary incrementer is required. The primary component to this module is a plus one adder. The address pointers are six bits, so a standard unsigned binary adder could be used to compute this sum. However, because the module needs to do only two operations—hold and increment by one—the module can be made smaller and faster by designing custom logic for the plus one operation. This module also has a ripple-carry path, so a carry-select architecture was used to reduce the propagation delay. A diagram of the plus one adder module is shown in Figure 5.17. In this case, implementing the carry-select architecture had little impact on the active area, which resulted in only a 17% increase from 290

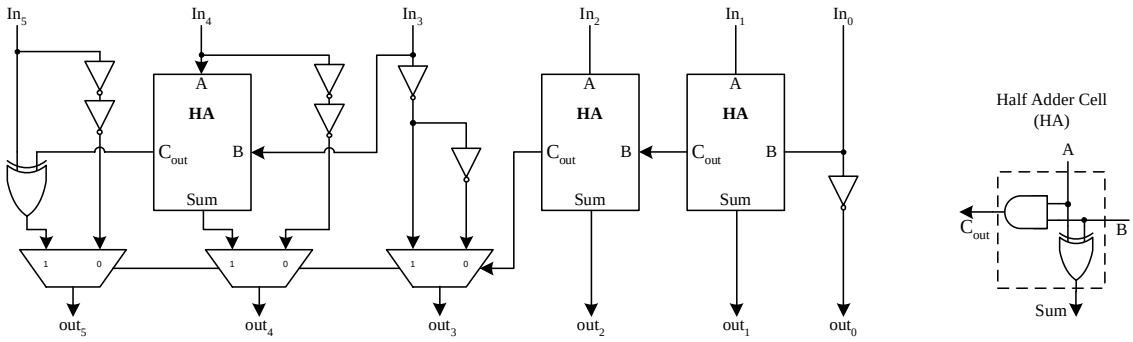

Figure 5.17: Six bit binary incrementer circuit

μm^2 to 340 μm^2. The AND gates used for the half-adder carry have delays of 125 ps, and the XOR gates had a 200 ps delay. Without the carry-select architecture, the total delay would have been 4-AND gates plus one XOR gate, yielding a total delay of 700 ps. Again this logic is in series with other logic in the pipeline stage, making this a critical delay. The carry-select reduced the worst case delay to 370 ps, which is an 89.5% speed-up.

The remainder of the incrementer module consists of two six-bit multiplexers and a six-bit register. The two control inputs are *increment* and *reset*. The complete module is shown in Figures 5.22 and 5.23. The *rd_request* signal coming into the FIFO arrives near the end of the clock cycle because of the amount of upstream logic required to generate it. Since the plus one adder takes some time and its result is needed by other units, the add is always computed as soon as the incrementer register changes. This way the result is ready by the time the *rd_request* signal arrives, and other units are not waiting for this signal to begin their computations.

5.6 Reserve Logic

The calculation of the "In Reserve" (*i.e.,* the FIFO cannot accept anymore data) signal is the most complicated arithmetic logic required in the design. The necessary operation is given in Equation 2.3. As noted previously, the form of the second equation is easier to implement in hardware and that is the form chosen here. For this implementation each of the input address vectors are 6-bits wide. The reserve constant needs to be only 5-bits since its value should always be less than the number of FIFO memory locations, which is 32 in this case. However, here it is just shown to be a standard input. A dot diagram detailing the operation is shown in Figure 5.18. Each dot represents a bit in the calculation. The first step taken is to use a full adder—also called

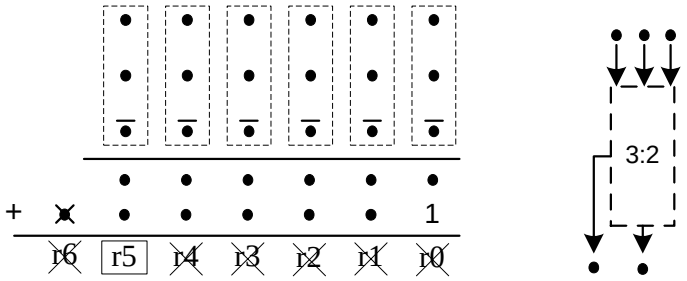

Figure 5.18: Dot diagram representing "In Reserve" adder function

a 3:2 compressor in this context—to compress the three input vectors into two vectors in carry-save format. The 3:2 function is shown to the right of the dot diagram. An interesting part of this adder design is that only bit used is the MSB ($r5$ in this case). If this bit is high then the space left in the FIFO is less than or equal to the amount of reserve space. The values with "X"'s through them are "don't care" values and are not necessary for determining the desired output $r5$. To perform the subtraction of the read pointer it can be converted to a negative number and then added. Accordingly, it needs to be inverted and a one needs to be added in the LSB. The inversion occurs before the vector enters the first row of 3:2s. Once the vectors are compressed, a final add needs to take place to transform the values out of carry-save format. This requires a standard binary adder. To achieve a lower propagation delay, we chose to use a carry-select implementation for this adder. Because this logic is in series with the Gray to binary conversion it was important to make it fast.

The full adder (3:2) implementation used for this module is shown in Figure 5.19. The 3-input XOR function used to compute the Sum output is built with a two-level structure. It is beneficial to choose the inputs to this structure carefully such that the latest arriving signal is connected to the C_{in} input. The earliest arriving input should be connected to the slower of the other two, which is B in this case. This way the first XOR has all ready computed its result and only one XOR delay is left when the late input arrives. The inputs are also internally connected to the carry logic with the same ordering preference to minimize the body-effect on the transistors in that logic. This carry-logic is built using a mirror adder [1] topology. This calculates the carry with a single stage, instead of the two stages required for a standard sum-of-products implementation. The caveat is that the signal produced is the inverted C_{out} value, however, this can be dealt with at the architecture level of the adder, without requiring the use of inverters in the carry-ripple path [1].

The final architecture for the reserve logic is shown in Figure 5.20. As noted, the ripple-

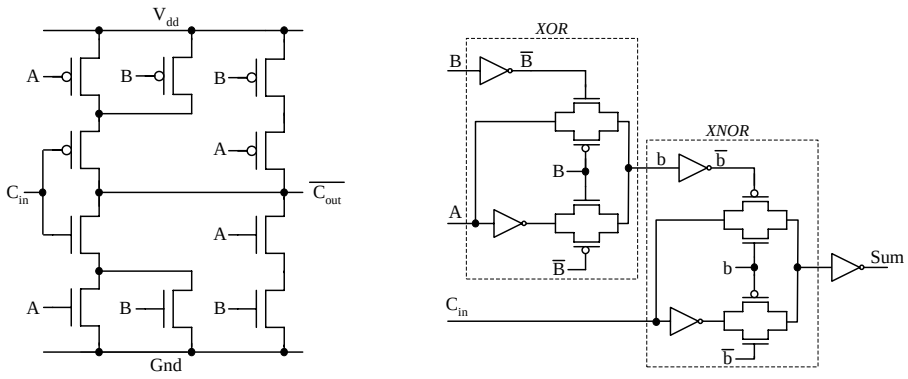

Figure 5.19: Hardware implementation of a full adder circuit

adder is broken apart, and the second ripple-carry $\overline{C_{out}}$ value (Co2 in Fig. 5.20) is used to select the correct output. To account for the inversion of the carry bit, every other carry-logic block has all of its inputs inverted. As shown in Figure 5.19, the carry logic used in the full adder implementation produces the inverted C_{out} as noted by the bubbles on these outputs from the first rows of 3:2s. Instead of adding an inverter and then subsequently adding an additional inverter to each carry output, every other set of inputs to the second row of carry logic is inverted. This produces alternating C_{out} and $\overline{C_{out}}$ values. The $\overline{C_{out}}$ values are indicated by inversion bubbles on the C_{out} ports of the logic blocks. The standard carry out logic is shown in Equation 5.4.

$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in} \tag{5.4}$$

Equation 5.6 shows the logic function that is implemented with the mirror adder carry logic. Through logical manipulation it can be shown that equations 5.5 and equations 5.6 are equivalent.

$$\overline{C_{out}} = \overline{(A \cdot B + A \cdot C_{in} + B \cdot C_{in})} \tag{5.5}$$

$$\overline{C_{out}} = \overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C_{in}} + \overline{B} \cdot \overline{C_{in}} \tag{5.6}$$

The result of this is that by complementing all of the inputs to the mirror carry cell, the uncomplemented version of C_{out} can be generated. This principle can be used to prevent having to add additional inverters in the ripple path. Instead inverters are added in parallel to some of the signals coming out of the first row of adders. This adds only one extra inverter delay to the worst case path instead of one per carry-ripple stage.

As mentioned above the input arrivals times should be matched to the appropriate input. In this case, the choices were easy to make because the input arrivals have a clear ordering. As shown in Figure 5.22, the three inputs to the adder are the write pointer, read pointer and reserve

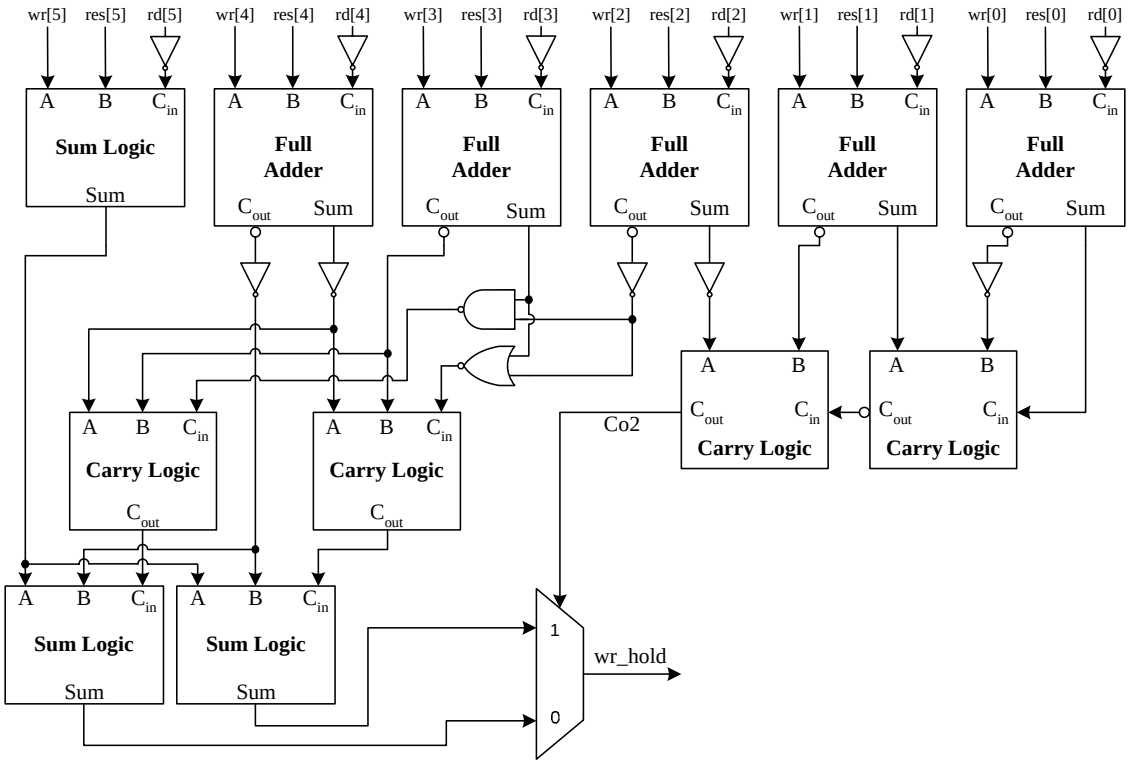

Figure 5.20: Three input, six bit adder for determing the "In Reserve" condition

value. The reserve value is static at run-time so it will clearly be the first available input. The write pointer comes directly out of a register, so it will be available after approximately 250 ps. The read pointer is the slow input and will average about 350 ps with 535 ps in the typical worst case.

The final circuit layout has an active area of 908 μm^2. The total worst case propagation delay for the circuit is 515 ps. The first row of adders takes approximately 200 ps for both the carry and sum logic, although this may be slightly improved based on the input arrival ordering mentioned above. The remainder of the worst case delay path is from the inputs to the NAND and NOR gates through the carry logic, through the sum logic and then through the output selection multiplexor.

5.7 Comparators

In order to determine the FIFO empty condition, the two address pointers need to be compared. If all six bits are equal then the FIFO is in the empty state. The implementation of the comparator logic is straightforward and logically consists of a bitwise XNOR of the two input vectors followed by an AND-word operation. In this case, a six input AND gate is needed, requiring a worst

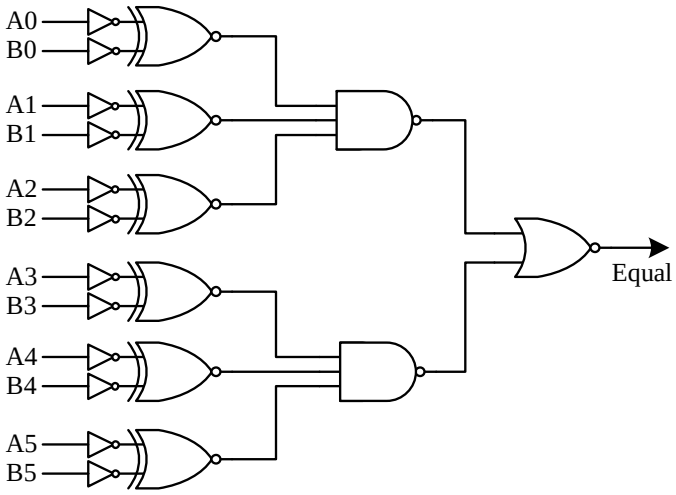

Figure 5.21: Six bit comparator for determing the "empty" condition

case path of six NMOS in series followed by a PMOS pull-up in the inverter. To better optimize the circuit, the AND function was split into two stages and then transformed into a NAND/NOR network. The resulting equation is logically implemented as shown in Figure 5.21. Minimum-sized inverters are placed on all inputs to reduce the input load capacitance and facilitate the use of a transmission gate XNOR without an output inverter. An example of this implementation is shown in the XNOR box of the second stage of the sum logic in Figure 5.19. Inverting all the inputs to the 2-input XNOR does not logically modify the gate because it is a symmetric function.

The final layout requires 300 μm^2 and the worst case delay of the extracted circuit is 315 ps.

5.8 Top-level FIFO Module

The final FIFO module requires the compilation of the modules described in Sections 5.2–5.7. Figure 4.1 shows a high-level diagram of the dual-clock FIFO design. As discussed, the final hardware module will be integrated into the AsAP processor. The FIFOs are placed in close proximity to the processor that is connected to the read side of the FIFO. The inputs and outputs for the write side of the FIFO are connected to the producer processor through longer busses. These buses are multiplexed just before they reach FIFO. The connections are controlled with configuration inputs so that they can be modified as required by the current application. The synchronizer configuration and *reserve* values are supplied to the FIFO externally by the local processor's configuration memory.

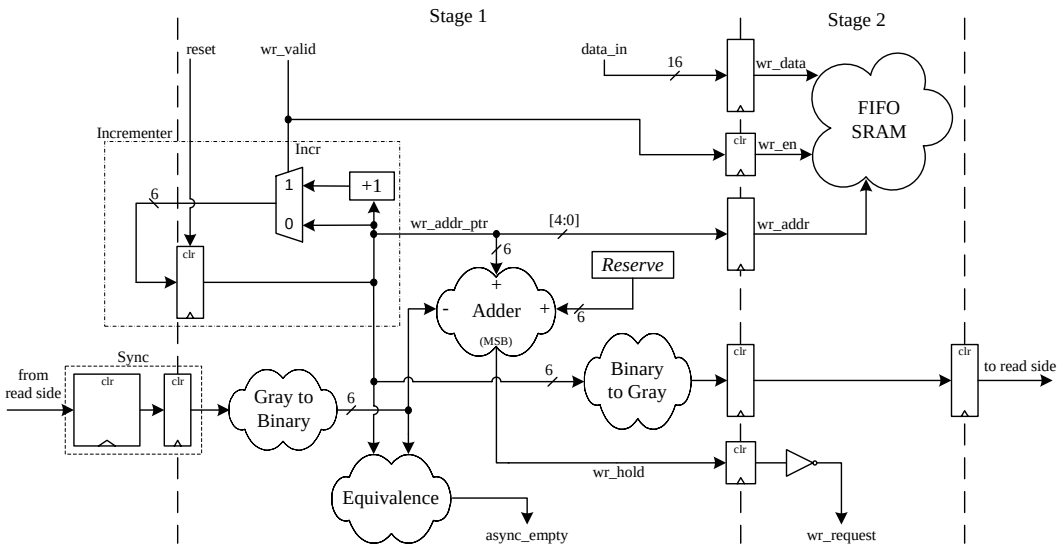

Figure 5.22: Pipeline diagram for the write side of the FIFO

The target application processor is pipelined for increased clock frequencies. The write side of the FIFO fits fairly easily into a pipelined design, since the reserve space is configurable and is accounted for in the adder module. A pipeline diagram for the write side of the FIFO is shown in 5.22. Wire latency and logic delays in the interface path are introduced in Section 2.2.2 and details for this design are discussed in Section 4.2.2. One design requirement from the AsAP architecture is that the write side of the FIFO must be asynchronously resettable. This is because the FIFO is located within the processor that is tied to the read side of the FIFO. The write side of the FIFO may or may not be in use. If the unit is not being used, it will not be supplied with a write clock. However, random initial values within the FIFO write side registers were shown to cause problems during application simulations. Accordingly, they need to be reset even if the FIFO write clock does not oscillate. All registers with asynchronous resets are marked with a *"clr"*.

The read side of the FIFO is also pipelined into two stages. Since this side is synchronous with respect to the local processor, all the resets on this side of the FIFO are synchronous. Registers with a reset capability are marked with a *"res"*. An additional concern is that the *rd_request* signal arrives towards the end of the clock period, so the logic is designed to be able to do most of its work prior to learning the true value of this signal.

As mentioned, both sides of the FIFO have an output that is used in the other clock domain even though it is asynchronous with respect to the other clock. This signal is used as an asynchronous wake-up for sleeping processors. For example, if the write side processor goes to sleep

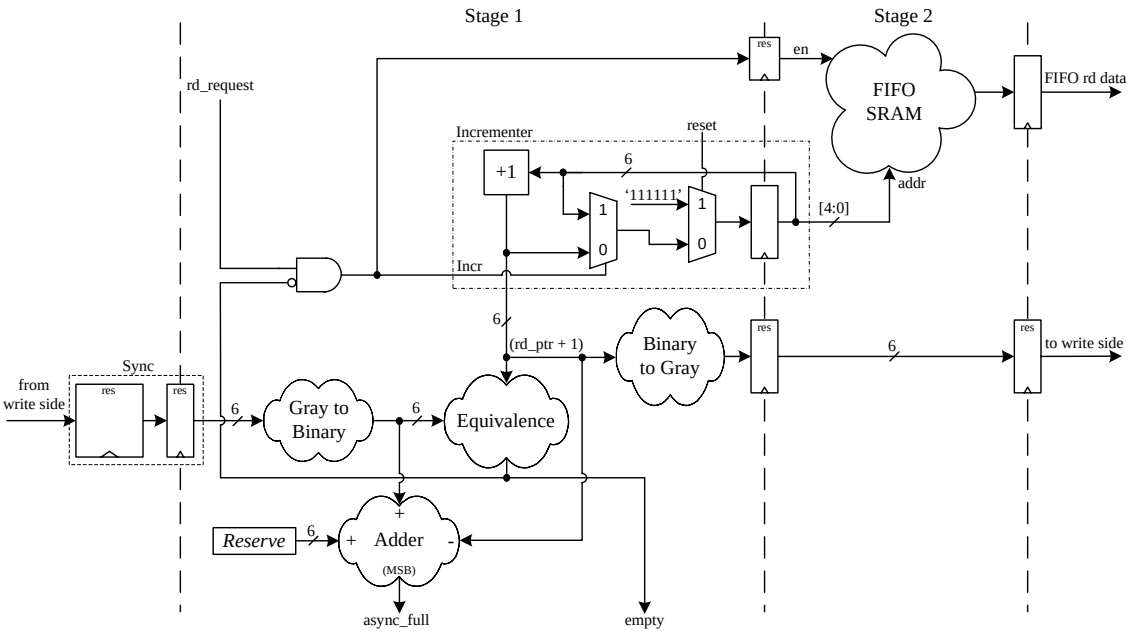

Figure 5.23: Pipeline diagram for the read side of the FIFO

waiting for the downstream FIFO to acquire free space, its clock will halt. This results in all the register values on the write side of the FIFO becoming frozen. Accordingly, the state of the write side of the FIFO is locked and it will never be able to indicate to the processor that the FIFO has free space (normally indicated with the *wr_request* signal). Accordingly, a replica of the logic to generate this signal is placed on the read side of the FIFO. In this way, when the read side begins emptying the FIFO it can indicate to the sleeping processor that there is free space so it can restart and complete its write to the downstream FIFO. The analogous case applies to the read side processor sleeping, waiting for data it needs to complete its current instruction to enter the FIFO. Since these signals are not being used synchronously within the clock domain that is generating them, there is no timing requirement for them.

As discussed, the FIFO employs a two line signalling convention. On the read side, the FIFO is always indicating whether or not it is empty. This is the active port. The interface logic responds to this signal. If the FIFO is not empty, it will supply a request signal indicating that it wants data. This data will be supplied one cycle later from the FIFO SRAM. On the write side, the FIFO supplies a request signal. If the FIFO is not requesting data, the producer should not try to write to the FIFO. Instead, it should wait until there is space in the FIFO (*i.e., wr_request* is asserted), and then signal that there is valid data using its control line (*wr_valid*).

	Units utilized per FIFO	Active area (μm^2)
SRAM	1	30,507
Synchronizer	2	3,867
Reserve logic	2	908
Comparator	2	300
Gray to binary conv.	2	336
Binary to Gray conv.	2	186
Incrementer	2	1530
Total		**44,761**

Table 5.5: Area breakdown for the FIFO hardware module

5.8.1 Performance and analysis

The final layout of the dual-clock FIFO module is shown in Figure 5.24 and has approximately 44,761 μm^2 of active area. The minimum rectangle it occupies is 66,500 μm^2 in area. The module shown is only the first pass layout. With further layout optimizations it would likely be possible to pack the modules more tightly and fit the design into a smaller rectangle. Table 5.5 shows the active areas for the individual components of the FIFO. The SRAM occupies the most area followed by the synchronizers. These modules occupy 67.7% and 17.5% of the active area respectively. In some applications the asynchronous wake-up signals would not be required. In this case, one reserve logic unit and one comparator unit could be removed, reducing the active are by about 1,200 μm^2.

The FIFO SRAM simulates correctly at 865 MHz with a 1.8 V supply. Unfortunately, the the first stage of the FIFO pipeline is the bottleneck in terms of speed, so the entire FIFO cannot operate at this speed.

Based on the HSPICE timing simulations, the critical path is equal on both sides of the FIFO. On the write side, this path consists of the Gray to binary converter module in series with the adder that produces the *wr_hold* signal. In order to increase the performance, the inverter that creates the *wr_request* signal was moved into the next pipeline stage. The total worst case delay in this path under typical conditions is 535 ps for the converter plus 515 ps for the adder plus a flip-time of 250 ps. The total delay is then 1.3 ns.

The read side's worst case path is through the Gray to binary converter, followed by the equivalence detector, then through an inverter/AND gate combination and final through two multiplexors. The respective delays are 535 ps, 315 ps, 150 ps and 50 ps. Combined with a flip-flop time of 250 ps, this results in a total delay of 1.3 ns. This assumes that the *rd_request* arrives before approximately 800 ps, which is reasonable based on the logic that is creating that signal. If that

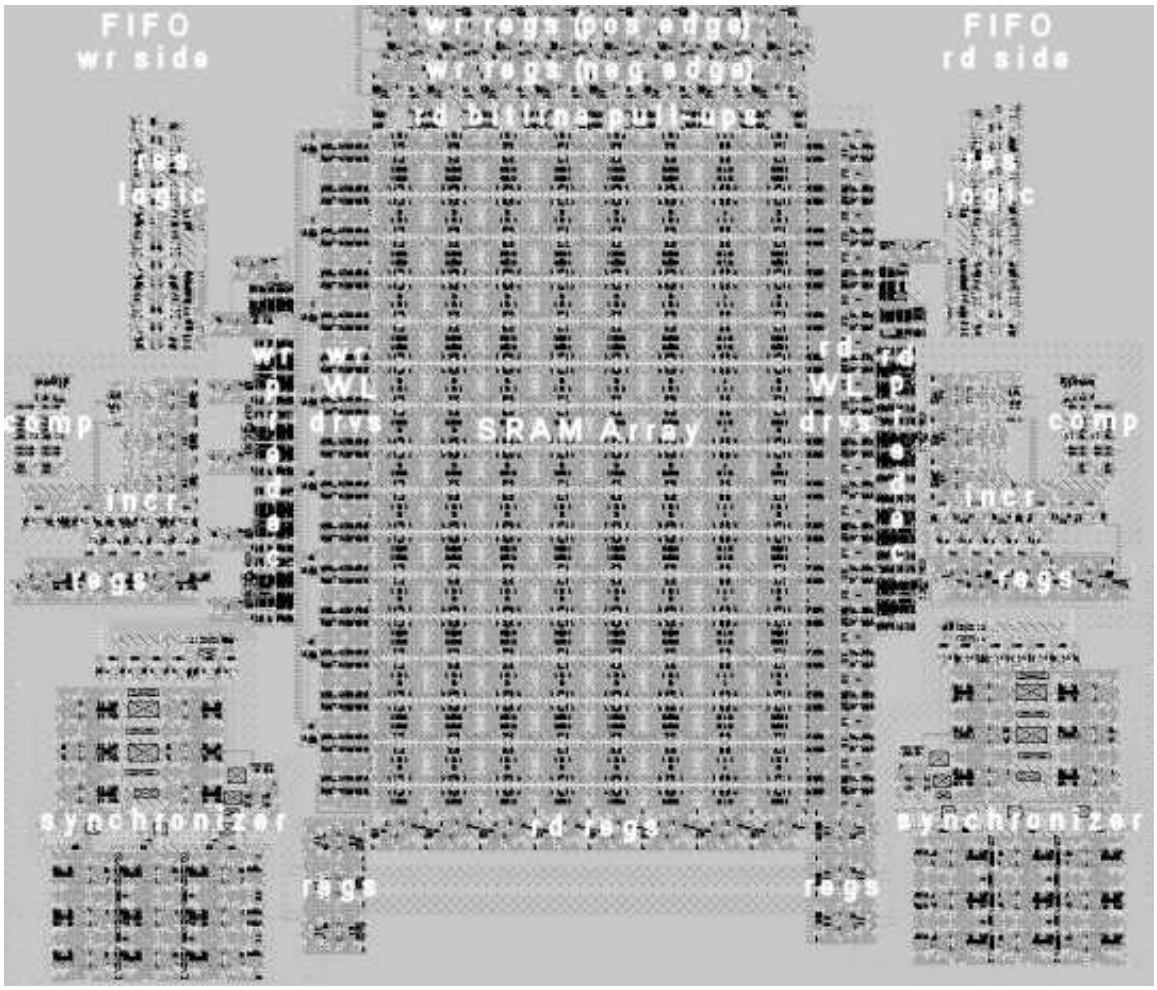

Figure 5.24: Final layout for the dual-clock FIFO module (incr= incrementer, comp= comparator, res= reserve, incr= incrementer, predec= predecoder, WL= wordline)

signal were delayed any longer than that, the read side would further limit the maximum frequency.

The resulting maximum clock frequency for the first stage, and subsequently the entire FIFO, is approximately 770 MHz.

As discussed in Chapter 2, some of the key performance metrics in a FIFO design are robustness, throughput, energy-efficiency, scalability, latency and clock rate.

For this design, robustness is the top design goal, so trade-offs (mainly speed) were made in order to ensure that the FIFO would operate robustly when fabricated.

Secondly, since the target application is an array of DSP processors, it is highly desirable to achieve a throughput of one datum per cycle. The final design can support a throughput of one datum per cycle up to its maximum clock frequency. This occurs when the consumption and production rates are similar, such that there are constant writes to and reads from the FIFO. This

throughput is not limited by the clock frequency, so if techniques were used to increase the peak clock frequency the throughput would remain at one datum per cycle.

The overall energy-efficiency will be more clear when hardware is available to make accurate power measurements. There are two main areas that will help to ensure the energy-efficiency of this design. The first is the utilization of enables within the memory core, which prevents high capacitance nodes inside the memory from switching unnecessarily. Secondly, through the support of dynamic frequency scaling, the FIFO can operate in a fashion where it can be sped up or slowed down at run time to more precisely match its requirements at a given time period, so it only uses enough energy to meet its current goal.

This design also achieves a high degree of scalability. Two primary components factor into the scalability of the design. The first is the FIFO logic itself and the second is the memory core. In the design demonstrated here, the FIFO logic is the bottleneck in terms of speed, however, as the overall FIFO size grows the memory will become the important factor. This is because the FIFO logic requires only minor changes, the highest impact being an address width increase of one bit every time the FIFO depth doubles. No control changes are required for the FIFO when the data width changes. As the memory size increases the latency will also increase. If the memory is adequately pipelined, the module will still be able to maintain a throughput of one datum per cycle. Accordingly, the two main issues that impede infinite scalability of the design are the throughput and latency requirements of the application. Being able to maintain the FIFO throughput is contingent upon being able to pipeline the memory enough to maintain the desired throughput through the memory core. The latency will likely become dominated by the memory as it continues to grow, so from this standpoint it is simply an issue of how much the latency of the memory can be reduced by and how much latency the application can tolerate.

The minimum latency of the design is the time it takes one item to propagate from the input of the FIFO to the output of the FIFO assuming that it does not have to wait for other items to be removed ahead of it. A case where this may happen is the situation where the consumer is constantly requesting data and the producer is supplying it infrequently. Determining a minimum latency for this FIFO is somewhat difficult due to the fact that the read and write side clocks are totally unrelated and dynamic, and total latency depends on both. To get a general idea of the size of this delay it can be measured for the case where the two clocks are operating at the same frequency, which is equal to the maximum frequency of the FIFO. This delay is also contingent upon the number of synchronizer stages utilized in the design. For this measurement two stages are used.

The latency can then be bounded to three write clock cycles plus three read clock cycles. The total worst case latency for this design is then six clock cycles, which yields a total latency of 7.8 ns at the minimum clock period of 1.3 ns. Depending on the relative phase of the two clocks, one of the clock cycle delays included above may be reduced to just one flip-flop clock-to-Q delay (250 ps). This is the case where the asynchronous signal is sampled immediately after it changes by the read side synchronizer. In this case the delay will be minimized and will be equal to 6.75 ns.

The final concern is more general and is the support of high clock rates. The demonstrated design can support clock rates of 770 MHz in a 0.18 μm process, which is fast enough for the target application. Higher clock rates can be achived by further device size optimization, utilizing a faster style of circuits, or changing the fabrication technology.

Chapter 6

Conclusion

6.1 Summary

The primary areas covered in this thesis are single-clock FIFOs, synchronization and metastability, dual-clock FIFO architectures, and a hardware implementation of a dual-clock FIFO.

This thesis provides a comprehensive exploration of dual-clock FIFO design. The module can be used for interfacing units with unrelated clocks in high-speed applications. The proposed architecture is well suited for all dual-clock applications and achieves high energy efficiency, good scalability and area utilization, and arbitrarily high robustness. This architecture can be utilized as a drop-in module to many applications. Additionally, it may also be customized using the trade-offs explored throughout this thesis.

The demonstrated hardware implementation occupies an active area of approximately 45,000 μm^2 in 0.18 μm CMOS technology and simulations indicate an operation range of up to 770 MHz under typical conditions.

6.2 Future Work

The VLSI implementation of the final hardware design is currently being integrated into AsAP project layout, which is projected to be fabricated in the near future.

Several areas will be of particular interest when the hardware returns. One of the primary questions is the actual robustness to metastability. As mentioned, in the AsAP chip design, there will be a parameter allowing the trade-off between latency and metastability to be exercised. It will be interesting to see how well the estimated MTBF values match the measured values in the hardware.

Additionally, investigations into utilizing alternative methods to synchronization within the FIFO would be worthwhile. The method employed in this case is considered the brute force method, and the synchronization circuits ended up occupying a substantial portion of the final layout. Another area that may be worth exploring is building the memory core out of alternative structures. Since the memory is the largest block in the FIFO, investigations into power and area reductions—while maintaining reliability—could have substantial impact on the overall module design. Furthermore, with additional circuit level and sizing optimizations it may be possible to improve the overall delay of the first stage of the FIFO. This would result in an increased operation range for the FIFO hardware design.

Bibliography

[1] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, A Design Perspective*, Prentice Hall, Upper Saddle River, NJ, 2003.

[2] R. Ho, K.W. Mai, and M.A. Horowitz, "The future of wires," in *Proceedings of the IEEE*, Apr. 2001, vol. 89, pp. 490–504.

[3] G. Semeraro, G. Magklis, and *et al.*, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *International Symposium on High-Performance Computer Architecture*, Feb. 2002, pp. 29–40.

[4] D. M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 1984.

[5] Bevan M. Baas, "A parallel programmable energy-efficient architecture for computationally-intensive DSP systems," in *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, Nov. 2003.

[6] M. Balch, *Complete Digital Design*, McGraw-Hill, New York, NY, first edition, 2003.

[7] I. Sutherland, "Micropipelines," in *Communications of the ACM*, 1989, vol. 32.

[8] E. Brunvand, "Low latency self-timed flow-through fifos," in *Advanced Research in VLSI*, Mar. 1995, pp. 76–90.

[9] I. Sutherland and S. Fairbanks, "GasP: A minimal FIFO control," in *Advanced Research in Asynchronous Circuits and Systems*, Mar. 2001, pp. 46–53.

[10] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev, "Low-latency asynchronous FIFO buffers," in *Proc. Asynchronous Design Methodologies*, May 1995.

[11] Chris J. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, Inc., 2001.

[12] W. J. Dally and J. W. Poulton, *Digital Systems Engineering*, Cambridge University Press, Cambridge, UK, 1998.

[13] J. F. Wakerly, *Digital Design: Principles and Practices*, Prentice-Hall, third edition, 1999.

[14] M. Hurtado and D. L. Elliot, "Ambiguous behavior of bistable elements," in *Allerton Conf. on Circuit and System Theory*, Oct. 1975, pp. 605–611.

[15] M. Pechoucek, "Anamolous response times of input synchronizers," in *IEEE Journal of Solid-State Circuits*, Feb. 1976, vol. 25, pp. 133–139.

[16] I. Soderquist, "Globally updated mesochronous design style," in *IEEE Journal of Solid-State Circuits*, July 2003, pp. 1242–1249.

[17] R. Ginosar and R. Kol, "Adaptive synchronization," in *IEEE International Conference on Computer Design*, Oct. 1998, pp. 188 –189.

[18] J. U. Horstmann, H. W. Eichel, and R. L. Coates, "Metastability behavior of CMOS ASIC flip-flops in theory and test," in *IEEE Journal of Solid-State Circuits*, Feb. 1989, vol. 24, pp. 146–157.

[19] M. Bolton, "A guided tour of 35 years of metastability research," in *Western Electronic Show and Convention (WESCON)*, Program Session 16, 1987.

[20] T. J. Chaney and C. E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," in *IEEE Transactions on Computers*, Apr. 1973, pp. 421–422.

[21] L. R. Marino, "General theory of metastability," in *IEEE Transactions on Computers*, 1981, pp. 107–115.

[22] J. H. Hohl, W. R. Larsen, and L.C. Schooley, "Prediction of error probabilities for integrated digital synchronizers," in *IEEE Journal of Solid-State Circuits*, Apr. 1984, vol. 19, pp. 236–244.

[23] S. T. Flannagan, "Synchronization reliability in CMOS technology," in *IEEE Journal of Solid-State Circuits*, Aug. 1985, pp. 880–882.

[24] C.L. Portmann and H.Y. Meng, "Metastability in CMOS library elements in reduced supply and technology scaled applications," in *IEEE Journal of Solid-State Circuits*, Jan. 1995, vol. 30, pp. 39–46.

[25] J. Jex and C. Dike, "A fast resolving BiNMOS synchronizer for parallel processor interconnect," in *IEEE Journal of Solid-State Circuits*, Feb. 1995, vol. 30, pp. 133 –139.

[26] C. Dike and E. Burton, "Miller and noise effects in a synchronizing flip-flop," in *IEEE Journal of Solid-State Circuits*, June 1999, pp. 849–855.

[27] Uming Ko and P. T. Balsara, "High-performance energy-efficient D-flip-flop circuits," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Feb. 2000, pp. 94–98.

[28] D. J. Kinniment, A. Bystrov, and A. V. Yakovlev, "Synchronization circuit performance," in *IEEE Journal of Solid-State Circuits*, Feb. 2002, pp. 202–209.

[29] Y. Semiat and R. Ginosar, "Timing measurements of synchronization circuits," in *Advanced Research in Asynchronous Circuits and Systems*, May 2003, pp. 68–77.

[30] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Advanced Research in Asynchronous Circuits and Systems*, May 2003, pp. 89–96.

[31] J. N. Seizovic, "Pipeline synchronization," in *Advanced Research in Asynchronous Circuits and Systems*, Nov. 1994, pp. 87–96.

[32] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T.-P Fang, "Q-modules: internally clocked delay-insensitive modules," in *IEEE Transactions on Computers*, Sept. 1988, vol. 37, pp. 1005–1018.

[33] K. Y. Yun and A. E. Dooply, "Pausible clocking-based heterogeneous systems," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Dec. 1999, pp. 482–488.

[34] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Advanced Research in Asynchronous Circuits and Systems*, Apr. 2000, pp. 52–59.

[35] S. Moore, G. Taylor, R. Mullins, and P. Robinson, "Point to point GALS interconnect," in *Advanced Research in Asynchronous Circuits and Systems*, Apr. 2002, pp. 62–68.

[36] D. S. Bormann and P. Y. K. Cheung, "Asynchronous wrapper for heterogeneous systems," in *IEEE International Conference on Computer Design*, Oct. 1997, pp. 307–314.

[37] C. J. Myers and A. E. Sjogren, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Oct. 2000, pp. 573–583.

[38] J. Kessels, A. Peeters, P. Wielage, and S. Kim, "Clock synchronization through handshake signalling," in *Advanced Research in Asynchronous Circuits and Systems*, Apr. 2002, pp. 59–68.

[39] J. Mekie, S. Chakraborty, and D. K. Sharma, "Evaluation of pausible clocking for interfacing high speed IP cores in GALS framework," in *International Conf. on VLSI Design*, Jan. 2004, pp. 559–564.

[40] R. Dobkin, R. Ginosar, and C. P. Sotiriou, "Data synchronization issues in GALS SoCs," in *Advanced Research in Asynchronous Circuits and Systems*, Apr. 2004, pp. 170–180.

[41] C. E. Molnar, I. W. Jones, W. S. Coates, and J. K. Lexau, "A FIFO ring performance experiment," in *Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997, pp. 279–289.

[42] J. Ebergen, "Squaring the FIFO in GasP," in *Advanced Research in Asynchronous Circuits and Systems*, Mar. 2001, pp. 194–205.

[43] T. Chelcea and S. M. Nowick, "A low-latency FIFO for mixed-clock systems," in *IEEE Computer Society Workshop on VLSI*, Apr. 2000, pp. 119–126.

[44] C. Cummings, "Simulation and synthesis techniques for asynchronous FIFO design," in *Synopsys Users Group*, Oct. 2002.

[45] "NC–Verilog," `http://www.cadence.com/products/functional_ver/nc-verilog/index.aspx`.

[46] "Magic – a VLSI layout system," `http://vlsi.cornell.edu/magic/`.

[47] "HSPICE," `http://www.synopsys.com/products/mixedsignal/hspice/`.

[48] "Digital integrated circuits – the IRSIM corner," `http://bwrc.eecs.berkeley.edu/Classes/IcBook/IRSIM/`.

[49] B. S. Amrutur, *Design and analysis of fast low power SRAMs*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 1999.

[50] L.-S. Kim and R. W. Dutton, "Metastability of CMOS latch/flip-flop," in *IEEE Journal of Solid-State Circuits*, Aug. 1990, pp. 942–951.

[51] F. U. Rosenberger, C. E. Molnar, and R. W. Dutton, "Comments, with reply, on 'metastability of CMOS latch/flip-flop'," in *IEEE Journal of Solid-State Circuits*, Jan. 1992, pp. 128–132.

[52] A. Iyer and D. Marculescu, "Power and performance evaluation of globally asynchronous locally synchronous processors," in *International Symposium on Computer Architecture*, May 2002, pp. 158–168.

[53] C. L. Seitz, *Introduction to VLSI Systems*, chapter 7, "System Timing", C.A. Mead and L.A. Conway, Eds. Addison-Wesley, Reading, MA, 1980.

[54] H.-S. Jung and M.-K. Lee, "Analysis and implementation of interface for heterogeneous system," in *Asia Pacific Conference on ASICs*, Aug. 2000, pp. 21–26.

[55] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems with application to latency-insensitive protocols," in *Design Automation Conference*, June 2001, pp. 147–150.