Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

Summer 9-24-2013

Self-Timed DRAM Data Interface

Rajesh Nerkar

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Digital Circuits Commons](#), and the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Nerkar, Rajesh, "Self-Timed DRAM Data Interface" (2013). *Dissertations and Theses*. Paper 1443.

<https://doi.org/10.15760/etd.1442>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Self-Timed DRAM Data Interface

by

Rajesh Nerkar

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
W. Robert Daasch, Chair
Ivan E. Sutherland
Mark Faust

Portland State University
2013

Abstract

A DRAM communicates with a processing unit via two interfaces: a data interface and a command interface. In today's DRAMs, also known as synchronous DRAMs (SDRAMs), both interfaces use a clock to communicate with the processing unit. The clock times the communication between the processing unit and the SDRAM on both the data interface and the command interface.

We propose a self-timed DRAM. The self-timed DRAM introduces more flexibility into the DRAM interface by eliminating the clock. The command interface and the data interface each communicate with the processing unit using a handshake protocol rather than a clock.

This thesis presents the data interface between the self-timed DRAM and the processing unit. The proposed data interface is self-timed. The self-timed data interface allows the DRAM to deliver data to or accept data from the processing unit as the processing unit demands rather than on a schedule set from the command interface.

The self-timed data interface is designed using GasP circuits and micropipeline circuits. The design is simulated in 180nm CMOs process technology using hspice. This thesis presents the effects of width mismatch on the self-timed data interface. The micropipeline is slightly faster than the GasP. Also, the thesis compares the self-timed DRAM data interface with synchronous DRAM for the data burst rate.

Acknowledgements

I would like to take this opportunity to thank all the wonderful people around me for their guidance and assistance during my graduate study at Portland State University. First and foremost, I would like to thank my research mentor Dr. Ivan E. Sutherland. This research would not have been possible without his constant guidance and motivation. His immense knowledge and research experience in the field of digital circuits not only assisted me in achieving my goal but also improve many aspects of the education like problem solving and writing skills. One of the most important things I learned from him is to THINK. I hope to inculcate the thinking process in every aspect of my life.

I would like to extend my gratitude to Professor Rob Daasch, my thesis advisor, for his valuable inputs in the process of my thesis. I would also like to thank Professor Mark Faust for serving on my M.S. Defense Examination Committee.

I would like to show my immense gratitude to Marly Roncken for her ability and efforts to make everyone around her feel comfortable. I am indebted to my colleagues and friends at Asynchronous Research Center (ARC) and ICDT. Special thanks to my friends Siddharth, Ajit, Prashant, Rahul, Tejas, Prachi and Chaitrali for their encouragement and wonderful times we spent together.

Finally, I owe my deepest gratitude to all the people closest to my heart. I would like to thank my parents and sister for their constant efforts and blessings that encouraged me to pursue my graduate studies.

Contents

Abstract	i
Acknowledgements	ii
List of Figures	v
1 Introduction	1
1.1 Synchronous DRAM Interface	1
1.2 Self-Timed DRAM Interface	5
1.3 Contribution	6
1.4 Thesis Organization	9
2 Background	10
2.1 GasP Circuits	11
2.1.1 Latches	13
2.2 Micropipeline Circuits	15
2.2.1 Muller-C Gate	16
2.2.2 Storage Element	16
2.2.3 Micropipeline Implementation of FIFO	17
2.3 Dual Rail Muller-C Element	19
2.4 Two-Phase Handshake	22
2.5 Two-Phase Handshake Control	25
2.5.1 Two-Phase Handshake Control on Self-Timed DRAM Chip .	26
2.5.2 Two-Phase Handshake Control on Processing Unit Chip . .	30

3	Serial to Parallel (S2P) FIFO	33
3.1	GasP Implementaion of S2P FIFO	34
3.2	Micropipeline Implementaion of S2P FIFO	47
3.2.1	Storage Element for Serial to Parallel FIFO	47
3.2.2	Operation of S2P Micropipeline	49
4	Parallel to Serial (P2S) FIFO	55
4.1	GasP Implementaion of P2S FIFO	56
4.2	Micropipeline Implementaion of P2S FIFO	69
4.2.1	Storage Element for Micropipeline P2S FIFO	69
4.2.2	Operations of P2S Micropipeline	71
5	Results	79
5.1	Read Simulation Waveform	79
5.2	Width Mismatch	80
5.3	Comparison of S2P and P2S FIFO	92
6	Conclusions	97
6.1	Future Work	97
	References	99

List of Figures

1.1	Synchronous DRAM Block Diagram	2
1.2	Execution of Read Commands in SDRAM	4
1.3	Self-Timed DRAM Block Diagram	6
1.4	Self-Timed DRAM Data Interface	8
2.1	GasP Working	12
2.2	Latch	14
2.3	Multiplexer Latch	15
2.4	Muller-C Gate	16
2.5	Storage Element	17
2.6	Micropipeline FIFO	19
2.7	Dual Rail Muller-C Gate	20
2.8	Muller-C true Gate	21
2.9	Muller-C false Gate	22
2.10	Self-Timed Data Interface	23
2.11	Two-phase Handshake Waveform for Read Cycles	24
2.12	Read Logic Block Diagram of Data Interface	26
2.13	Two-Phase Handshake Control on self-Timed DRAM Chip	27
2.14	Schematic of First Stage of the Two-Phase Control Block on the Self-Timed DRAM	28
2.15	Two-phase to GasP Stage	29
2.16	Schematic of the Second Stage of the Two-Phase Control Block on the Self-Timed DRAM	30

2.17	Two-Phase Handshake Control on Processing Unit Chip	31
2.18	Schematic for GasP Merge	32
3.1	Block Diagram of Write Logic Operation of the Self-Timed DRAM	34
3.2	Full Detector for GasP S2P FIFO	35
3.3	Schematic of GasP S2P FIFO	36
3.4	Working Trace of GasP S2P FIFO	38
3.5	Simulation Trace of S2P GasP	40
3.6	Last GasP stage: Serial to Parallel FIFO	41
3.7	First Stage GasP Circuit: Serial to Parallel FIFO	42
3.8	Second Stage GasP Circuit: Serial to Parallel FIFO	43
3.9	Third Stage GasP Circuit: Serial to Parallel FIFO	45
3.10	Fourth Stage GasP Circuit: Serial to Parallel FIFO	46
3.11	Storage Element for Serial to Parallel FIFO	48
3.12	Schematic of S2P Micropipeline	50
3.13	Working Trace of S2P Micropipeline	52
3.14	Simulation Trace of S2P Micropipeline	54
4.1	Block Diagram of Read Logic Circuit on Self-Timed DRAM	56
4.2	Empty Detector for GasP P2S FIFO	57
4.3	Schematic of GasP P2S FIFO	58
4.4	Working Trace of P2S GasP	60
4.5	Simulation Trace of P2S GasP	63
4.6	First stage GasP Circuit: Parallel to Serial FIFO	64
4.7	Second Stage GasP Circuit: Parallel to Serial FIFO	66
4.8	GasP Stage: Parallel to Serial FIFO	67

4.9	<i>DQS</i> signal generator for Parallel to Serial FIFO	68
4.10	Storage Element for Parallel to Serial FIFO	71
4.11	Schematic P2S Micropipeline	73
4.12	Working Trace of P2S Micropipeline	75
4.13	Simulation Trace of P2S Micropipeline	78
5.1	Simulation of Read Operation	80
5.2	Effect of Width Mismatch on Transistor Output Voltage	84
5.3	Block Diagram for Monte-Carlo Simulations of P2S FIFO	85
5.4	Shmoo Plot of GasP implementation of P2S FIFO	86
5.5	Shmoo Plot of Micropipeline implementation of P2S FIFO	87
5.6	Block Diagram for Monte-Carlo Simulations of S2P FIFO	88
5.7	Shmoo Plot of GasP implementation of S2P FIFO	90
5.8	Shmoo Plot of Micropipeline implementation of S2P FIFO	91
5.9	Graph for Data Burst Rate Comparison of Micropipeline and GasP Implementation of S2P FIFO	93
5.10	Graph for Data Burst Rate Comparison of Micropipeline and GasP Implementation of P2S FIFO	94
5.11	Self-Timed DRAM Data Interface	95

Introduction

A DRAM communicates with a processing unit via two interfaces: a data interface and a command interface. In today's DRAMs, also known as synchronous DRAMs, both interfaces use synchronous communication. Synchronous DRAMs communicate with the processing unit with a clock, fed from the processing unit to the synchronous DRAM [1]. The processing unit provides the clock at one of several specified frequencies.

1.1 Synchronous DRAM Interface

Figure 1.1 shows the block diagram of a synchronous DRAM chip with two interfaces: the data interface on the right and the command interface on the left. In the command interface, $CK[t,f]$ is the differential clock signal. Also in the command interface are a multiplexed address and command bus, $ADDR/CMD$. The $ADDR/CMD$ bus tells the synchronous DRAM what operation to perform.

The command interface accepts commands like read, write or refresh from the processing unit. Depending on the command issued, data transfer takes place in burst on the data interface of the synchronous DRAM chip.

The data interface allows data transfer between the synchronous DRAM and the processing unit. The data interface has: a reversible bus for data, $Data[0:7]$, and a differential data-timing signal, $DQS[t,f]$. The data burst is timed by the differential data timing signal, $DQS[t,f]$. The DRAM chip provides the $DQS[t,f]$

signal for sending data to the processing unit, and the processing unit provides the $DQS[t,f]$ signal for sending data to the DRAM chip. The data transfer follows the double data rate (DDR) protocol [2]. The DDR protocol transfers data on both the rising and the falling edges of the $DQS[t,f]$ signal.

Figure 1.1: *Synchronous DRAM Block Diagram: The interface on the left accepts commands from the processing unit. The interface on the right communicates data either way between the synchronous DRAM and the processing unit.*

For write commands, the synchronous DRAM data interface accepts data from the processing unit in serial form. The DRAM array accepts data in parallel form from the data interface. A *serial to parallel FIFO (S2P)* converts the serial data into parallel form. In figure 1.1, the S2P FIFO accepts eight blocks of data each eight bit wide as input and produces 64 bit output. The data burst length of data interface shown is eight.

For read commands, the synchronous DRAM data interface provides data to the processing unit in serial form. The DRAM array provides read data in parallel form. A parallel to serial (P2S) FIFO serializes the parallel data from the DRAM array. The P2S FIFO shown in the figure accepts 64 bit input and produces serial eight blocks of data each eight bit wide, as output. In the synchronous DRAM the P2S and S2P FIFOs are synchronous to the clock.

The frequency of the data-timing signal, $DQS[t,f]$, and the clock, $CK[t,f]$, match. The data-timing signal, $DQS[t,f]$ is synchronized to the input clock signal, $CK[t,f]$. To achieve synchronization between $CK[t,f]$ and $DQS[t,f]$, the synchronous DRAM uses timing circuits such as a delay locked loop (*DLL*) or a phase locked loop (*PLL*) [3]. To compensate for the delay in the wires between the synchronous DRAM and the processing unit, these circuits are relatively complex. As the DRAM array itself is self-timed, a *Delay Line* block is used to mimic the timing of the DRAM array. The *Delay Line* block tells the timing circuit when to start the synchronization.

After receiving a read command and an address, the DRAM array requires time to deliver the data. The time between receiving a read command and delivering the data is called the *latency*. For a synchronous DRAM, latency is specified in clock cycles. The number of clock cycles specified for latency depend on the clock frequency. For example, if a synchronous DRAM has a specified latency of 16 clock cycles for a clock frequency of 600 MHz, then the same synchronous DRAM will have a latency of eight clock cycles for a clock frequency of 300 MHz.

Figure 1.2 shows the execution of non-consecutive read commands of the synchronous DRAM with fixed latency. After issuing the read command and address, shown on the second and third panel, the synchronous DRAM delivers the data, shown in the fifth panel. The synchronous DRAM requires a fixed time to deliver the read data. This figure shows a latency of eight clock cycles, $CL=8$.

Figure 1.2: *Execution of Read Commands of Micron's DDR3 Synchronous DRAM: The figure demonstrates the non consecutive read commands of the Micron's DDR3 synchronous DRAM [1].*

One issue with the synchronous DRAM is that it treats latency as a fixed time. The fixed latency specification must always consider the worst case. The specification for latency of synchronous DRAM must be the time required to fetch the data from a bank farthest from the data interface. The DRAM must deliver data from every bank in the specified number of clock cycles. Even if the data happen to come from a part of the DRAM closer to the data interface, it is impossible for the DRAM to take advantage of the geometric proximity. If the DRAM array could fetch some

data in less time, a potential reduction in average latency is lost. For example the latency for data located in a bank nearer to the data interface might be 5.5 clock cycles, $CL=5.5$ as shown in the figure. The synchronous DRAM cannot deliver the data before eight clock cycles because of the fixed latency specification.

Another issue is the synchronization of the data-timing signal, $DQS[t,f]$, to the input clock signal, $CK[t,f]$. The point at which the synchronization takes place is called the *synchronization point*. The synchronization points are unavoidable in a synchronous DRAM. This is because latency is defined in clock cycles, not in nanoseconds. So, at the very least, the final stage must be synchronized to the clock [3]. The synchronization point adds additional clock cycles to the latency.

1.2 Self-Timed DRAM Interface

The aim of this project is to design a self-timed DRAM. The self-timed DRAM also has two interfaces: a data interface and a command interface. The self-timed DRAM uses a handshake protocol to communicate with the processing unit. Asynchronous design methods eliminate much of the complexity of the timing circuits involved in using the synchronous DRAM. For example, the processing unit will no longer need to count clock cycles to know when data will be delivered. The self-timed DRAM eliminates the need for the clock, the timing circuits such as a delay locked loop (DLL) or a phase locked loop (PLL), and the counter.

Figure 1.3 shows the block diagram of a self-timed DRAM chip. The self-timed DRAM has two interfaces: a command interface on the left and a data interface on the right. The command interface has multiplexed command and address bus, $CMD/ADDR$, and two wires for the handshake protocol, REQ and ACK .

The self-timed DRAM data interface differs from the data interface of the synchronous DRAM by the addition of four handshake wires, *Send Data (SD)*, *Data Available (DA)*, *Take Data (TD)*, and *Space Available (SA)*. The actual data transfer occurs in a burst identical to the synchronous DRAM timed by a differential timing signal, $DQS[t,f]$. The handshake control wires at the data interface control a burst of data transfer.

Figure 1.3: *Self-Timed DRAM Block Diagram: The interface on the left accepts commands from the processing unit. The interface on the right communicates data both ways between the self-timed DRAM and the processing unit. The self-timed DRAM command interface replaces the clock with the handshake wires.*

1.3 Contribution

My thesis concentrates on the data interface between the self-timed DRAM and the processing unit. The proposed data interface is self-timed in nature. The self-timed data interface allows the self-timed DRAM to deliver or accept data

from the processing unit as the processing unit demands rather than on a schedule set by a clock signal. This eliminates the need for clock, and timing circuits for synchronization of DQS signal and the clock at the data interface.

My thesis ignores the command interface between the self-timed DRAM and the processing unit. The form of more complex commands made possible by the self-times interfaces, and how the self-timed DRAM could use them, is outside the scope of this thesis.

Figure 1.4 shows the self-timed DRAM data interface. The self-timed DRAM data interface has: a reversible bus for data, $Data[0:7]$, and the data-timing signal, $DQS[t,f]$. The self-timed DRAM data interface also has two handshake wires for write, *Take Data (TD)*, and *Space Available (SA)*, and two handshake wires for read, *Send Data (SD)*, and *Data Available (DA)*. The self-timed DRAM data interface uses the DDR protocol for data transfer.

The synchronous DRAM data interface uses a delay locked loop *DLL* for synchronization with the clock, the self-timed DRAM data interface replaces the *DLL* with the *Two Phase Control Bock* as shown in figure 1.4. Unlike the synchronous DRAM data interface, the P2S and S2P FIFOs of the self-timed DRAM data interface are fully self-timed. The read FIFO and write FIFO buffer data from the DRAM array for multiple read and write commands, respectively. The read and write FIFOs are also self-timed.

Figure 1.4: *Self-Timed DRAM Data Interface: The self-timed DRAM data interface has reversible bus, $Data[0:7]$, $DQS[t,f]$, and four handshake wires.*

The self-timed DRAM data interface uses a handshake protocol to communicate between the DRAM and the processing unit. The handshake wires for read, *Send Data (SD)* and *Data Available (DA)*, and for write, *Take Data (TD)* and *Space Available (SA)*, use the same handshake protocol. The processing unit controls the handshake wires *SD* and *TD*. The self-timed DRAM controls the handshake wires *DA* and *SA*.

Because the data and the data timing bus are reversible, data transfer for read and write must be mutually exclusive. The direction of the reversible bus is set by the processing unit. The self-timed DRAM tells the processing unit when read data from DRAM array are available for delivery. The processing unit responds when it can accept read data. The self-timed DRAM also tells the processing unit

when the self-timed DRAM can accept data to write in the DRAM array. The processing unit responds when it can send data to write.

My thesis compares the self-timed data interface with the synchronous data interface for the data burst rate. The P2S FIFO, the S2P FIFO, and the two phase control block are designed using self-timed circuits. To explore data burst rate of the self-timed data interface, I implemented two designs of the P2S and the S2P FIFO. This thesis also shows the effect of width mismatch on the two implementations of P2S and S2P FIFO.

1.4 Thesis Organization

This thesis is organized in six chapters. The second chapter contains background about asynchronous communication and circuits. Two different implementations of the serial to parallel (S2P) FIFO and the parallel to serial (P2S) FIFO are explained in chapters three and four, respectively. The fifth chapter explains the simulation results. Conclusions about the design are in the sixth chapter.

Background

The data interface for the self-timed DRAM communicates data between the self-timed DRAM and the processing unit. To reduce the number pins at the data interface, the DRAM delivers or accepts bursts of data at the data interface. The blocks of a data burst are timed by a differential data timing signal. The data transfer follows the double data rate (DDR) protocol.

This thesis propose to use a two-phase handshake protocol to control the start of each data burst. The two-phase handshake protocol initiates the data burst with a differential data timing signal at the data interface.

As shown in Figure 1.4 of "Chapter-1", a DRAM array accepts or provides 64-bit data word. To deliver the read data, the self-timed DRAM converts the 64-bit data word from the DRAM array into a data burst of 8 blocks each 8 bits wide using a parallel to serial FIFO (P2S). The P2S FIFO also the generates the differential data timing signal to time the blocks of the data burst.

Similarly, to write data to the DRAM array, the self-timed DRAM converts the data burst of 8 blocks each 8 bits wide from the processing unit into a 64-bit data word using a serial to parallel FIFO (S2P). This thesis reports two different implementations of the S2P and P2S FIFOs using asynchronous circuits. One implementation using GasP circuits [4] and the second using micropipeline circuits [5].

This chapter explains the two-phase handshake protocol. This chapter also explains: the GasP circuits, the micropipeline circuits, the latches and the dual rail Muller-C gate. These circuits are used to implement the P2S and the S2P FIFO and the two-phase handshake protocol.

2.1 GasP Circuits

GasP circuits are used as control blocks for the data FIFO, the two-phase handshake protocol, and for implementing the parallel to serial (P2S) and the serial to parallel (S2P) FIFO in the design [4]. A GasP circuit plus the latches it controls forms a stage of the data FIFO. A stage of the data FIFO is also referred as a GasP stage.

Figure 2.1 shows schematic of two GasP stages with the working trace. The GasP circuit uses one wire for communication with its predecessor stage, this wire is called the *predecessor state wire*. The GasP circuit uses one wire for communication with its successor stage, this wire is called the *successor state wire*.

The GasP circuit in figure 2.1 has three pins: two bidirectional pins for successor and predecessor state wires, *pred* and *succ* and one output pin, *fire*. The GasP circuit produces a high at its *fire* pin, when its predecessor state wire, *pred* is in the high state, indicating predecessor stage is FULL and when its successor state wire, *succ* is in the low state, indicating that the successor stage is EMPTY.

Property of Predecessor and Successor wire	State of Predecessor and Successor wire
FULL	HIGH
EMPTY	LOW

Table 2.1: *Property of Predecessor (pred) and Successor (succ) wire*

Figure 2.1: *GasP Working: The GasP stage produces a fire pulse when its predecessor state wire is HIGH meaning FULL and successor state wire is LOW meaning EMPTY.*

The *fire* signal of the GasP stage is a self-defeating signal meaning the *fire* signal of the GasP stage a pulse. The *fire* pulse does three operations: the fire pulse drives the predecessor wire, *pred* to low state through an NMOS transistor; the

fire pulse drives the successor wire, *succ* to high state through the inverter and PMOS transistor, and the fire pulse makes the latches momentarily transparent.

2.1.1 Latches

This section explains the latches used in the GasP implementation of the data FIFO, the P2S and the S2P FIFO. The first latch is the single-input latch and the second latch is a multiplexer latch.

Figure 2.2 shows the schematic of the single-input latch with the symbolic representation in the top right corner. The latch has a data input, *in*, a control input, *hcl* and a data output, *out*. The two NMOS transistors act as switches. When the *hcl* signal goes high the two NMOS transistors are ON and the latch is transparent that is the input, *in* is copied to the output, *out*. The inverter labeled *Wk* represents weak inverter. The loop of two weak inverters forms the weak keeper for the latch. Both the inverters are weak because the two NMOS transistors drive the weak inverters. An inverter at the output acts as a driver to provide enough drive current at the output of the latch to drive the latches of the next stage.

Figure 2.2: *Latch: The latch is transparent when hcl is high.*

Figure 2.3 shows the schematic of a two input latch. The two input latch acts as 2:1 multiplexer. The latch has two data inputs, *in1* and *in2*, two control inputs, *hcl1* and *hcl2*, and a data output, *out*. The latch has two identical conventional latch circuits with a common weak keeper and common output driver. The control input, *hcl1* copies the data input, *in1* to the latch output, *out* and the control input, *hcl2* copies the input, *in2* to the latch output, *out*. The control signals, *hcl1* and *hcl2* must be mutually exclusive. An inverter at the output acts as a driver to provide enough drive current at the output of the latch to drive the latches of the next stage.

Figure 2.3: *Multiplexer Latch: The latch is transparent to in1 when hcl1 is high. The latch is transparent to in2 when hcl2 is high. The control signals hcl1 and hcl2 are mutually exclusive.*

2.2 Micropipeline Circuits

My second implementation of the P2S and S2P FIFOs uses micropipeline circuits. The micropipeline circuits follow DDR protocol. The micropipeline implementation of a FIFO has a control path and a data path [5]. The control path uses Muller-C gates, and the data path uses storage elements. A micropipeline uses two wires for asynchronous communication. This section explains the Muller-C gate, storage elements and the micropipeline implementation of FIFO.

2.2.1 Muller-C Gate

The two elements of micropipeline are the Muller-C gate and the storage element. Figure 2.4 shows the schematic of the Muller-C gate with the state table and the symbolic representation. The Muller-C gate has an inverter at one of the inputs denoted by b . The inverter is indicated by a bubble at the input of the Muller-C gate. The state table shows the Muller-C gate will hold the previous state of output pin, c , if the state of input pins, a and b match. If the states of the inputs, a and b differ then the Muller-C gate will copy the input, a to the output, c .

Figure 2.4: *Muller-C Gate: If the state of the inputs, a and b , differs, this Muller-C gate will copy the input, a , to the output, c . Else Muller-C gate holds the previous state of the output, c . A keeper is required to hold the state of the output, c . The keeper is omitted from the figure.*

2.2.2 Storage Element

Figure 2.5 shows the transistor implementation of the storage element with the state table and the symbolic representation. The transistor implementation makes

use of both the true and the false form of its control signal, $C[t,f]$. The inverter with a switch shown on top right corner in figure is an icon for this storage element. The icon is drawn assuming the control signal, $C[t,f]$ is false. When the control signal, $C[t,f]$ is false, the storage element copies input, b inverted to output, z . When the control signal, $C[t,f]$ is true, the storage element copies input, a inverted to output, z . The storage element responds to the rising and the falling edge of the control signal, $C[t,f]$.

Figure 2.5: *Storage Element: The storage element copies the input a to the output, z when its control input C is in the high state, and it copies the input b to, the output z , when the control input, C , is in the low state. In the state table a' and b' signify inversion.*

2.2.3 Micropipeline Implementation of FIFO

Figure 2.6 shows a section of a micropipeline FIFO with two stages denoted by blue boxes. The FIFO shown is one bit wide and two bits long. The FIFO can

be made wider with more storage elements in a stage. The FIFO can be made longer by extending the stages to the right. In this figure dashed lines carry control signals, and solid line carry data values. The switches are drawn as they would be for an empty FIFO. When the FIFO is empty as shown in the figure, the data path is transparent: one can trace a direct path from left to right. The FIFO is empty when the state of $R1$ and $R2$ match. The FIFO is full when the state of $R1$ and $R2$ differ. In the figure there are two data paths, INA and INB . The first stage, $Stage1$ forwards inversion of INA to the second stage, $Stage2$ when $R1$ is false. The first stage, $Stage1$ forwards inversion of INB to the second stage, $Stage2$ when $R1$ is true. The micropipeline FIFO forwards data on both the rising edge and falling edge of controlling signal, hence the micropipeline follows the DDR protocol.

Figure 2.6: *Micropipeline FIFO: The FIFO shown is one bit wide and can store two bits. The FIFO can be made wider with more storage elements in a stage. The FIFO can be made longer by extending the stages to the right.*

2.3 Dual Rail Muller-C Element

The micropipeline implementations of the P2S and S2P FIFO use a dual rail Muller-C gate for the control path. The output of the dual rail Muller-C element matches the DDR protocol. The dual rail Muller-C gate accepts the inputs in true and false form. It also produces its output in true and false form. The dual rail Muller-C gate has two single rail Muller-C gates to accept the true and the false values of the inputs, and produce the true and the false values of the output.

Figure 2.7 shows the dual rail Muller-C gate used in the P2S FIFO. The dual rail Muller-C gate has two single rail Muller-C gates, CT and CF , to accept the inputs in true and false form and produces the output in true and false form respectively. The black lines with labels represent single wires. The green lines represent more than one wire, in this case two wires. A symbolic representation of the dual rail Muller-C gate appears in top right corner of the figure.

Figure 2.7: *Dual Rail Muller-C Gate: The dual rail Muller-C gate accepts inputs in true and false form, $ack[t,f]$ and $req[t,f]$, and produces output in true and false form, $out[t,f]$. Two inverters at the output form a keeper to hold the state of the output, $out[t,f]$ when the inputs differ.*

Each of the inputs and the output of the dual rail Muller-C gate has two wires, carrying the true and false values, $ack[t,f]$, $req[t,f]$, and $out[t,f]$ respectively. The master-clear input, mc initializes the output, $out[t,f]$ of the dual rail Muller-C gate to false, where $out[t]$ is in the low state and $out[f]$ is in the high state.

Figure 2.8 shows the single rail Muller-C gate, *CT*. This Muller-C gate accepts the true value of the inputs and produce the true value of the output. This Muller-C gate has $req[t]$ and $ack[t]$ as inputs and $out[t]$ as output. This Muller-C gate copies $req[t]$ to $out[t]$, if $req[t]$ and $ack[t]$ differ in state. If the state of $req[t]$ and $ack[t]$ match, it holds the previous state of $out[t]$. The mc input initializes the Muller-C gate to false. When the mc input is high it pulls $out[t]$ low through an NMOS transistor, $n1$.

Figure 2.8: *Muller-C[t] Gate: The Muller-C[t] gate accepts inputs in true form, $ack[t]$ and $req[t]$, and produces output in true form, $out[t]$.*

Figure 2.9 shows the Muller-C gate, *CF*. This Muller-C gate accepts the false form of the inputs and produce the false value of the output. This Muller-C gate has $req[f]$ and $ack[f]$ as inputs and $out[f]$ as output. This Muller-C gate copies $req[f]$ to $out[f]$, if $req[f]$ and $ack[f]$ differ in state. If the state of $req[f]$ and $ack[f]$ match, it holds the previous state of $out[f]$. The master-clear mc input initializes the

Muller-C gate to false. When the mc input is in the high state it pulls $out[f]$ high through a PMOS transistor, $p1$.

Figure 2.9: *Muller-C[f] Gate: This Muller-C[f] gate accepts inputs in false form, $ack[f]$ and $req[f]$, and produces output in false form, $out[f]$.*

2.4 Two-Phase Handshake

A two-phase handshake protocol allows data transfer between the self-timed DRAM and the processing unit without use of a clock signal. Data transfer between the self-timed DRAM and the processing unit takes place in burst of data. The two phase handshake controls a burst of data. The blocks of the data burst are timed by a differential timing signal, $DQS[t,f]$.

Figure 2.10 shows the data interface between the self-timed DRAM and the processing unit. The data interface has: a reversible bus for data, $Data[0:7]$ and differential timing signal, $DQS[t,f]$. The data interface also has: two wires for a

two-phase handshake for read, *Data Available (DA)* and *Take Data (TD)*, and two wires for a two-phase handshake for write, *Take Data (TD)* and *Space Available (SA)*.

Figure 2.10: *Self-Timed Data Interface: Self-Timed Data Interface allows communication between the self-timed DRAM and the processing unit without the use of a clock signal.*

The two-phase handshake transfers a data burst on every transitions of the request and the acknowledge wire. The two-phase handshake protocol is a non-return to zero (NRZ) handshake protocol. A NRZ handshake protocol transfers information on both the low and the high state of the request and the acknowledge wires. There is no neutral state.

Figure 2.11 shows the two-phase handshake communication between the self-timed DRAM and the processing unit for a read operation. The waveform shows continuous bursts of data. The self-timed DRAM and the processing unit control the green and blue color traces in the waveform, respectively. The black color arrows are the causality arrows.

Figure 2.11: *Two-phase Handshake Waveform for Read Cycles: When the state of the DA and SD wires match and the self-timed DRAM has data, then the self-timed DRAM makes a transition on the DA wire. When the processing unit has space for a burst of data, then it makes a transition on the SD wire. The self-timed DRAM drives the data bus, Data[0:7] when the state of both the SD and DA wires match.*

The two-phase handshake wires for read are *Data Available* (*DA*) and *Send Data* (*SD*). When state of the *DA* and *SD* wires match and the self-timed DRAM has data, then the self-timed DRAM makes a transition on the *DA* wire indicating that it could send a burst of data. When the processing unit has space for a burst of data, then it makes a transition on the *SD* wire. When the state of both the *SD* and *DA* wires match, the self-timed DRAM drives the data bus, *Data[0:7]*. For this explanation, the *DA* wire acts as request wire and *SD* wire acts as acknowledge

wire. One data burst transfer takes place on every transition of both the *SD* and *DA* wires.

The processing unit can also request data before the data has been fetched by the self-timed DRAM. In this case the *SD* wire acts as a request wire and *DA* wire acts as a acknowledge wire. If the state of DA and SD wires match and the processing unit has space for data, then the processing unit makes a transition on the SD wire. When the self-timed DRAM has data, then it makes a transition on the DA wire and drives the data bus, Data[0:7] to send a burst of data.

2.5 Two-Phase Handshake Control

This section explains the two-phase handshake control block for read and write operation. The two-phase handshake control blocks for read and write use different circuits. For a read operation the DRAM chip acts as a sender and for a write operation the DRAM chip acts as a receiver. To explain the circuits for two-phase handshake control block I choose the read data interface. The two-phase handshake control block for write on the self-timed DRAM chip is same as the two-phase handshake control block for read operation on the processing unit chip. For a read operation the processing unit acts as a receiver.

Figure 2.12 shows the read logic block diagram of the data interface between the self-timed DRAM and the processing unit. The read logic on the self-timed DRAM chip has a read FIFO, a two-phase control block and a P2S FIFO. The read logic on the processing unit chip has a S2P FIFO, a two-phase control block and a read FIFO.

Figure 2.12: *Read Logic Block Diagram of Data Interface*

2.5.1 Two-Phase Handshake Control on Self-Timed DRAM Chip

The two phase control block on the self-timed DRAM does two operation: first it makes a transition on the *DA* wire if data are available for delivery, and second it forwards data to the P2S FIFO when the state of *DA* and *SD* wires match. The two-phase handshake control is implemented using GasP circuits.

Figure 2.13 shows the two-phase handshake control. The two-phase control block has two GasP stages. The two-phase control block controls the data flow to the P2S FIFO depending on the state of *DA* and *SD* wires.

Figure 2.13: *Two-Phase Handshake Control on self-Timed DRAM Chip.*

Figure 2.14 shows first stage of the two-phase control block on the self-timed DRAM. The first GasP stage is basically a GasP to two-phase block. The GasP to two-phase stage makes transition to the *DA* wire when data are available from the GasP FIFO. The GasP to two-phase block is a special GasP module with two successor state wires, *ACK* and *DA*, one predecessor state wire, *pred* and a fire signal wire, *fire*. The GasP to two-phase stage fires when the *pred* wire is in the high state, and the state of *ACK* and *DA* wires differ. The fire pulse from this stage does three operations: it makes the latch momentarily transparent, it drives the *pred* wire to low state and it inverts the state of *DA* wire through the inverter and latch. The inversion of the *DA* wire makes a transition on the *DA* wire. After making the transition to the *DA* wire the self-timed DRAM waits for a transition on the *SD* wire, to start the data transfer.

Figure 2.14: *Schematic of First Stage of the Two-Phase Control Block on the Self-Timed DRAM: The first GasP stage is basically a GasP to two-phase block. The GasP to two-phase module makes transition on the DA wire when data are available from the GasP FIFO.*

Before explaining the second stage of the two-phase handshake control block on self-timed DRAM, I will explain the two-phase to GasP block because the second stage is similar to two-phase to GasP block. Figure 2.15 shows a two-phase to GasP stage. The two-phase to GasP is a special GasP module with two predecessor state wires *SD* and *DA*, one successor state wire, *succ* and a fire signal wire, *fire*. The two-phase to GasP stage makes a transition on the *SD* wire only if the state of *DA* and *SD* wire differ, and the processing unit has free space, meaning the *succ* wire is in the low state. The two-phase to GasP block is also present on the processing unit chip for two-phase handshake control for read operation.

Figure 2.15: *Two-phase to GasP Stage*

Figure 2.16 shows the second stage of the two-phase control block on the self-timed DRAM. The second GasP stage is basically a two-phase to GasP stage with an extra pin. After making a transition to the *DA* wire the DRAM waits for a transition on the *SD* wire to start the data transfer. The two-phase to GasP stage ensures this functionality. This module forwards the read data to the P2S FIFO if the state of *DA* and *SD* wires match. This module has five pins: three pins for predecessor state wires, *ACK*, *DA*, and *SD*, one pin for successor state wire, *succ* and one pin for fire signal, *fire*. This module fires when the state of *ACK*, *DA*, and *SD* wires match, and the successor state wire *succ* is in the low state. The fire pulse does three operations: it makes the latch momentarily transparent, it pulls the *succ* wire to a high state and it copies the state of the *DA* wire to the *ACK* wire through the latch.

Figure 2.16: *Schematic of the Second Stage of the Two-Phase Control Block on the Self-Timed DRAM: The second GasP stage is basically a two-phase to GasP stage with an extra pin. The two-phase to GasP module forwards data to P2S FIFO when the state of DA, and SD wires match.*

2.5.2 Two-Phase Handshake Control on Processing Unit Chip

Figure 2.17 shows the two-phase handshake control block on the processing unit chip. The two-phase handshake control has: two GasP stages, a GasP merge stage and a two-phase to GasP stage. The two-phase to GasP stage is shown in figure 2.15.

The two-phase handshake control has two independent paths: the two-phase control path, *pred2* and the data path, *pred1*. The data path is slower than the two-phase control path, because the S2P FIFO takes more time to complete the conversion than the two-phase to GasP block. The GasP merge stage ensures latching of data for the corresponding transition on the *SD* wire. The GasP stage on the data path and control path ensures two bursts data.

Figure 2.17: *Two-Phase Handshake Control on Processing Unit Chip.*

Figure 2.18 shows the schematic of GasP merge block. The GasP merge stage fires when both the predecessor state wires, *pred1* and *pred2* are in the high state and successor state wire, *succ* is in the low state. The fire pulse of this stage does three operations: it drives the *pred1* and *pred2* wires to low state; it drives *succ* wire to high state; and it makes the latches momentarily transparent.

Figure 2.18: *Schematic for GasP Merge: The GasP merge stage ensures latching of 64-bit data for the corresponding transition on the SD wire.*

Serial to Parallel (S2P) FIFO

The processing unit provides the data to write to the DRAM in a serial burst to reduce the number of pins at the data interface. The self-timed DRAM data interface must convert the serial data burst into parallel bits which the DRAM array accepts as input. To convert the serial data burst into parallel bits, the self-timed DRAM data interface requires a serial to parallel FIFO (S2P).

The serial data burst fills the stages of the S2P FIFO sequentially. The number of stages of S2P FIFO depend on the number of blocks in the serial data burst. The S2P FIFO will not produce a parallel output until all the stages of FIFO are full, meaning all the blocks of the serial data burst are shifted into the S2P FIFO. To produce a parallel output, the S2P FIFO uses a full detector.

To explore conversion rate, I simulated two designs of S2P FIFO using self-timed circuits: one uses GasP circuits and other uses micropipeline circuits. This chapter explains the two implementations of S2P FIFO.

Figure 3.1 shows the write logic block of the self-timed DRAM. The write logic block of the self-timed DRAM accepts serial data from the self-timed DRAM and provides parallel form of the data as output. Inputs to the write logic block are eight bit wide data bus, $Data [0:7]$, from the self-timed DRAM, the data-timing signal, $DQS[t,f]$, and the two-phase handshake control wire, *Space Available (SA)*, from the processing unit. Outputs of this block are the 64-bit wide data,

$Data_{ap}[0:63]$, and the two-phase handshake control wire, *Take Data (TD)*.

Figure 3.1: *Block Diagram of Write Logic Operation of the Self-Timed DRAM:* The write logic block of the self-timed DRAM accepts a serial data burst of eight blocks long each eight bit wide from the self-timed DRAM and produces a 64-bit word as output.

3.1 GasP Implementaion of S2P FIFO

This section explains the GasP implementation of the S2P FIFO [4]. The GasP implementation of the S2P FIFO is eight bit wide and eight stages long. For simplicity of explanation this section explains a one bit wide and four stages long S2P FIFO for clarity.

The S2P FIFO produces a parallel output only after all the stages of FIFO are full. To produce a parallel output, the S2P FIFO uses a full detector. Figure 3.2 shows the circuit for full detection used in the S2P FIFO. The full detector circuit

shown detects the fullness of four GasP stages. The full detector has four stages denoted by the black rectangles. Each stage of the full detector circuit is embedded inside the GasP stage of the S2P FIFO. Each stage of the full detector is an AND function of the state wires except the leftmost stage because the output of the full detector is an active low signal, $full(low)$. Output of the full detector is low when all the inputs, $pred1$, $pred2$, $pred3$, $pred4$ and $pred5$ are in the high state. The full detection takes place from right to left because as the first serial data block shifts in the S2P FIFO, it fills the last stage, then the second serial data blocks fills the second last stage and so on. The last stage gets full first, and the first stage gets full last.

Figure 3.2: *Full Detector for GasP S2P FIFO*

Figure 3.3 shows the block diagram of the GasP implementation of S2P FIFO. The S2P FIFO has five GasP stages. Inputs to the S2P GasP FIFO are the one bit wide data input, Din , and the data-timing signal, $DQS[t,f]$. Output from the S2P GasP FIFO is the four bit data, $Dout[0:3]$. The S2P FIFO has a bidirectional pin for a successor state wire, $succ$. The $succ$ wire informs the next GasP stage when the current operation of the S2P FIFO completes.

Figure 3.3: Schematic of GasP S2P FIFO: The S2P FIFO shown can be made wider with more latches on each stage. The S2P FIFO can be made longer by adding GasP stages to the right stage3.

The input, $DQS[t,f]$ which is the data-timing signal, fills the first four stages of the S2P FIFO with the serial data. When all the four stages of the S2P FIFO are full, the four state wires $sw[1]$, $sw[2]$, $sw[3]$, and $sw[4]$ are all in the high state. At that time the last stage of the S2P FIFO captures the bits from all the four stages and delivers the parallel four bit wide output. The last stage controls a wider latch, shown larger in the block diagram, to capture the four bit wide data.

The last GasP stage of the S2P FIFO produces a fire pulse when two things are true. First, all the stages to the left of the last stage in the S2P FIFO are full indicating that they hold data. Second, its *succ* wire is in the low state which indicates its successor stage is empty. The fire pulse from this stage does three operation: it latches the data from all the stages which are $data[0:3]$; it pulls the successor state wire high to inform the next GasP stage that the S2P FIFO's current operation is complete; and it makes all the stages of the S2P FIFO empty by pulling the state wires $sw[1]$, $sw[2]$, $sw[3]$, and $sw[4]$ to a low state.

Figure 3.4 shows the working trace of the GasP S2P FIFO. In the figure all the red color traces represent fire signals, the green color traces represent state wires, the black color traces represent data wires, and the orange and the purple traces represent the true and false values of $DQS[t,f]$ signal. The yellow and brown color arrows are causality arrows. In the figure more than one causality arrow pointing to a trace indicates an AND function of the traces from which these causality arrows originate.

The GasP S2P FIFO starts its action with the $DQS[t,f]$ pulse, as shown in panel1. The *fire1* pulse happens on every transition of the $DQS[t,f]$ signal as shown by

the yellow causality arrow. The $DQS[t,f]$ fills the S2P FIFO with serial data, $Din[0]$, $Din[1]$, $Din[2]$ and $Din[3]$. The $fire1$ pulse happens four times for one S2P conversion as shown in panel3.

Figure 3.4: Working Trace of GasP S2P FIFO

The $fire2$ pulse happens thrice as shown in panel5. The $fire2$ pulse forwards $Din[0]$, $Din[1]$, and $Din[2]$. The $fire3$ pulse happens twice as shown in panel7. The $fire3$ pulse forwards $Din[0]$, and $Din[1]$. The $fire4$ pulse happens once as shown in panel5. The $fire4$ pulse forwards $Din[0]$.

The *fire5* pulse can happen only when the S2P FIFO is full. The AND function for full detection is embedded in the GasP stages of S2P FIFO. The output of AND function is an active low signal, *allfull(low)*. The *allfull(low)* signal appears as the blue trace in panel9.

The *fire5* pulse happens when the *allfull(low)* signal is in the low state indicating the S2P FIFO is full, *sw[1]* is in the high state and *succ* is in the low state. This is shown by the yellow causality arrow coming on the rising edge of *fire5* signal.

The *fire5* signal does several things. First, *fire5* copies *Din[0]*, *Din[1]*, *Din[2]* and *Din[3]* from all the four stages to *Dout[0:3]*. Second, the *fire5* signal makes *sw[1]*, *sw[2]*, *sw[3]* and *sw[4]* low declaring the S2P empty to start the next operation. The yellow causality arrow from the rising edge of the *fire5* signal to the falling edges of *sw[1]*, *sw[2]*, *sw[3]* and *sw[4]* shows this operation.

Figure 3.5 shows the trace from the simulation of GasP S2P FIFO. The GasP S2P FIFO simulated is one bit wide and four stages long. The color of traces of the simulation are similar to the waveform trace shown in figure 3.4. The figure omits the traces for the data values.

In the figure 3.5 all the simulation traces match the working trace shown in figure 3.4. In the simulation trace width of the *DQS[t,f]* signal pulse is 0.8ns, and width of the *fire* signal pulses shown as red traces is 0.4ns.

Figure 3.5: *Simulation Trace of GasP S2P FIFO*

Figure 3.6 shows the GasP circuit of the last stage in the S2P FIFO. The last GasP stage has two input pins, *in1* and *in2*, two bidirectional pins, *succ* and *pred*, and one output pin, *fire*. The last stage produces a fire pulse when the inputs *in1*, *in2*, and *succ* are in the low state. The low state of input, *in1* indicates that all the stages in the S2P FIFO are full, except the first stage. The low state of input, *in2* indicates that the first stage of the S2P FIFO is full. The data in the S2P FIFO move from left to right, the first stage becomes full last. To start the next operation with minimum delay, the state of the first stage, *in2* is provided as input to last GasP stage of the S2P FIFO.

Figure 3.6: *Last GasP stage for Serial to Parallel FIFO: The last stage stage fires when its succ wire and the inputs, in1 and in2, are in the low state. The low state of the inputs, in1 and in2, indicates all stages of the FIFO are full.*

The fire pulse from last stage does three things: it latches the four bit data, it pulls the *succ* wire high through the successor driver, *SD*, and it pulls the *pred* wire low through the predecessor driver, *PD*.

Figure 3.7 shows the GasP circuit of the first stage in the S2P FIFO. The first stage of the S2P FIFO converts the double data rate (DDR) protocol to a single wire GasP protocol. The first GasP stage latches data on the rising and the falling edge of the data timing signal, $DQS[t,f]$.

Figure 3.7: *First Stage GasP Circuit of Serial to Parallel FIFO: The two latches at the left hold the previous state of $DQS[t,f]$ on the wires called a and b . The first stage can fire when new state of $DQS[t,f]$ differs from the previous state, a and b .*

The first stage GasP circuit has $DQS[t,f]$ as input pins, $fire$ as output pin and $succ$ as bidirectional pin. The two latches at the left hold the previous state of $DQS[t,f]$ on the wires called a and b . The first stage produces a fire pulse: when present state of $DQS[t,f]$ differs from the previous state, a and b , indicating there is a transition on the inputs $DQS[t]$ or $DQS[f]$; and the $succ$ wire is in low state indicating there is space. The three-input bubbled ANDs, NOR gate and the two latches evaluate the logic for the $fire$ signal. The fire pulse from first stage does three operations: it makes the data latch transparent momentarily, it drives the

succ wire high through the successor driver, *SD* and it copies the previous state of *DQS[t,f]* signals into the latches labeled *latch*.

Figure 3.8 shows the GasP circuit of the second stage in the S2P FIFO. The second GasP circuit has two bidirectional pins, *succ* and *pred*, two output pins, *fire* and *aout*, and one input pin, *fire5*. The second stage produces a fire pulse when the *pred* wire is in the high state and the *succ* wire is in the low state. The fire signal from last stage, *fire5* or the fire signal from this stage, *fire* can pull the predecessor wire of this stage low through an *OR* predecessor driver, *PD*. The second stage also forwards the state of the first stage as output *aout* to the last stage of the *S2P* FIFO.

Figure 3.8: *Second Stage GasP Circuit of the Serial to Parallel FIFO: Output aout indicates the state of first stage of the S2P FIFO.*

Figure 3.9 shows the GasP circuit of the third stage in the S2P FIFO. The third GasP circuit has two bidirectional pins, *succ* and *pred*, two output pins, *fire* and *aout*, and two input pins, *fire5* and *ain*. The third stage produces a fire pulse when the *pred* wire is in the high state and *succ* wire is in the low state. The fire signal from this stage does three operations: it pulls the *pred* wire low; it pulls the *succ* wire high; and it makes the latch momentarily transparent. Either the fire signal from last stage, *fire5* or the fire signal from the third stage, *fire* can pull the predecessor wire, *pred* of the third stage low through an *OR* predecessor driver, *PD*. The NAND gate is a stage of the full detector embedded in this stage. The inputs to the NAND gate are, *ain* and *pred*, which is the predecessor state wire of this stage. The high state of input, *ain* indicates that the stages to the right including the third stage are full. The output, *aout* of the NAND gate forwards the state of all the stages to its right including the third stage and its predecessor stage to the last stage of the S2P FIFO.

Figure 3.9: *Third Stage GasP Circuit of the Serial to Parallel FIFO: Input ain indicates that all the stages to the right including the third stage are full. Output, $aout$, of the NAND gate forwards the state of all the stages to its right including the third stage and its predecessor stage to the last stage of the S2P FIFO.*

Figure 3.10 shows the GasP circuit of the fourth stage in the S2P FIFO. The fourth GasP circuit has two bidirectional pins, $succ$ and $pred$, two output pins, $fire$ and $aout$, and two input pins, $fire5$ and ain . The fourth stage produces a fire pulse when the predecessor wire, $pred$ is in the high state, and its successor wire, $succ$ is in the low state. The fire pulse from this stage does three operations: it pulls the $pred$ wire low; it pulls the $succ$ wire high; and it makes the latches momentarily transparent. Either the fire signal from last stage, $fire5$, or the fire signal from the fourth stage, $fire$ can pull the predecessor wire, $pred$, of this stage low through the *OR* predecessor driver, *PD*. The NAND gate followed by an inverter is a stage

of the full detector embedded in this stage. The inputs to the AND gate are ain and $pred$, which is the predecessor state wire of this stage. The high state of the input, ain indicates that the stages to the right including the fourth stage are full. The output $aout$ of the NAND gate forwards the state of all the stages to its right including the fourth stage and its predecessor stage as an input to the predecessor state.

Figure 3.10: *Fourth Stage GasP Circuit of the Serial to Parallel FIFO: High state of the input, ain , indicates that all the stages to the right including this stage are full. The output, $aout$ of the AND gate forwards the state of all the stages to its right including the fourth stage and its predecessor stage as an input to the predecessor state.*

3.2 Micropipeline Implementaion of S2P FIFO

The Micropipeline implementation of the S2P FIFO has a control path and a data path. The control path uses the dual rail Muller-C gates explained in "Chapter-2", and the data path uses storage elements. This section explains the micropipeline implementation of the S2P FIFO [5].

3.2.1 Storage Element for Serial to Parallel FIFO

Figure 3.11 shows the schematic of the storage element for the data path. Inputs to the storage element can be divided into two parts: the data inputs and the control inputs. The data inputs are a , k_a , b , and k_b . The control inputs are $req[t,f]$. The $req[t,f]$ is true when the $req[t]$ is high and the $req[f]$ is false when the $req[f]$ is high. The control inputs $req[t,f]$ are mutually exclusive. Outputs of the storage element are, z and y .

The storage element has two identical multiplexers with different data signals. One multiplexer forwards data when its control signal is true and the other forwards data when its control signal is false. Each multiplexer has two data inputs, two control inputs, and one data output. The storage element can be represented with either of two icons. The two icons of the storage element are shown on the right hand side. The symbols of the storage element are drawn assuming that the control signal, $req[t,f]$ is false.

In figure 3.11 the top multiplexer copies the input, a to its output, z when the control signal, $req[t,f]$ is false. The storage element copies input, k_a to its output, z when the control signal, $req[t,f]$ is true.

Figure 3.11: *Storage Element for Serial to Parallel FIFO: The storage element has two multiplexers. Each multiplexer copies one of its two data inputs to its output, depending on the state of its control signals.*

In figure 3.11 the bottom multiplexer copies the input, b its output, y when the control signal $req[t,f]$ is true. The storage element copies the input, k_b to its output, y when the control signal, $req[t,f]$ is false.

3.2.2 Operation of S2P Micropipeline

The micropipeline implementation of the S2P FIFO appears in the figure 3.12. The micropipeline implementation of the S2P FIFO is eight bit wide and eight stages long, which implies the serial data input is eight bit wide, and eight blocks long. For simplicity of explanation this section explains a one bit wide and 4 stages long S2P FIFO for clarity.

Figure 3.12 also shows the circuit for the full detector circuit for micropipeline S2P FIFO. Recall a micropipeline FIFO is full when the outputs of the side-by-side Muller-C gates differ. To avoid the use of a counter in the design, the S2P FIFO has a *Full Detector* which is a string of AND gates. The AND gates evaluate the full state of the FIFO. As the S2P FIFO fills with the serial input data the output of the side-by-side dual rail Muller-C gates differ in state because one end of the S2P FIFO is blocked by holding one of the inputs of the dual rail Muller-C gate of the last stage, $ACK[t,f]$ to an *OFF* state that is the $ACK[t]$ is tied to *GND* and $ACK[f]$ is tied to *VDD*. When all the outputs of all the side-by-side Muller-C gates differ then the output of the AND gate string is high. A high output of the AND string implies that the four bit data are ready for delivery from the S2P FIFO.

Inputs to the S2P FIFO are: the data-timing signal, $DQS[t,f]$, the serial data input, Din , the master-clear signal, mc , and the fire signal from the next GasP stage, cap . The input, cap , from the next GasP stage indicates that the next GasP stage has captured the parallel data. Outputs of the S2P FIFO are: the parallel data output, $Dout[0:3]$, and the *take* signal. The *take* output acts like a state wire between the next GasP stage and the S2P FIFO. The *take* output informs the next GasP stage that the current operation of the S2P FIFO is complete. The control signal mc or cap signal empties the S2P FIFO by setting its dual rail Muller-C gates to false.

The micropipeline implementation of the S2P FIFO has a row of the storage elements, controlled by the dual rail Muller-C gates. The row of the storage elements forms the data path of the S2P FIFO. The dual rail Muller-C gates form the control path of the S2P FIFO. The dual rail Muller-C gate controlling the storage element forms one stage of the FIFO.

Figure 3.13 shows the output of several Muller-C gates, for example $x1[t,f]$, as twin traces, one purple and one orange. Recall that a stage is full if two Muller-C gates differ in state, and empty if they match. Thus at the very left of the traces for $x1[t,f]$ to $x6[t,f]$, after master clear, the dual traces all match. The FIFO is empty. The yellow color arrows are causality arrows. In the figure more than one causality arrow pointing to a trace indicates an AND function of the traces from which these causality arrows originate.

Figure 3.13: *Working Trace of S2P Micropipeline*

As the $DQS[t,f]$ signal appears with the serial input data, Din , the $DQS[t,f]$ signal fills the S2P FIFO with the serial input data. This is denoted by the transitions on the $x1[t,f]$ to $x6[t,f]$ signals. As shown in figure 3.12 one of the inputs of the last Muller-C gate to the right is tied to ground so there is only one transition at the output, $x6[t,f]$ of the last Muller-C gate.

As the S2P FIFO becomes full, the outputs of the side-by-side Muller-C gates differ in state. Thus at the right, traces for $x3[t,f]$ to $x6[t,f]$ differ in state. This indicates that the S2P FIFO is full. The output of the full detector is shown in blue trace, *allfull* in panel9. This is an active low signal. A falling transition of the *allfull* signal indicates to the next stage that parallel data are available.

The falling transition of the *allfull* signal makes a transition to the *take* signal that is state wire of the successor GasP stage not shown in figure 3.12. The yellow causality arrow shows this from falling edge of *allfull* signal to the rising edge of the *take* signal.

When the successor GasP stage fire is in the high state, shown as the green trace *cap* in figure 3.13, the *cap* pulse captures the parallel data, *Dout[0:3]*, the *cap* makes the *take* wire low, and the *cap* makes the S2P FIFO empty. This is shown by the yellow causality arrow from rising edge of the *cap* signal to the traces *take*, $x6[t,f]$, $x4[t,f]$, and $x2[t,f]$ in panel 10, 8, 6, and 4 respectively.

Figure 3.14 shows the trace from the simulation of micropipeline S2P FIFO. The micropipeline S2P FIFO simulated is four bits wide. The color of traces of the simulation are similar to the waveform trace shown in figure 3.13. The figure omits the traces for the data values. In the figure 3.14 all the simulation traces match the working trace shown in figure 3.13. In the simulation trace width of the *DQS[t,f]* signal pulse is 0.5ns, width of the output of the Muller-C gates, $x1[t,f]$, $x2[t,f]$, $x3[t,f]$, $x4[t,f]$, $x5[t,f]$ and $x6[t,f]$ is also 0.5ns, and width of the *take* signal pulse shown as red trace is 0.4ns.

Figure 3.14: *Simulation Trace of Micropipeline S2P FIFO*

Parallel to Serial (P2S) FIFO

After receiving a read command, the DRAM array provides data bits from the requested address location in parallel. A parallel to serial FIFO (P2S) converts the parallel data bits into a serial data burst to reduce number of pins at the data interface of the self-timed DRAM. The P2S FIFO also provides the data timing signal for the data burst.

The P2S FIFO can accept a parallel load, if all the stages of the P2S FIFO are empty. After the parallel load, each stage of P2S FIFO holds one block of the serial burst. The number stages of the P2S FIFO depends on the number blocks in the serial burst. The P2S FIFO shifts out the blocks serially. The parallel load will not happen again until the P2S FIFO is empty meaning all the blocks of the serial burst have been shifted out. To know when to load the parallel load, the P2S FIFO uses a empty detector.

To explore conversion speed, I simulated two designs of P2S FIFO using self-timed circuits: one uses GasP circuits and other uses micropipeline circuits. This chapter explains the two implementations of P2S FIFO.

Figure 4.1 shows the read logic block on the self-timed DRAM. The read logic block on the self-timed DRAM accepts 64-bit parallel data from the DRAM array and produces a data burst of eight blocks long each eight bits wide. Inputs to the read logic block are 64-bit data from the DRAM array, *Datan*[0:63], and the

two-phase handshake wire, *Send Data (SD)*, from the processing unit. Outputs from this block are eight bit wide data bus, $Data[0:7]$, the read data-timing signal, $DQS[t,f]$, and the two-phase handshake wire, *Data Available (DA)*.

Figure 4.1: *Block Diagram of Read Logic Operation on Self-Timed DRAM: The read logic block on the self-timed DRAM accepts parallel data from the DRAM array and provides the serial form of the input data.*

4.1 GasP Implementaion of P2S FIFO

This section explains the implementation of the P2S FIFO using GasP circuits [4]. The GasP implementation of P2S FIFO accepts parallel data input of 64 bits, and produces the serial data of eight bits wide and eight blocks long. To simplify the explanation this section explains a one bit wide and four blocks long P2S FIFO.

The P2S FIFO can accept a parallel load if all the stages of the P2S FIFO are empty. To know when to load the parallel load, the P2S FIFO uses a empty

detector. Figure 4.2 shows the circuit of the empty detector. The empty detector has four stages denoted by the black rectangles. Each stage of the empty detector circuit is embedded inside one GasP stage of the S2P FIFO. Each stage of the empty detector is an AND function of the state wires. Output of the empty detector is an active low signal, *empty(low)*. Output of the empty detector is low when all the inputs, *pred1*, *pred2*, *pred3*, *pred4* and *pred5* are in the low state. The empty detection takes place from left to right. The first stage becomes empty first because it has to move only one block of the serial burst. The last stage of the FIFO has to move all the blocks of the serial burst and hence the last stage becomes empty last.

Figure 4.2: *Empty Detector for GasP P2S FIFO*

Figure 4.3 shows the GasP implementation of the P2S FIFO. The P2S FIFO has four GasP stages. The GasP implementation of the P2S FIFO has: 4-bit wide input data pins, *Din[0:3]*, two bidirectional pins, *pred* and *succ*, and three output pins, *DQS[t,f]* and *Dout*. The P2S FIFO shown can be made wider with more latches on each stage. The P2S FIFO can be made longer by adding GasP stages to the right of P2S FIFO.

Figure 4.3: Schematic of GasP P2S FIFO: The P2S FIFO shown can be made wider with more latches on each stage. The P2S FIFO can be made longer by adding GasP stages to the right.

The fire pulse from the first GasP stage, *fire1* loads the bits from the parallel input data into all the four stages of the P2S FIFO. The fire pulse from the first stage makes the successor state wires of the all the stages of the P2S FIFO full, by pulling the four state wires *sw[1]*, *sw[2]*, *sw[3]*, and *succ* high. When the successor state wire of all the stages of P2S GasP FIFO are in the high state, it indicates the P2S FIFO is FULL, and ready for parallel to serial conversion.

The first stage will avoid producing a fire pulse again until the current operation of the P2S FIFO completes. There are two conditions for the first stage to produce a fire pulse: first, when predecessor state wire from first stage is in the high state indicating that input data are available, and second, when all the successor state wires of all the stages of the P2S FIFO are in the low state, indicating that all the stages are empty.

Figure 4.4 shows the working trace of the GasP P2S FIFO. The red color traces are fire signals; the green color traces are state wires; the black color traces are data wires; the orange and the purple traces shown are the true and false values of the DQS[t,f] signal. The yellow, black, grey and brown color arrows are causality arrows. In the figure more than one causality arrow pointing to a trace indicates an AND function of the traces from which these causality arrows originate.

Figure 4.4: Working Trace of P2S GasP FIFO

The GasP Parallel to Serial FIFO (P2S) starts its action with a *fire1* pulse, as shown in the second panel. The *fire1* pulse can happen only when the state wire input, *pred* is high, as shown in second panel, and the P2S FIFO is *empty*, as shown in panel 12. At the time of the *fire1* pulse, the input data to the P2S circuit must be present, as shown in the first panel.

The *fire1* signal does several things. First, it loads the input, *Din[0:3]* into all four of the stages of the P2S FIFO. Second, the *fire1* signal makes the four state wires, *sw[1]*, *sw[2]*, *sw[3]*, and *succ* high as shown in panel 4, 6, 8, and 11. The yellow causality arrow from the rising edge of the *fire1* to the state wires *sw[1]*, *sw[2]*, *sw[3]*, and *succ* shows this operation. The *fire1* signal also captures the first serial data, so that the first data appears on the output pin, *Dout*. Finally, the *fire1* signal makes a transition on the data-timing signal, *DQS[t,f]* as indicated by the longest yellow causality arrow. The first stage will not produce another fire pulse, *fire1* until after the current operation of the P2S FIFO completes.

An active low AND function of all the state wires detects the completion of the P2S FIFO operation. The output of this function is the active low signal, *empty*, shown near the bottom in the blue trace. The FIFO is empty only when all its stages are empty. If any stage is full, the FIFO is not yet empty. A part of this AND function is embedded in each of the GasP stages of the P2S FIFO. Notice that the falling edge of the *empty* signal initiates the second *fire1* pulse near the end of the time illustrated in the figure.

The AND function for the *empty* signal is embedded in the GasP stages of the P2S FIFO. The signals called *OR2* and *empty* are outputs of this AND function. They appear as blue traces in the diagram. The black causality arrows show how the circuit constructs the empty signal. The *OR2* signal is high when either the state wire *sw[1]* or the state wire *sw[2]* is high; *OR2* is low only when the state wires, *sw[1]* and *sw[2]* are both low. Thus *OR2* indicates that the first half of the FIFO is empty. An AND gate in the third FIFO stage combines *OR2* with the state wire, *sw[3]* to produce the signal called *empty*. The *empty* signal actually

indicates that the first three stages are all empty. The first stage of the FIFO combines the *empty* signal with *sw[4]* so that *fire1* can happen only when all four stages are empty.

A second *fire1* pulse can happen only when all four stages of the FIFO are empty. The second *fire1* pulse starts when three conditions are true: 1) the *empty* signal is low, indicating that *sw[1]*, *sw[2]*, *sw[3]* are all low, and 2) *succ* is low, indicating that the last serial output has been taken, and 3) the *pred* input is high, indicating that new parallel data are available. The saffron causality arrows from the falling edges of *sw[1]*, *succ*, *empty* and the rising edge of *pred* illustrate this.

The pulses for *fire2*, *fire3* and *fire4* are drawn red in panels 5, 7, and 10 of the figure respectively. Notice that they form a pyramid shape. The *fire4* signal from the last stage acts three times because it has to move three data bits, *Din[1]*, *Din[2]*, and *Din[3]*. The *fire3* signal acts twice because it has to move only two data bits, *Din[2]* and *Din[3]*. And the *fire2* signal acts only once because it has to move only data bit *Din[3]*.

The data timing signal, *DQS[t,f]* appears near the bottom of the figure. It has both the true and false forms. The *DQS[t,f]* is true when the *DQS[t]* is high and the *DQS[t,f]* is false when the *DQS[f]* is high. The *DQS[t]* and *DQS[f]* signals are mutually exclusive. The *fire1* pulse and the three *fire4* pulses generate transitions on *DQS[t,f]*, shown by the yellow causality arrow from in the figure. This matches the fact that the *fire1* pulse loads the first data bit *Din[0]*, and the three *fire4* pulses load the other three data bits *Din[1]*, *Din[2]*, and *Din[3]*.

Figure 4.5 shows the trace from the simulation of GasP P2S FIFO. The GasP P2S FIFO simulated is 1-bit wide and 4 stages long. The color of the traces of the simulation are similar to the waveform traces shown in figure 4.4. The figure omits the traces for the data values.

In the figure 4.5 all the simulation traces match the working traces shown in figure 3.4. In the simulation trace width of the $DQS[t,f]$ signal pulse is 0.8ns, and width of the *fire* signal pulses shown as red traces is 0.4ns.

Figure 4.5: *Simulation Trace of GasP P2S FIFO*

Figure 4.6 shows the GasP circuit of the first stage. The pin configuration of the first stage GasP circuit is: two input pins *in1* and *in2*, two bidirectional pins *pred* and *succ*, and an output pin *fire*. The *fire* pulse from the first GasP circuit does four operations: the fire pulse loads bits of the input data into all the stages of the P2S FIFO; the fire pulse pulls the successor state wire high; the fire pulse pulls the predecessor state wire low; and the fire pulse also pulls successor state wire of all stages high indicating that the P2S FIFO is FULL.

Figure 4.6: *First stage GasP Circuit of Parallel to Serial FIFO: The inputs *in1* and *in2* indicate all stages of the FIFO are empty. The first stage GasP circuit produces a fire pulse only when the predecessor state wire from the first stage is high and the inputs *in1* and *in2* are low.*

The first stage GasP circuit has a three input bubbled AND gate for the fire control. The first GasP circuit produces a fire pulse when *pred*, the predecessor wire is in the high state, *in1* and *in2* are in the low state indicating that all stages of the P2S FIFO are empty. The input *in1* is the output of the AND function embedded in the last stage of the P2S FIFO. To start the next operation with minimum delay, the successor state wire of the last stage, *in2* is the third input of the bubbled input AND gate because the successor wire of the last stage goes low last, and remains low until the P2S FIFO loaded with parallel load. The first stage GasP circuit controls the conventional one-input latch explained in "Chapter-2".

The second stage of the P2S GasP FIFO has GasP circuit controlling a two-input latch explained in "2". Figure 4.7 shows the GasP circuit used in the second stage. The second stage has two bidirectional pins *succ* and *pred*, two output pins *fire* and *aout*, and one input pin *fire1*. The second stage GasP circuit is similar to the conventional GasP circuit except the successor driver, *SD*. The *fire1* input which is the fire signal from first stage or the fire signal from this stage, *fire* drives the successor state wire of this GasP circuit. The second stage GasP circuit forwards the state of the first stage to the third stage. The control signals for the two-input latch of this stage are fire signal from first stage and this stage.

Figure 4.7: *Second Stage GasP Circuit of Parallel to Serial FIFO: The second stage GasP circuit forwards state of first stage of the P2S FIFO to the third stage as output, aout.*

Figure 4.8 shows the schematic of the GasP circuit of the remaining stages. The pin configuration of this stage is: two output pins *fire* and *aout*, two bidirectional pins *pred* and *succ*, and two input pins *ain* and *fire1*. The GasP circuit for remaining stages is similar to the second stage of the P2S FIFO with an extra AND function. Output of the AND function is in the high state when the input, *ain*, is in the low state and the predecessor of this stage is in the low state. The high output of the AND gate indicates to the successor stage that all the previous stages are

empty. The inverted output of the AND function is the input to the AND function embedded in the successor stage of the P2S FIFO. Either the fire signal from first stage, which is the input *fire1*, or the fire signal from this stage drives the successor state wire of this GasP circuit. The control signals for the two-input latch are the fire signal from this stage and the first stage.

Figure 4.8: *GasP Stage for Parallel to Serial FIFO: Input *ain* indicates that all the stages to the left except the predecessor stage are empty. Output *aout* indicates that all the predecessor stages are empty.*

Figure 4.9 shows the circuit for generating the differential data timing signal, $DQS[t,f]$. This simple circuit generates the $DQS[t,f]$ signal with three conventional latches. The trigger for the circuit is the $fire1$ signal from the first GasP stage and $fire4$ signal from the last GasP stage. The $fire1$ signal shifts out the first serial bit and the other bits are shifted out by the $fire4$ signal. The $fire1$ and $fire4$ pulses are the sampling signals for the output serial data. The rising and the falling edges of the $DQS[t]$ signal must be aligned with the falling and the rising edges of the $DQS[f]$ signal. Two latches at the output are used to achieve the alignment. The two latches are controlled by the same control signal, $L2$ and are fed with true and false value of the input.

Figure 4.9: DQS signal generator for Parallel to Serial FIFO.

4.2 Micropipeline Implementaion of P2S FIFO

The micropipeline implementation of P2S FIFO is hard for three reasons. First, the storage elements for the P2S FIFO must have 4 input multiplexer. Recall that the ordinary micropipeline storage element has only two inputs. Second reason that the P2S FIFO is hard because of the empty detector. For the empty detection a shadow path of the storage elements is introduced in addition to the data path storage elements. Each time we load data into the data path, we load a group of ones into the shadow path. As parallel to serial conversion happens, zeroes enter the shadow path. When a zero reaches the output of the shadow path we know to reload the data path. The third reason is the need to do the parallel load in two parts. The odd indexed and even indexed bits must be loaded separately into the P2S FIFO for a smooth output pulse.

The micropipeline implementation of a P2S FIFO has a control path and a data path. The control path uses dual rail Muller-C gates explained in "2", and the data path uses storage elements. This section explains the micropipeline implementation of the P2S FIFO [5] and the micropipeline parts of P2S FIFO.

4.2.1 Storage Element for Micropipeline P2S FIFO

Figure 4.10 shows the schematic of the storage element for the data path. Inputs to the storage element can be divided into two parts: the data part and the control part. The data inputs are a , ld_a , b , ld_b , k_a , and k_b . The control inputs are master-clear, mc , capture-even, c_e , capture-odd, c_o , and $req[t,f]$. The $req[t,f]$ is true when the $req[t]$ is high and the $req[f]$ is false when the $req[f]$ is high. The $req[t]$ and $req[f]$ signals are mutually exclusive. Outputs of the storage element

are, z and y .

The storage element has two identical multiplexers with different control signals and data signals. One multiplexer forwards data when its control signal is true and the other forwards data when its control signal is false. Each multiplexer has three data inputs, four control inputs, and one data output. The storage element can be represented with either of two icons. The two icons for the storage element are shown on the right hand side. The icons of the storage element are drawn assuming that the control signal, $req[t,f]$ is false.

In the figure 4.10 the top multiplexer initializes its output, z , to a low state when the control input mc is in the high state. The multiplexer copies the input, ld_a , to its output if the control signal, c_e , is in the high state. The multiplexer copies the input, a , to its output, when the control signals, $req[t,f]$ is false and c_e is in the low state. The multiplexer copies the input, k_a , to its output, if the control signals, $req[t,f]$ is true and c_e is in the low state.

In the figure 4.10 the bottom multiplexer initializes its output, y to a low state when the control input, mc is in the high state. The multiplexer copies its input, ld_b , to its output when control signal, c_o , is in the high state. The multiplexer copies the input, b , to its output, when the control signals, $req[t,f]$ is true and c_o is in the low state. The multiplexer copies the input, k_b , to its output, when the control signals $req[t,f]$ is false and c_o is in the low state.

Figure 4.10: *Storage Element for Parallel to Serial FIFO: The storage element has two multiplexers with four control inputs. Each multiplexer copies the data inputs to the output, depending on the state of the control signals.*

4.2.2 Operations of P2S Micropipeline

The micropipeline implementation of the P2S FIFO is shown in the figure 4.11. The micropipeline implementation of the P2S FIFO accepts parallel data input of 64 bits, and produces a serial data output of eight bits wide and eight blocks long. This section explains a one bit wide and four blocks long P2S FIFO to simplify

the explanation. The bubbles in the figure 4.11 signify inversion.

Inputs to the P2S FIFO are: four bit parallel data, $Din[0:3]$, and the $ACK[t,f]$. Outputs of the P2S FIFO are the data timing signal, $DQS[t,f]$, and the serialized data, $Dout$. The pin sw is the bidirectional port.

In the figure 4.11 the rectangular boxes represents two storage elements, SE_C and SE_D . The storage element, SE_C is for the shadow control path shown by the red wires and the storage element, SE_D is for data path shown by the blue wires. The Muller-C element controlling the storage element form a stage of the P2S FIFO.

Each row of storage elements has two data paths. The two data paths are brought together again only at the output end of the P2S FIFO by an output selector similar to a storage element. This storage element is controlled by the $ACK[t,f]$ signal.

To avoid the use of a counter in the design, the upper row of the shadow storage elements, SE_C is loaded with ones at the start of operation of the P2S FIFO. Every time a bit shifts out of the P2S FIFO, a zero enters from the first storage element of the P2S FIFO. When the upper row of storage elements fills with zeroes then the current operation of the P2S FIFO is complete, and the next operation can start.

When the control signals capture-even, c_e , and capture-odd, c_o , are in the high state, the upper row storage elements, SEC , of the data path are loaded with ones. One end of the control path is grounded. Every time the data bit shifts out of the P2S FIFO, a zero comes into the first storage element of the P2S FIFO. When the upper row of storage elements fills with zeroes then the current operation of the P2S FIFO is complete, and the next operation can start.

Figure 4.12 shows the working trace of the micropipeline P2S FIFO. The yellow, brown and grey color arrows are causality arrows. In the figure more than one causality arrow pointing to a trace indicates an AND function of the traces from which these causality arrows originate.

In the figure the outputs of several Muller-C gates, for example $x1[t,f]$, are shown as twin traces, one purple and one orange. Recall that a stage is full if two Muller-C gates differ in state, and empty if they match. Thus at the very left of the traces for $x1[t,f]$ to $x4[t,f]$, after master clear, the dual traces all match. The FIFO is empty. In order to load parallel data into the FIFO we want the FIFO to be full. Following master clear, an inverter to the left of the first stage fills the FIFO. The resulting action appears at the left of the $x1[t,f]$ to $x4[t,f]$ traces at the left of the figure. In the next idle period of these traces successive stages differ in state as can be seen in the traces after the rapid initialization action at the left of the figure. This leaves the control of the FIFO ready for loading the first set of data.

When the micropipeline P2S circuit operates continuously, we cannot load all the parallel data at once. Loading all data at once would disturb the final stage of the FIFO as it delivers data and the $DQS[t,f]$ signal. Instead, we load the even

and odd bits separately. The load signals for this purpose appear in the timing diagram as the red pulses c_e and c_o . Notice that for the first load, they appear simultaneously, but for subsequent loads, they appear separately.

Figure 4.12: *Working Trace of Micropipeline P2S FIFO*

There is a second problem related to telling when the micropipeline is empty. Unlike the GasP circuit, the micropipeline lacks a simple indication of full or empty. How are we to know when to reload the P2S micropipeline with the next set of parallel data? We provide a shadow data path in parallel with the first. Each time we load

data into the data path, we load a group of ones into the shadow path. As parallel to serial conversion happens, zeroes enter the shadow path. When a zero reaches the output of the shadow path we know to reload the data path.

Master clear for the micropipeline occurs in two parts. First, all of the dual rail Muller-C gates are initialized to a low state, making the P2S FIFO empty. This condition appears at the left of the timing diagram. Next, an inverter at the input of the micropipeline fills its control. This action can be seen at the left of traces 8-11, ending in a longer quiet period for these traces. Notice that successive Muller-C element outputs differ: purple low in panel8, purple high in panel9, purple low in panel10, and purple high in panel11. The control of the Micropipeline is now full.

In fact, the control of the micropipeline P2S converter remains full throughout all subsequent operation. From time to time a jam transfer inserts data into its data latches, and ones into the shadow latches. When to insert data and ones is controlled by zeros appearing at the output of the shadow latches.

When parallel input data, $Din[0:3]$ are present, as shown in first panel, and the input, sw , is high shown in second panel, the c_e and c_o pulses, shown in panel 3, and 5, load the even and odd bits respectively of the input, $Din[0:3]$ into the lower row, SE_D , of the P2S FIFO. The c_e and c_o also load the upper row, SE_C with ones. The outputs of the last stage storage elements of the shadow FIFO, called $b0$ and $b1$, are shown in panel 4, and 6. They become one in response to the loading pulses c_e and c_o .

The c_e makes $b0$ high shown by yellow causality arrow from rising edge of the c_e signal. The c_o makes $b1$ high shown by yellow causality arrow from rising edge of

the c_o signal. The $b0$ and $b1$ then make the c_e and c_o signal low respectively shown by the yellow causality arrow from rising edge of the $b0$ and $b1$ signal to the falling edge of the c_e and c_o signal respectively.

The rising edge of the $b0$ and $b1$ signals make $data[t,f]$ true as shown in panel7. The $data[t,f]$ signal copies the $x4[t,f]$, which is the output of the last dual rail Muller-C gate at the right of the P2S FIFO to the data timing signal, $DQS[t,f]$ shown in panel13. The $ACK[t,f]$ signal in panel12 sets the frequency of the operation of P2S FIFO.

The $b0$ signal goes low again when the last even bit $Din[2]$ moves out of the P2S FIFO from the right shown by the yellow causality arrow from $x4[t,f]$ to the falling edge of the $b0$ signal. If there are data $Din[0:3]$ and $succ$ is high then the falling edge of $b0$ causes the c_e to go high shown by the yellow causality arrow from falling edge of the $b0$. The c_e loads the even bits $Din[0,2]$ into the lower row, SE_D of the P2S FIFO. The c_e also loads ones at even bit position in upper FIFO which makes the $b0$ signal high.

The $b1$ signal goes low again when the last odd bit $Din[3]$ moves out of the P2S FIFO from the right shown by the yellow causality arrow from $x4[t,f]$ to the falling edge of the $b1$ signal. If there are data $Din[0:3]$ and $succ$ is high then the falling edge of $b1$ causes the c_o to go high shown by the yellow causality arrow from falling edge of the $b1$. The c_o loads the odd bits $Din[1,3]$ into the lower row, SE_D of the P2S FIFO and makes the sw signal low. The c_o also loads ones at odd bit position in upper FIFO which makes the $b1$ signal high. The c_e and c_o signal load the even and odd bits at different times. This allows the next operation

of the P2S FIFO to start with minimum delay of the control path.

Figure 4.13 shows the working trace of the micropipeline P2S FIFO from simulation. The micropipeline P2S FIFO simulated is one bit wide and four stages long. The color of traces of the simulation are similar to the waveform traces shown in figure 4.12. The figure omits the traces for the data values. In the figure 4.13 all the simulation traces match the working trace shown in figure 4.12. In the simulation trace width of the $DQS[t,f]$ signal pulse is 0.5ns, width of the c_e and c_o signal pulse shown as red trace is 0.3ns.

Figure 4.13: *Simulation Trace of Micropipeline P2S FIFO*

Results

This chapter explains my simulations and reports my results. The CMOS process technology I used for simulations is 180nm with a nominal supply voltage of 1.8V. Simulation reveals the effect of the width mismatch on the GasP and micropipeline implementation of parallel to serial (P2S) FIFO and serial to parallel (S2P) FIFO. Simulation also reveals the maximum data burst rate of the two implementations of the P2S FIFO and the S2P FIFO.

5.1 Read Simulation Waveform

Figure 5.1 shows the output hspice simulation of read cycles for 180nm process technology. The simulation waveform shows two consecutive read data bursts. For simulation purpose the read data bursts are one bit wide and eight blocks long. Recall for read, the two-phase handshake wires are *Data Available (DA)* and *Send Data (SD)*. The self-timed DRAM controls the DA wire and the processing unit controls the SD wires. In the figure, states of the DA wire shown in Panel1 and the SD wire shown in panel2 match at start of the simulation. Also, $DQS[t,f]$ signal shown in panel3 is idle at start. When the self-timed DRAM has data ready for delivery, it makes a transition on the DA wire. When the processing unit has space for data, it makes a transition on the SD wire. When the state of SD and DA match, data transfer starts. The black arrow in the figure indicates the start of first data burst. The hex value of the data burst is 0xFF. After the first transition on the SD wire, the self-timed DRAM makes another transition on the DA wire

requesting the second burst. Notice that the self-timed DRAM makes this request during transfer of the first data burst. The second burst starts after the processing unit makes transition on the SD wire. The hex value of the data burst is 0x1B.

Figure 5.1: *Simulation of Read Operation: These simulation waveforms demonstrate consecutive read data burst.*

5.2 Width Mismatch

After a chip is fabricated there are some variations in the transistor attributes like lengths, widths, oxide thickness etc. These variations are not uniform throughout a chip. These variations effect the transistors with smaller widths more than the transistors with larger widths.

The circuits of the self-timed DRAM data interface is designed with careful sizing of the transistors using logical effort calculation. Due to precise sizing of the

transistors in the circuits any change in the width of the transistors due to fabrication will have a significant effect on the performance of the circuits. To show the maximum effect of the variations due to fabrication on the self-timed DRAM data interface, I choose to vary only the width of transistors in the design. To model the fabrication effects the percentage deviation in width of the minimum width transistor is applied as standard deviation for widths to transistors of all widths.

To determine the width mismatch for the 180nm CMOS process technology I use the saturation current equation [6] of a transistor because the saturation current equation takes into consideration transistor attributes like length, width and oxide thickness. I calculated the saturation current of a transistor for threshold voltage variation, and how much width variation results in the same saturation current variation. The threshold voltage variation for 180nm process technology is about 30mV [7]. Then I calculated how much width variation results in the same saturation current for a minimum width transistor.

Saturation Current

$$I_{ds} = \mu C_{OX} \cdot \frac{W}{L} \cdot (V_{gs} - V_t)^2$$

where,

μ = mobility of electrons

C_{OX} = capacitance per unit area of gate oxide

W = channel width

L = channel length

V_{gs} = gate to source voltage for 180nm CMOS process technology (1.8V)

V_t = threshold voltage for 180nm CMOS process technology (0.39V)

Saturation current with threshold voltage variation

$$I_{ds} = \mu C_{OX} \cdot \frac{W}{L} \cdot [V_{gs} - (V_t \pm \Delta V_t)]^2 \quad (5.1)$$

where,

ΔV_t = threshold voltage deviation (30mV)

Saturation current with width variation

$$I_{ds} = \mu C_{OX} \cdot \frac{(W \pm \Delta W)}{L} \cdot [V_{gs} - V_t]^2 \quad (5.2)$$

where,

ΔW = channel width deviation

Percentage deviation in width for 180nm technology can be modeled by equating equation 5.1 and 5.2

$$\begin{aligned}\mu C_{OX} \cdot \frac{W}{L} \cdot [V_{gs} - (V_t \pm \Delta V_t)]^2 &= \mu C_{OX} \cdot \frac{(W \pm \Delta W)}{L} \cdot [V_{gs} - V_t]^2 \\ W \cdot [V_{gs} - (V_t \pm \Delta V_t)]^2 &= (W \pm \Delta W) \cdot [V_{gs} - V_t]^2\end{aligned}\tag{5.3}$$

After solving equation 5.3 for minimum width transistor we get

$$\frac{\Delta W}{W} = 4.30\%$$

where,

$$\Delta W = W \pm W'$$

where,

ΔW = percentage deviation in width of minimum width transistor

The effect of width mismatch diminishes as the width of the transistors increases. Figure 5.2 shows the effect of width mismatch on the logic threshold voltage of a inverter with input and output tied together, as a function of width of the transistor. The *y-axis* the output voltage and the *x-axis* is the inverter width. The graph shows how the effect of width mismatch diminishes as the transistors width increases.

Figure 5.2: *Effect of Width Mismatch on Transistor Output Voltage: Effect of width mismatch diminishes as the transistors width increases.*

The effect of width mismatch on the P2S and the S2P FIFO is shown using shmoo plots. Both the P2S and the S2P FIFO are simulated using the Monte-Carlo method in hspice. In these simulations each Monte-Carlo run is of 20 iterations that is 20 different instances of the circuit are simulated in one Monte-Carlo run. All the circuits simulated in each Monte-Carlo run differ in their width attribute.

Figure 5.3 shows the simulation setup for the Monte-Carlo simulations of P2S FIFO. The *Data Source* block consists of GasP circuit controlling a shift register. The *Oscillator* block is a pulse generator. The *Data Sink* block is a GasP sink circuit controlling a latch. For simulations, the P2S FIFO is fed with parallel input data. The serial data are shifted out of the P2S FIFO at a rate that depends on

the oscillator frequency. The design is simulated for different values of voltages and the maximum data burst rate of the design is measured. For a given voltage the oscillator frequency is tuned to get correct expected serial serial data burst, $Data[0:7]$ for the minimum DQS cycle time.

Figure 5.3: *Block Diagram for Monte-Carlo Simulations of P2S FIFO*

Figure 5.4 and 5.5 show shmoo plots for the GasP and the micropipeline implementations of P2S FIFO, respectively. The simulated P2S FIFO accepts 64-bit wide input and produces eight blocks each eight bits wide as output. In both the figures, the y-axis is the DQS cycle time and the x-axis is the supply voltage. Each dot on the shmoo plots represents the result of a Monte-Carlo run on the P2S FIFO for a given supply voltage and DQS cycle time. A dot on the shmoo plot reports in fail if the P2S FIFO produces output different than the expected output, $DQS[t,f]$ and $Data[0:7]$, for a given correct input, $Data[0:63]$.

The color code for dots in the shmoo plot represents percentage pass or fail for a Monte-Carlo run of 20 iterations. The green color dots represent 100% pass and the red color dots represent 100% fail. The dots between the red and green dot

show the effect of width mismatch on the yield of P2S FIFO. The width mismatch results in fails shown by different color code in the shmoo plot.

Table 5.1 and 5.2 shows the shmoo plot summary for three supply voltages. The table compares the simulation results with width mismatch from the shmoo plots with the simulation results without width mismatch. Comparison of the two results shows that introduction of width mismatch to P2S FIFO reduces the yield of the design. Introduction of width mismatch in the design changes the specification of maximum data burst rate of the design at a given supply voltage.

Figure 5.4: *Shmoo Plot of GasP implementation of P2S FIFO*

Supply Voltage	DQS Cycle Time with Width variation for Yield of 85%-100%	DQS Cycle Time without Width Variation	Yield for DQS Cycle Time without Width Variation
1.7V	1.61ns to 1.63ns	1.6ns	50%-70%
1.8V	1.51ns to 1.54ns	1.45ns	0%
1.9V	1.41ns to 1.44ns	1.37ns	50%-70%

Table 5.1: *Shmoo Plot Summary for GasP P2S FIFO: The table compares DQS cycle time for width mismatch and without width mismatch on GasP P2S FIFO*

Figure 5.5: *Shmoo Plot of Micropipeline implementation of P2S FIFO*

Figure 5.6 shows the simulation setup for the Monte-Carlo simulations of S2P FIFO. The *Data Source* block consists of GasP circuit controlling a shift register.

Supply Voltage	DQS Cycle Time with Width variation for Yield of 85%-100%	DQS Cycle Time without Width Variation	Yield for DQS Cycle Time without Width Variation
1.7V	1.6ns to 1.64ns	1.6ns	50%-70%
1.8V	1.5ns to 1.53ns	1.49ns	70%-90%
1.9V	1.43ns to 1.46ns	1.41ns	70%-90%

Table 5.2: *Shmoo Plot Summary for Micropipeline P2S FIFO: The table compares DQS cycle time for width mismatch and without width mismatch on Micropipeline P2S FIFO*

The *Oscillator* block is a pulse generator. The *Data Sink* block is a GasP sink circuit controlling a latch. For simulations, the S2P FIFO is fed with serial data burst. The serial data are shifted into the S2P FIFO at a rate that depend on the oscillator frequency. The design is simulated for different values of voltages and the maximum data burst rate of the design is measured. For a given voltage the oscillator frequency is tuned to get correct expected parallel output for the minimum DQS cycle time.

Figure 5.6: *Block Diagram for Monte-Carlo Simulations of S2P FIFO*

Figure 5.7 and 5.8 show shmoo plots for the GasP and the micropipeline implementations of S2P FIFO. The simulated S2P FIFO accepts eight blocks each eight bits wide as input and produce 64-bit wide output. In both the figures, the y-axis

is the DQS cycle time and the x-axis is the supply voltage. Each dot on the shmoo plots represents the result of a Monte-Carlo run on the S2P FIFO for a given supply voltage and DQS cycle time. A dot on the shmoo plot reports a fail if the S2P FIFO produces output different than the expected, $Data[0:63]$, for a given correct input, $DQS[t,f]$ and $Data[0:7]$.

The color code for dots in the shmoo plot represents percentage pass or fail for a Monte-Carlo run of 20 iterations. The green color dots represent 100% pass and the red color dots represent 100% fail. The dots between the red and green dot show the effect of width mismatch on the yield of S2P FIFO. The width mismatch results in fails shown by different color code in the shmoo plot.

Table 5.3 and 5.4 shows the shmoo plot summary for both implementations of the S2P FIFO. The table compares the simulation results with width mismatch from the shmoo plots with the simulation results without width mismatch. Comparison of the two results shows that introduction of width mismatch to S2P FIFO reduces the yield of the design. Introduction of width mismatch in the design changes the specification of maximum data burst rate of the design at a given supply voltage.

Figure 5.7: *Shmoo Plot of GasP implementation of S2P FIFO*

Supply Voltage	DQS Cycle Time with Width variation for Yield of 85%-100%	DQS Cycle Time without Width Variation	Yield for DQS Cycle Time without Width Variation
1.7V	1.6ns to 1.63ns	1.59ns	50%-70%
1.8V	1.53ns to 1.57ns	1.5ns	50%-70%
1.9V	1.47ns to 1.5ns	1.46ns	70%-90%

Table 5.3: *Shmoo Plot Summary for GasP S2P FIFO: The table compares DQS cycle time for width mismatch and without width mismatch on GasP S2P FIFO*

Figure 5.8: *Shmoo Plot of Micropipeline implementation of S2P FIFO*

Supply Voltage	DQS Cycle Time with Width variation for Yield of 85%-100%	DQS Cycle Time without Width Variation	Yield for DQS Cycle Time without Width Variation
1.7V	1.46ns to 1.5ns	1.46ns	50%-70%
1.8V	1.41ns to 1.43ns	1.35ns	50%-70%
1.9V	1.35ns to 1.38ns	1.32ns	70%-90%

Table 5.4: *Shmoo Plot Summary for Micropipeline S2P FIFO: The table compares DQS cycle time for width mismatch and without width mismatch on Micropipeline S2P FIFO*

5.3 Comparison of S2P and P2S FIFO

This section compares the data burst rate of two implementations of P2S and S2P FIFO. This section also compares the features of the self-timed data interface and the synchronous DDR3 data interface.

Figure 5.9 compares the serial to parallel data burst rate of the GasP and micropipeline implementations of the S2P FIFO. The red curve and blue curve represent GasP and micropipeline implementation of S2P FIFO, respectively. The blue and green curves are derived by connecting the green dots of the shmoo plots at minimum DQS cycle time. The micropipeline S2P FIFO is faster than the GasP S2P FIFO because it matches the DDR protocol and requires less logic.

Figure 5.9: *Graph for Data Burst Rate Comparison of Micropipeline and GasP Implementation of S2P FIFO: The data burst rate of the micropipeline implementation of S2P FIFO is better than the GasP implementation of S2P FIFO at different supply voltage.*

Figure 5.10 compares the parallel to serial data burst rate of the GasP and micropipeline implementations of the P2S FIFO. The red curve and blue curve represent GasP and micropipeline implementation of P2S FIFO, respectively. The blue and green curves are derived by connecting the green dots of the shmoo plots at minimum DQS cycle time. The micropipeline P2S FIFO is slower because loading parallel data into the FIFO requires more logic, and the storage elements are 3-input multiplexer.

Figure 5.10: *Graph for Data Burst Rate Comparison of Micropipeline and GasP Implementation of P2S FIFO: The data burst rate of the micropipeline and GasP implementation of P2S FIFO are almost same at different supply voltage.*

Figure 5.11 shows the self-timed DRAM data interface. The data burst rate of the self-timed DRAM data interface defined by the slowest of the P2S FIFO and S2P FIFO. The data burst rate of the self-timed interface is defined by the P2S FIFO and the S2P FIFO combination.

Figure 5.11: *Self-Timed DRAM Data Interface: The data burst rate of the self-timed data interface is maximum when the P2S FIFO is implemented using GasP and the S2P FIFO is implemented using micropipeline.*

Table 5.5 shows the data burst rate of the self-timed data interface with different combinations of P2S and S2P FIFO. The data burst rate of the self-timed interface is maximum when the P2S FIFO is implemented using GasP and the S2P FIFO is implemented using micropipeline.

Self-Timed DRAM	Processing Unit	Data burst Rate
P2S GasP	S2P Micropipeline	1.5ns
P2S GasP	S2P GasP	1.57ns
P2S Micropipeline	S2P Micropipeline	1.53ns
P2S Micropipeline	S2P GasP	1.57ns

Table 5.5: *Data burst Rate of the Self-Timed Data Interface: The table records the data burst rate of the self-timed data interface with different combination of P2S and S2P FIFO.*

Table 5.6 compares the features of the self-timed data interface with present synchronous DDR3 interface [1]. The synchronous DDR3 interface has better data burst rate than the self-timed data interface, and the synchronous DDR3 data interface operates at less supply voltage. But the simulation results of the self-timed data interface are for 180nm CMOS process technology and the features of the synchronous DDR3 interface are for 90nm CMOS process technology. The circuits simulated in 90nm CMOS process technology have higher current drives, reduced gate delays, and operates at lower supply voltage than the circuits in 180nm CMOS process technology.

Features	Self-Timed Data Interface	Synchronous DDR3 Interface
CMOS Process Technology	180nm	90nm
Supply Voltage	1.8V	1.5V
DQS Cycle Time	1.5ns	1.25ns to 1.87 ns
Communication Protocol	Two-Phase Handshake	Clock

Table 5.6: *Comparison of the Self-Timed Data Interface and the Synchronous DDR3 Interface*

Conclusions

This thesis introduces the data interface for the self-timed DRAM. The self-timed DRAM data interface uses the two-phase handshake protocol to transfer data to and from the processing unit. The self-timed data interface eliminates the need for clock for communication. The self-timed data interface also eliminates the need for timing circuits like a delay locked loop (DLL) or phase locked loop (PLL) for synchronization. The absence of DLL saves chip area.

The self-timed DRAM data interface takes advantage of the geometric proximity. After receiving a command the DRAM array requires time to fetch the data. This time is termed as the *latency*. The latency of the DRAM varies depending on the data location from the data interface. If the data are in a bank nearer to the data interface then the latency of the DRAM is less. The two-phase handshake protocol communicates with the processing unit when the data are ready for delivery. This allows the self-timed DRAM data interface to treat latency as a variable time, which depends on the data location from the data interface.

6.1 Future Work

The data transfer rate of the self-timed DRAM data interface depends on the parallel to serial FIFO (P2S) and serial to parallel FIFO (S2P). The S2P FIFO conversion rate is faster than the P2S FIFO. The data transfer rate of the self-timed DRAM data interface is defined by the P2S FIFO, as its conversion rate is less than the S2P FIFO. The data transfer rate of the P2S FIFO can be increased

by improving the conversion rate of the P2S FIFO.

The self-timed data interface is a part of a larger self-timed DRAM project. The next step is to implement the command interface for the self-timed DRAM. The self-timed DRAM command interface will allow the processing unit to issue successive read commands or write commands as rapidly as the self-timed DRAM can accept the commands. The self-timed command interface will also allow the processing unit to issue more complex commands to the self-timed DRAM. For example the processing unit can issue a command to the self-timed DRAM to read 1024 bytes of data starting from a given address.

References

- [1] Micron DDR3 SDRAM. [Online]. Available: http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf
- [2] Double Data Rate. [Online]. Available: https://en.wikipedia.org/wiki/Double_data_rate
- [3] B. J. Brent Keeth, R. Jacob Baker and F. Lin, *DRAM Circuit Design: Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2008.
- [4] I. Sutherland and S. Fairbanks, “Gasp: a minimal fifo control,” in *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, 2001, pp. 46–53.
- [5] I. E. Sutherland, “Micropipelines,” *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989. [Online]. Available: <http://doi.acm.org/10.1145/63526.63532>
- [6] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, March 2012.
- [7] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” in *Design Automation Conference, 2003. Proceedings*, 2003, pp. 338–342.