FIFOs and Shared-Memory Designs for Telecommunications

Steven K. Knapp
Regional Sales Manager
Xilinx® Asia Pacific

Introduction

As the telecommunications market has grown, so has the number of various systems and protocols. Linking the different telecommunication systems together is a high value-added function. Usually, these various systems have different data transmission rates which require interfacing two asynchronous systems together. Designers need to buffer data between two systems running at different speeds.

First-In, First-Out memories (FIFOs), or other forms of shared memory, offer an easy solution to the data buffering problem. FIFOs automatically solve the timing problems associated with interfacing two asynchronous systems.

This paper demonstrates various applications of FIFOs and shared memories using programmable logic in real-world telecommunication applications. Some designs require only small buffers, a few bytes deeps. Others require deeper FIFOs, often 32 or more bytes deep. In some cases, systems require much larger FIFOs—often more than 64K bytes deep. And finally some systems require fully shared memories with multiple, simultaneous read and write ports.

The two types of programmable logic architectures used in these applications include Field Programmable Gate Arrays (FPGAs) and Erasable Programmable Logic Devices (EPLDs). While either architecture can implement most any function, the FPGAs are better at register- and I/O-intensive applications. EPLDs are better suited to complex control functions and high-speed state machines.

The FIFO and shared memory applications demonstrated include:

- **Small, Register-Based General-Purpose FIFO (pages 3 through 7)**

  This application describes a small 8x8 FIFO used to synchronize two systems with roughly similar data transmission rates. The design demonstrates how to build a small FIFO using a Xilinx XC3000A or XC3100 FPGA device. The maximum clock frequency for the FIFO is 42 Mhz when built in an XC3100-3 device which is fast enough for CEPT E3 protocol applications.

- **Larger, High-Speed FIFO Using On-Chip RAM (pages 8 through 12)**

  Larger FIFOs, up to 64 or more bytes deep, fit into the Xilinx XC4000 family FPGAs. The XC4000 family FPGAs contain on-chip RAM making FIFOs very efficient. This application shows a high-speed version of a FIFO used in video-conferencing applications. The example design is only 16-bytes deep. However, the techniques shown here can be expanded to deeper and wider FIFOs.

- **Very Large FIFO Using DRAM (pages 14 through 16)**

  Some applications, with widely different transmission rates, require much larger FIFOs. Usually, these FIFOs are too large to fit into a single programmable logic device.

  However, building a 1Mx1 or 128Kx8 FIFO is easy using a Xilinx XC3020A and a 1Mbit DRAM memory. The FPGA contains the FIFO controller circuitry while the DRAM becomes the FIFO memory. Even larger FIFOs are possible using larger DRAMs.

- **Four-Port Shared Memory Controller (pages 17 through 19)**

  Some applications require simultaneous memory access from different read and write ports. This is a common function in local area network (LAN) bridges where various protocols are connected together.

  The controller must arbitrate between the various, simultaneous requests to determine which protocol actually accesses the memory at any one time. The arbitration state machine logic can be quite complex, requiring a device capable of handling such complexity.

  This design uses a Xilinx XC7200A EPLD which supports up to 17 product terms per macrocell and offers predictable 60 Mhz performance with 100% routing and utilization.

Why Use Programmable Logic?

Many very high volume telecommunication applications require very low-cost logic. These are generally long-established functions like telephone handsets, etc. Chip-sets are usually available for these applications, or the volumes are high enough to justify an ASIC solution.

However, many high-value applications are in new and emerging telecommunications markets. These functions are generally require flexibility as standards emerge. For example, the interface between the existing telephone infrastructure and new carriers like SDH and new networking structures like Asynchronous Transfer Mode (ATM) require adaptability as standards change and solidify.

The benefits of programmable logic in telecommunications applications include:

- **High density logic.** Up to 25,000 gates on a single programmable device.
- **Plenty of flip-flops** for building data registers, shift registers, FIFOs, state machines, and counters. One programmable logic device may contain thousands of flip-flops.
- **High speed logic.** Today's programmable logic can operate at over 150 Mhz for some functions. Most telecommunications applications, including high-speed functions like SONET and SDH, can now use programmable logic.
- **Easy modifications.** Make design changes without incurring additional cost. There are not any Non-Recurring Engineering (NRE) charges as there are with ASICs. This reduces the development costs for a project.
- **Faster product development.** Make modifications quickly, in only a few minutes to hours. Even field hardware updates are possible with reprogrammable logic devices.
- **Highly reliable.** The high level of testability gives programmable logic extremely high device reliability. This is especially true for reprogrammable devices like SRAM-based FPGAs.
- **Low power operation.** FPGAs offer low operating power—down to 10µA in standby mode. EPLDs are much lower power than the PAL®s that they replace.

Conclusions

Interfacing different telecommunications systems together is a high value-added portion of the market. Such systems require FIFOs and shared memories to resolve asynchronous timing problems. Programmable logic provides simple and reliable solutions for building FIFO circuits. In addition, programmable logic offers high-density, cost-effective, flexible solutions for many telecommunications applications.

Designs on Demand

All four applications are available on diskette as completed designs. Most of the designs are drawn using the VIEWlogic® VIEWdraw® schematic editor. PLUSASM™ equation files are available for the multi-port shared memory controller built using an EPLD.

Please contact your local Xilinx representative for a copy or contact the Xilinx Asia Pacific office at:

Xilinx Asia Pacific
Unit 2308-2319, Tower 1
Metroplaza
Hing Fong Road
Kwai Fong, N.T.
HONG KONG

852-410-2717 (phone)
852-418-1600 (FAX)
hongkong@xilinx.com (Internet)

Register-Based FIFO

XAPP 005.002 🖭

Application Note By BERNIE NEW AND WOLFGANG HÖFLICH

Summary

While XC3000-series LCA devices do not provide RAM, it is possible to construct small register-based FIFOs. A basic synchronous FIFO requires one CLB for each two bits of FIFO capacity, plus one CLB for each word in the FIFO. Optional asynchronous input and output circuits are provided. Design files are available for two implementations of this design. The fastest of the two implementations uses a constraints file to achieve better placement.

Specifications		***Xilinx Family***
Size	8 x 8 Bits	XC3000/XC3100
Maximum Clock Frequency XC3100-3	42 MHz	
Number of CLBs	40	

Introduction

In the absence of RAM, XC3000 FIFOs must be constructed with registers. Using both flip-flops, one CLB is required for each two bits of FIFO capacity. For a synchronous FIFO, an additional one CLB per word is required for control. Thus an 8-word by 8-bit FIFO can be implemented in 40 CLBs. Speed is a function of depth, with an 8-word FIFO able to achieve speeds of up to 42 MHz.

Asynchronous inputs and outputs may be added if desired. Each of these adds n/2 CLBs for an n-bit wide FIFO, plus a few additional CLBs for control logic. Typically, asynchronous inputs and outputs operate more slowly because of the handshake required for synchronization. Where burst input or output speed is required for data transfer, the FIFO should be operated in synchronism with the high-speed port.

The basic designs shown use simple flags that permit the input and output of single words. For block transfers, flags could be generated for signaling the availability of a block of data or space for a block of data.

Synchronous FIFOs

The basic FIFO design, shown in Figure 1, comprises a broadside shift register; each word has a separate shift enable. A control flip-flop, associated with each word, contains a valid flag that is shifted with the data. The shift-control logic uses these valid flags to generate shift enables and control the flow of data through the FIFO.

Whenever a register does not contain valid data, shift is enabled for that register, and for all the registers upstream from it. This causes data to continuously shift through the FIFO, with valid words backing-up at the output. They remain there until a POP command enables the shift in all the registers in the FIFO. Invalid data is not retained.

Figure 2 shows the detail of the FIFO. For simplicity, only two data bits are shown (the top two rows of flip-flops); all other data bits are identical. The bottom row of flip-flops contains the valid bits. The shift control logic is the chain of OR gates; a column of flip-flops is enabled if its valid bit, or any valid bit to the right, is not asserted.

The POP command acts like an additional active-Low valid bit, which is to the right of all the columns in the FIFO. When it is High, all the registers shift. If the second to last register contains valid data, this is shifted into the last register, and the VALID flag remains High. Otherwise, invalid data is shifted into the last register, and the VALID flag goes Low. The last register continues shifting until it receives valid data, when the VALID flag goes High.

Data can only be written into the FIFO if the first register contains invalid data or valid data that is about to be shifted out. This condition is signaled by the RDY flag, that is also the shift enable for the first register. Conse-

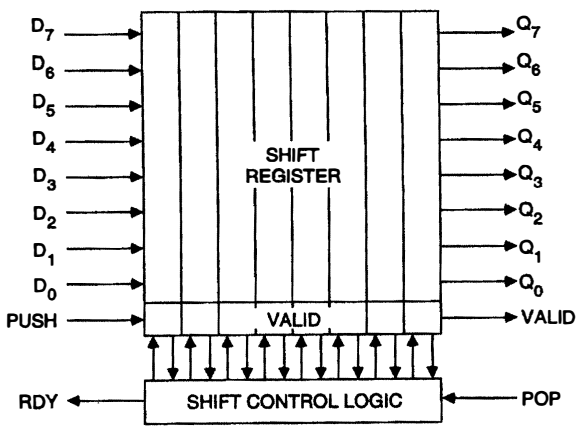

Figure 1. 8-Word x 8-Bit Synchronous FIFO (40 CLBs)

🖭 Supporting design files are available on the Xilinx

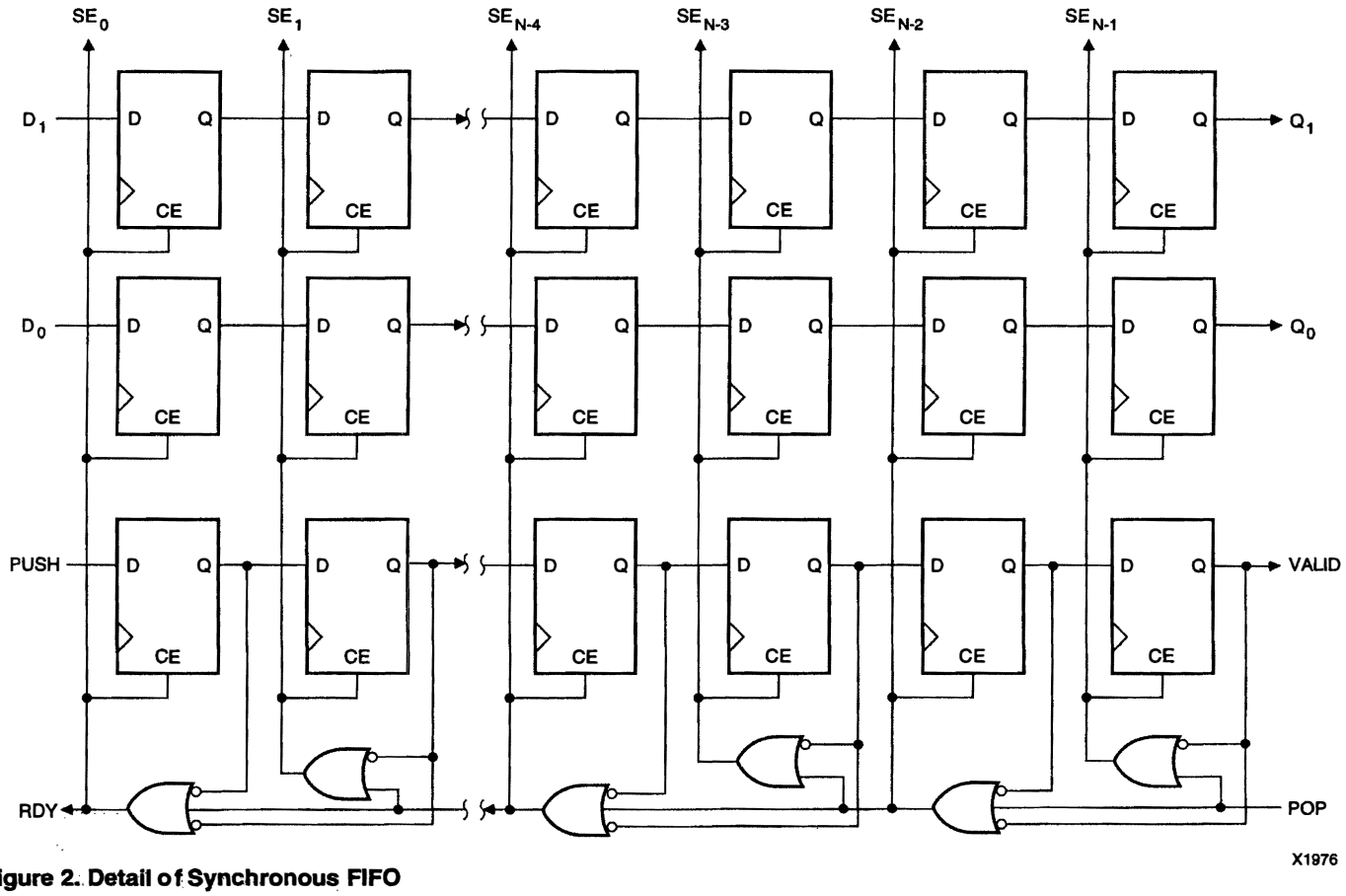

Figure 2. Detail of Synchronous FIFO

quently, data is always being shifted in when the FIFO is ready. The function of PUSH is simply to identify the data being shifted in as valid, so that it is retained in the FIFO.

In the diagram, the CLB clock enable (CE) is used as shift enable. When combining pairs of flip-flops into CLBs, CE can only be used if adjacent bits of the same register are combined. If it is more convenient, bits of equal weight from adjacent registers may be combined. In this case, function generators must be used to implement shift enable. This entails a simple 2-input multiplexer that selects input data when shift is enabled, and selects existing data from the flip-flop when it is not enabled.

The speed of the FIFO is determined by the ripple-OR time of the shift-control logic, and the distribution and set-up times of the shift-enable signals. This defines the set-up time for the POP command. The settling time for the shift-control logic is one CLB delay per two words of FIFO depth. Longlines should be used to distribute the shift-enable signals.

Asynchronous Input Stage

Asynchronous data may be entered into the FIFO using the circuit shown in Figure 3. An additional input holding register is provided to facilitate edge-triggered input. If appropriate, this can be implemented in IOB registers.

Data may only be entered when the RDY flag signals that the input register is available to accept it. The input clock (PUSH) also asserts the PUSH INP signal which removes the RDY flag. On the next internal clock, PUSH INT is asserted and PUSH INP cleared. When shift is enabled into the first register of the FIFO, data is transferred out of the holding register, PUSH INT is cleared and RDY is re-asserted.

If data is being input from a synchronous system that is not synchronized to the FIFO internal clock, the circuit shown in Figure 4 should be used. Again, an input holding register is provided. However, it is enabled by PUSH, instead of being clocked by it (an IOB register cannot be used). As before, PUSH causes PUSH INP to be asserted. Feedback around the flip-flop sustains PUSH INP until it is recognized by the internal clock, permitting the PUSH command to be removed after the one input clock.

The entry of data into the FIFO proceeds as in the previous scheme. RDY is registered to synchronize it to the input clock. The negative clock edge is used for this, so

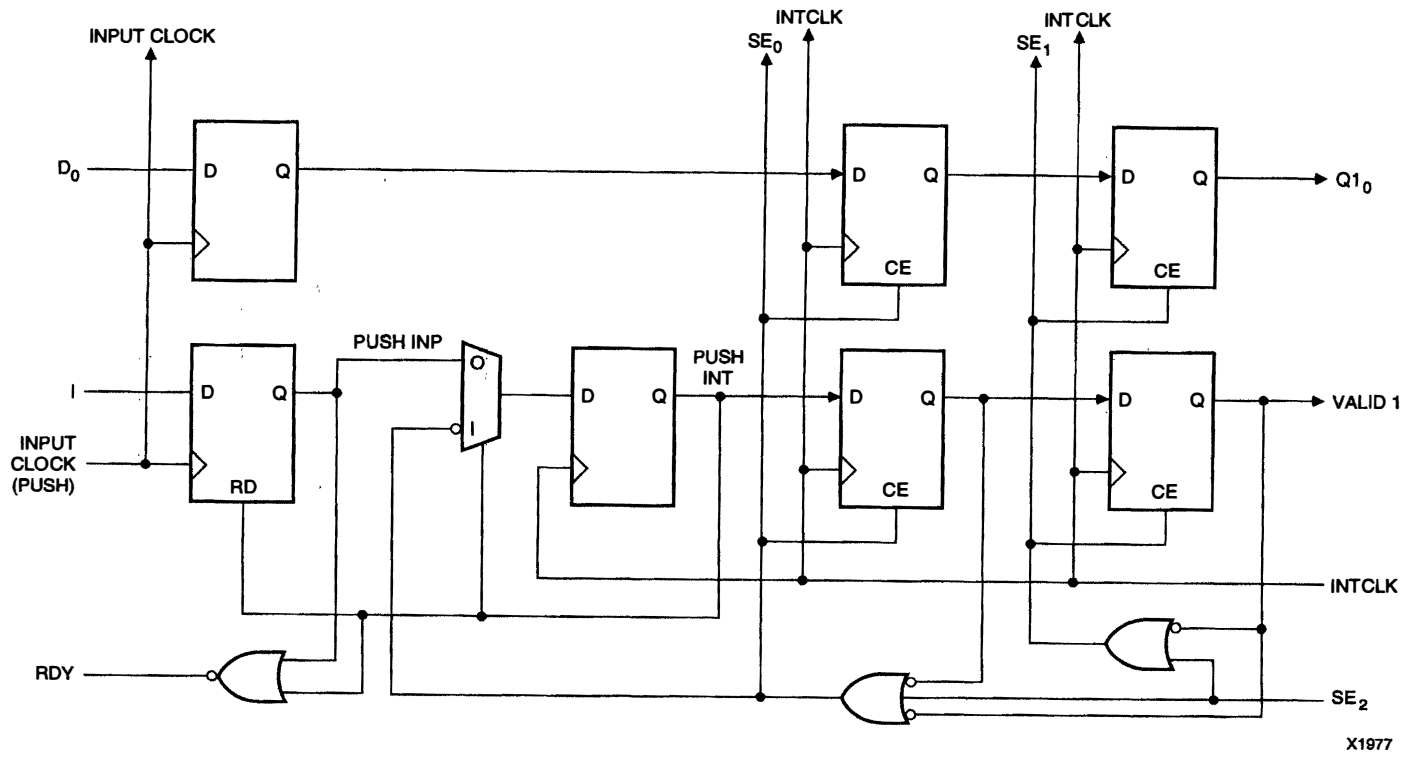

Figure 3. Asynchronous Input Stage

that, if the FIFO is sufficiently fast and is not full, the RDY flag will remain set, and data can be entered on successive input clocks. If the positive clock edge had been used, RDY would always be Low for at least one clock. At best, this would only permit data to be entered on alternate input clocks, no matter how slow.

Asynchronous Output Stage

The circuit shown in Figure 5 should be used, if an asynchronous output is required. For an immediate, edge-triggered output, a holding register is provided, which is clocked by the output clock (POP). IOB flip-flops may be used for this register.

The output register may only be clocked when the RDY flag signals that data is available in the last register of the FIFO. The output clock causes data to be transferred out of the FIFO, and asserts POP OUT. This removes the RDY flag. On the next internal clock, POP INT is asserted and POP OUT is cleared. POP INT is held, and the FIFO shifts, until the last register again contains valid data. It is then cleared, and the RDY flag is re-asserted.

If data is being output to a synchronous system that is not synchronized to the FIFO internal clock, the circuit shown in Figure 6 should be used. The output register, which cannot be implemented in IOBs, is enabled by POP. POP also causes POP OUT to be asserted. Feedback around the register sustains POP OUT until it is recognized by the internal clock, even if POP is removed and another output clock occurs.

The transfer of data out of the FIFO proceeds as in the previous scheme. RDY is synchronized with the negative edge of the output clock. As a result, data can be output on successive clocks if the FIFO is fast enough and data is available.

Implementation Notes

The obvious organization for the FIFO is as a rectangular array of CLBs, with the control logic in the bottom row. The flip-flops may be partitioned into CLBs in two ways. If adjacent bits of the same word are combined, the result is a FIFO that is twice as wide as it is tall (assuming equal numbers of bits and words).

Alternatively, two bits of equal rank from adjacent words may be combined. This gives a FIFO that is twice as tall as it is wide and is potentially faster. The critical path through the control logic passes through a chain of half as many gates as there are words. The tall, narrow organization allows these gates to be implemented in adjacent CLBs with zero-delay direct interconnects.

Both forms of the FIFO are available as macros, using CLBMAPs.

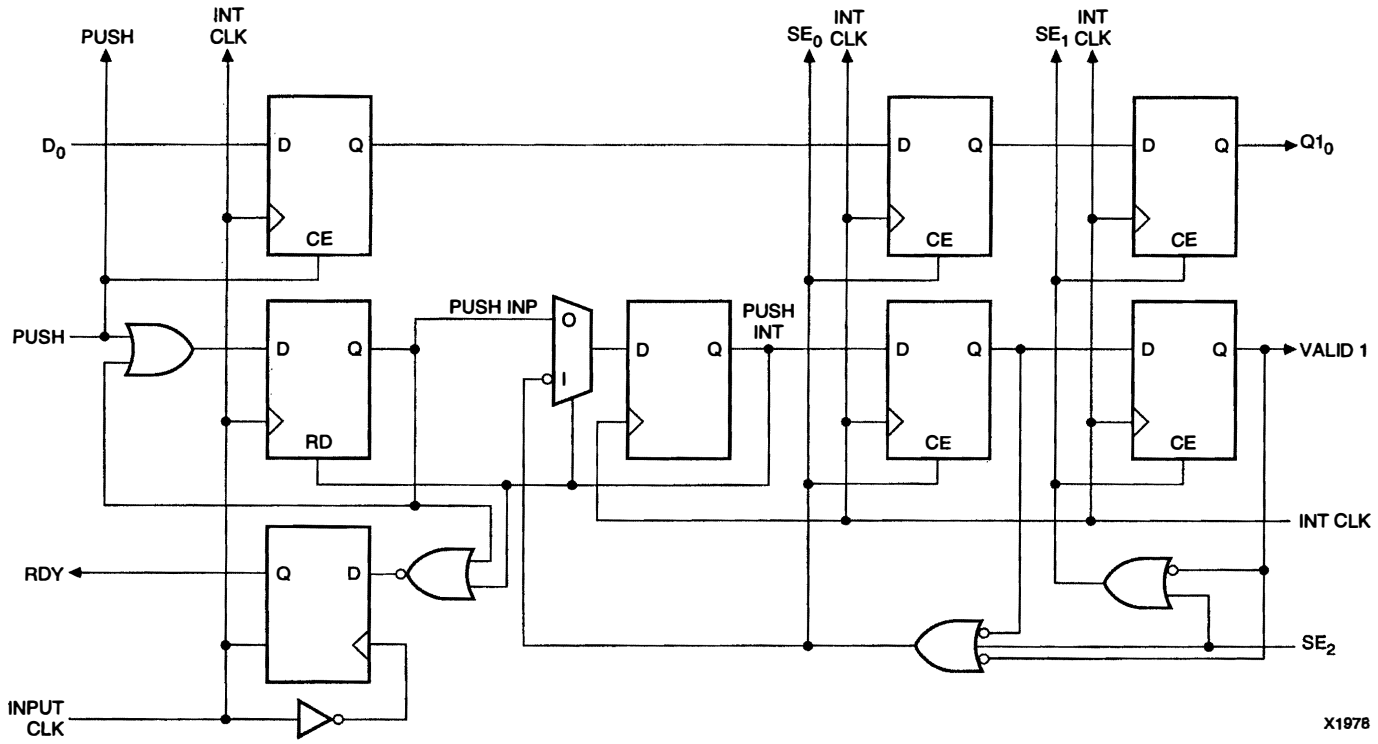

Figure 4. Asynchronous Input Stage (From Synchronous System)

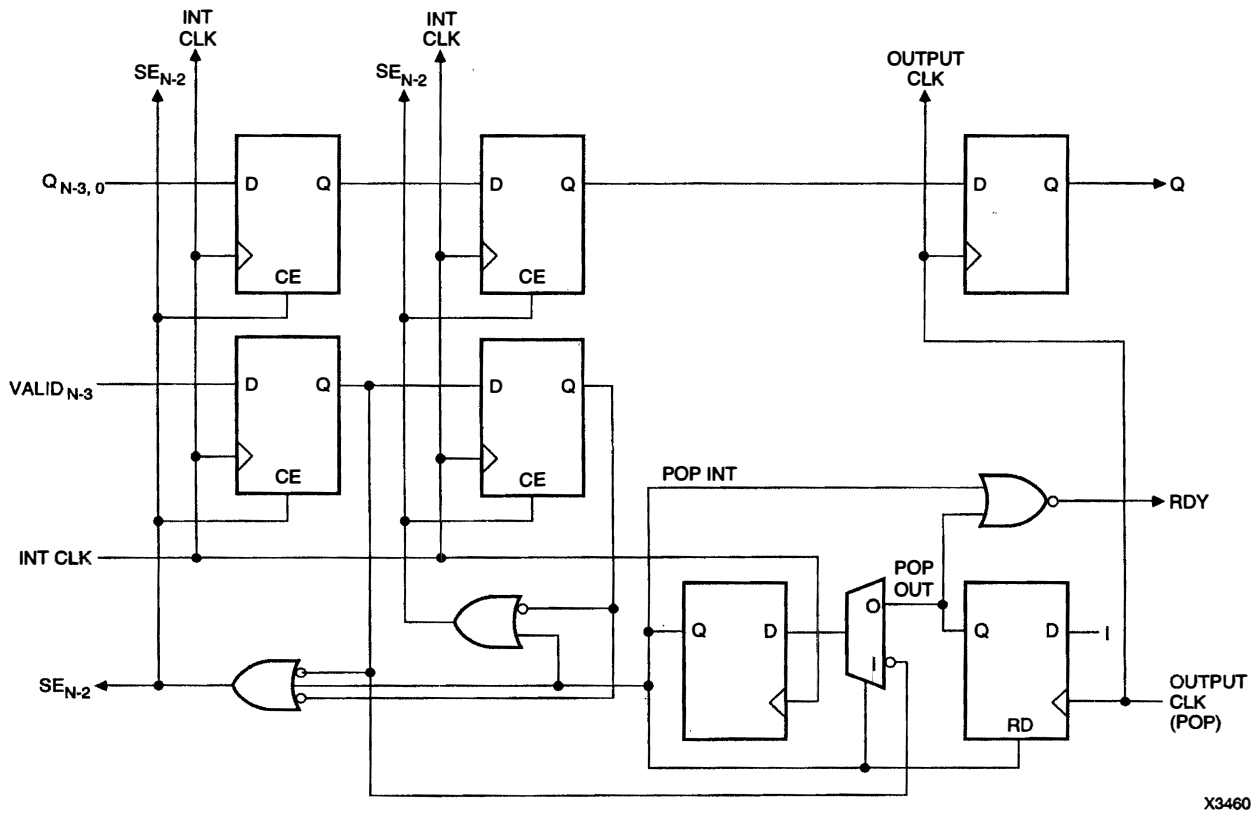

Figure 5. Asynchronous Output Stage

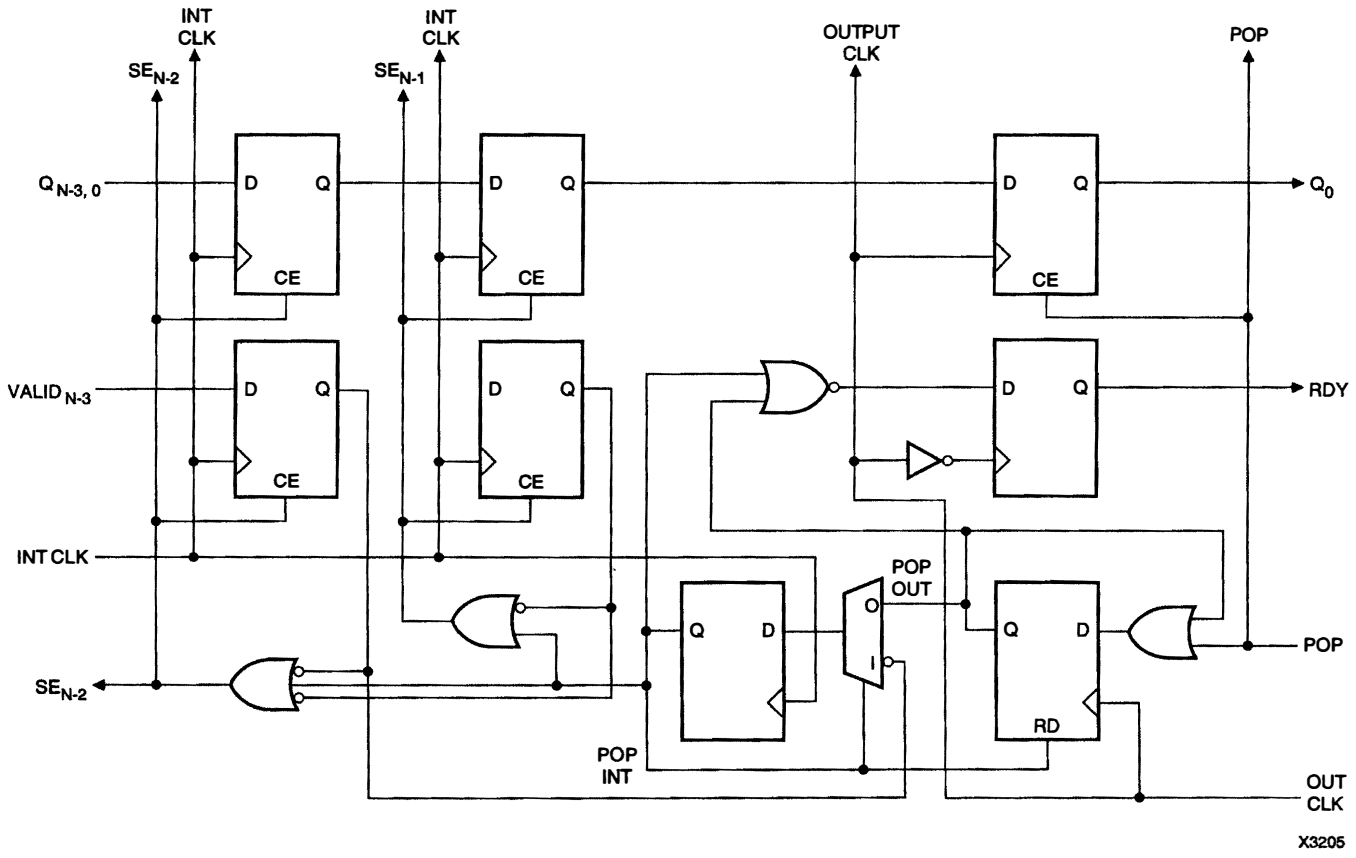

Figure 6. Asynchronous Output Stage (To Synchronous System)

High-Performance RAM-Based FIFO

XAPP 044.000

Application Note By BERNIE NEW

Summary

Two FIFO designs are described. In both cases, arbitration permits any RAM cycle to be a PUSH or a POP. XC4000 RAM performance is improved through read-modify-write operation, and the fastest clock required is at the RAM-cycle rate. The first design is expandable to any size FIFO, while the second, faster design is restricted to 16 or 32 words.

Specifications

Maximum Clock Frequency (estimated for XC4000-5)

16 x 8-bit FIFO 40 MHz

LCA Family

XC4000

Demonstrates

RAM Operation

Introduction

The four components of a RAM-based FIFO are shown in Figure 1. To control the RAM, the address-logic block maintains two addresses, one for the current write location, where data is PUSHed, and one for the current read location, from which data is POPped. Following a PUSH or a POP, the corresponding address is incremented, causing data to be written and read sequentially.

The flag logic uses the read and write addresses to determine the status of the memory. If the memory contains no valid data, an EMPTY flag is created; a FULL flag is created when all memory locations are occupied.

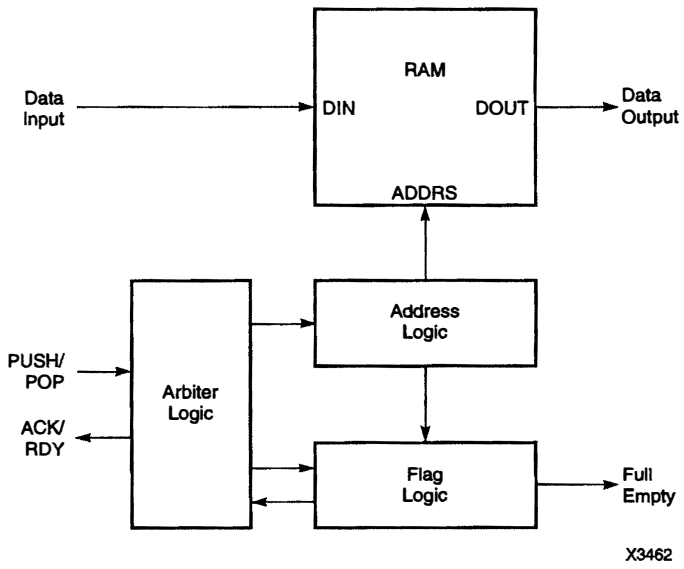

Figure 1. FIFO Block Diagram

The arbitration logic, for the most part, simply passes PUSH and POP requests to the RAM. Simultaneous PUSH and POP requests, however, must be resolved. The simplest schemes have a fixed priority, with either PUSH or POP being designated as the priority operation. If the priority operation is not possible because the RAM is full or empty, the priority should be overridden. Other schemes can alternate in priority between PUSH and POP, or favor one operation while its request persists.

Both FIFOs described depend on read-modify-write operation of the RAM, with Write Enable asserted every cycle. In the first design, a data multiplexer permits "non-write" cycles by rewriting existing data into the RAM. The same multiplexer provides bank selection for RAM expansion, permitting any size FIFO.

In the second design, the data multiplexer is omitted for faster operation. As a consequence, however, bank selection is not possible, and the RAM depth is limited to one CLB. Without a data multiplexer, only new data can be written in the RAM. During non-POP cycles, new data is always written to the current write address. If PUSH is asserted, the data is valid and the write address is incremented before the next write. If PUSH is not asserted, the address is not incremented, and the invalid data is subsequently overwritten. During a POP cycle, data read from the RAM is stored in a register, and the RAM location immediately overwritten with invalid data.

Operating Description

Expandable FIFO

Figure 2 shows the RAM and its address counters. A key element is the use of a BUFGS to drive the RAM Write Enable and clock the address register. The low skew of the global net ensures that the data and address hold

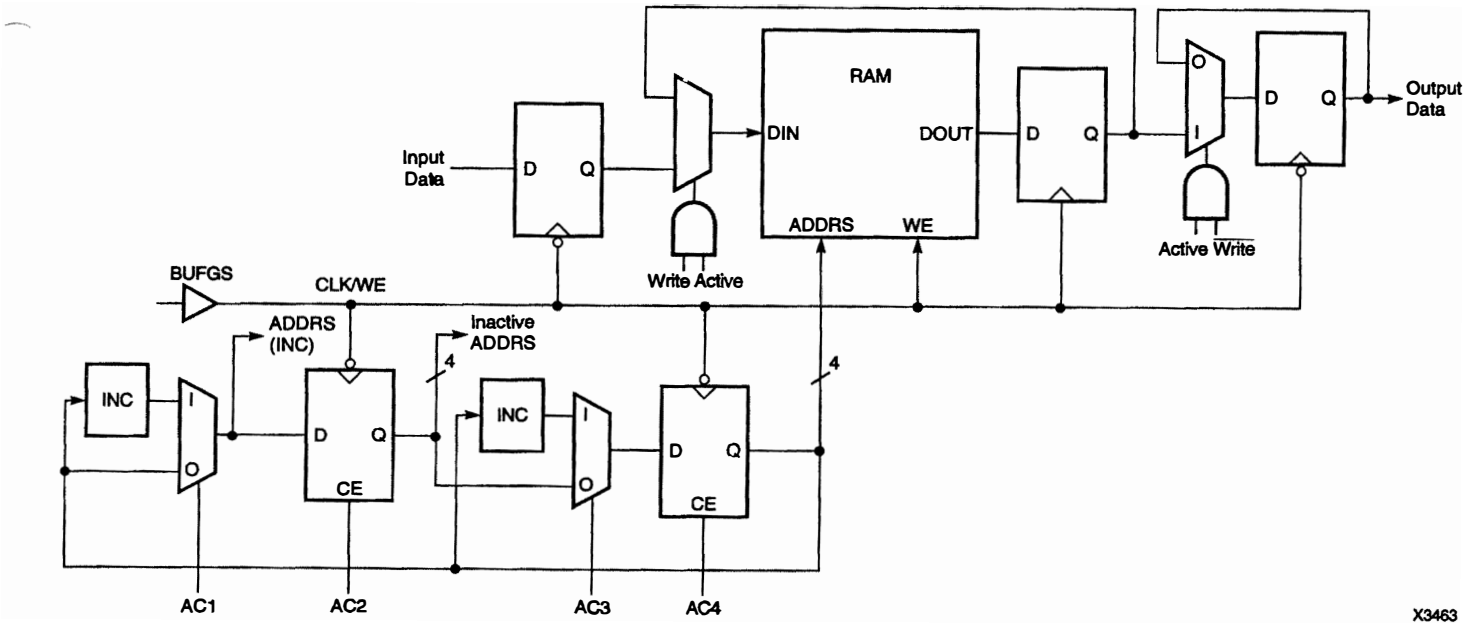

Figure 2. RAM and Address Logic

times are met. The RAM is written every clock cycle, and the multiplexer preceding the RAM determines whether new data is entered or the old data is re-entered. If it is necessary to expand the RAM, a bank select signal, derived from the address counter, can be ANDed with the Active signal to limit writing to a single bank of RAM. In addition, a read-data-select multiplexer must be provided.

The address-generation logic is shown in the lower portion of Figure 2. The right-hand register contains the address currently being used by the RAM, the write address during a PUSH and the read address during a POP. The left-hand register contains the address that is not being used.

The address-logic instruction set, Table 1, permits the active address to be incremented and remain active for successive PUSHes or POPs, or be incremented and become inactive, for a PUSH followed by a POP, or vice versa. It also permits the addresses to remain unchanged or simply interchanged.

FULL and EMPTY flags are generated in the flag logic, Figure 3. Flags can only be asserted or de-asserted during active PUSH or POP cycles, and both are triggered by the incremented active RAM address becoming equal to the inactive address. If this equality occurs during a PUSH cycle, the new write address contains the next data to be POPed, and the FIFO is full. If equality is reached during a POP, the FIFO is empty since the new read address is waiting to be written in the next PUSH operation.

Table 1. Address Logic Instruction Set

Current Op R/W	Act/Inact	Next Op R/W	AC 1 2 3 4
R	Active	R	1 0 1 1
R	Inactive	R	X 0 X 0
W	Active	R	1 1 0 1
W	Inactive	R	0 1 0 1
R	Active	W	1 1 0 1
R	Inactive	W	0 1 0 1
W	Active	W	1 0 1 1
W	Inactive	W	X 0 X 0

X3464

Consequently, the flags can be generated by gating the comparator output with the Write signal and registering it during active RAM cycles. The address-logic instruction set is constructed such that the inactive address and the incremented active address are always available to the comparator. The flags clear on the next active RAM cycle, when the addresses become non-equal. For correct operation, this cycle must be a PUSH if the FIFO is empty or a POP if it is full.

Figure 4 shows a simple arbitration circuit. The PUSH and POP inputs control a multiplexer that determines the operation in the next RAM cycle. If PUSH only is asserted, the next cycle is a write. If POP only is asserted, it is a read

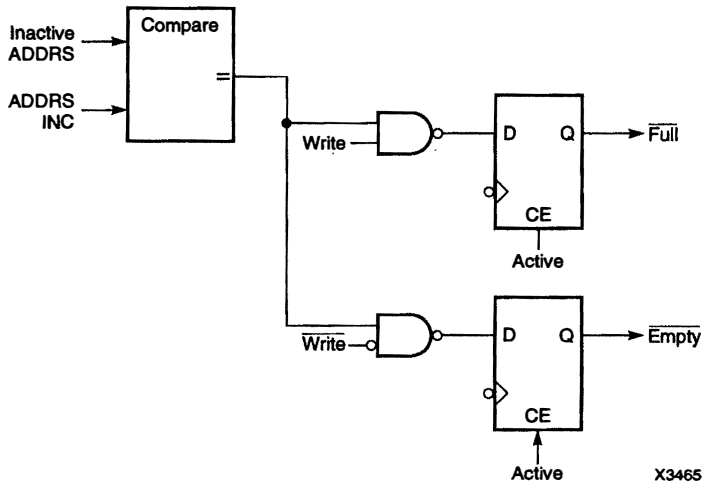

Figure 3. Flag Logic

cycle. If both are asserted together, the next operation in the next RAM cycle is determined by a user-defined Priority signal. Several options for the Priority signal are discussed later.

If a PUSH or a POP is requested and the FIFO is not full or empty, respectively, the next cycle is declared Active. A write or a read occurs and the corresponding address is incremented. Otherwise, the cycle is inactive; no read or write occurs and the addresses remain unchanged. In an inactive cycle, the Write signal is de-asserted by default.

Two handshake signals are generated. ACK acknowledges that a PUSH request will be honored in the next RAM cycle. Input data is captured on the falling clock edge that starts the RAM cycle. RDY indicates that a POP request will be honored during the next RAM cycle. Output data is made available on the falling clock edge that ends the RAM cycle.

In deciding the next operation when both PUSH and POP are asserted, the most straightforward Priority functions simply default to one operation or the other. To always write, a logic High could be used, and to always read, a logic Low. In practice, however, this can lead to wasted cycle. For example, PUSH could win when the FIFO is full and the operation cannot be performed. A better choice is to use FULL as Priority to always select write unless the FIFO is full. Similarly, using EMPTY will cause POP to always win unless the FIFO is empty.

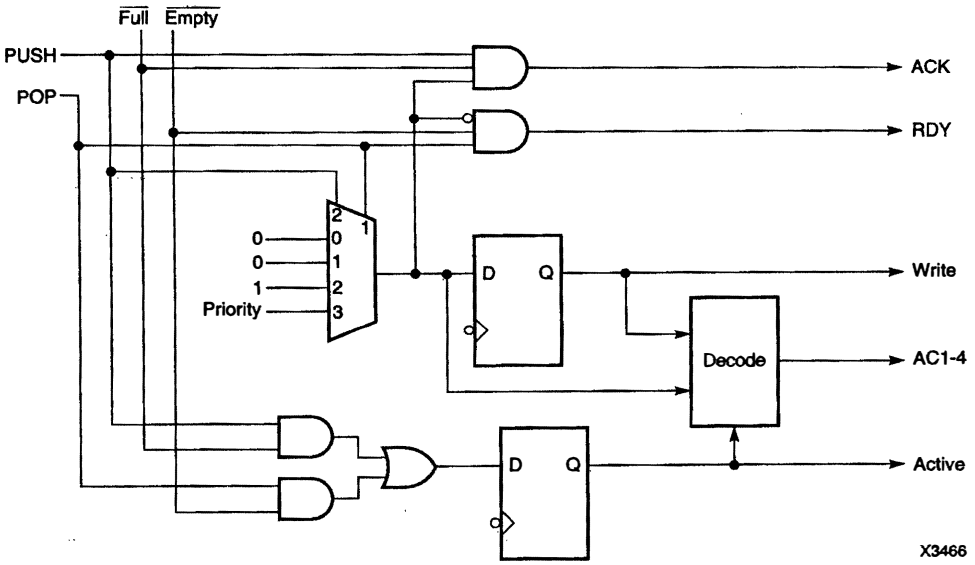

Figure 4. PUSH/POP Arbitration Logic

The above priorities are useful when receiving data from a burst source, such as a bus, or transmitting burst data. While burst is in progress, however, the other operation can be locked out for many cycles. If a more even resource allocation is required, Write can be used as Priority. In this case, requesting PUSH and POP continuously results in alternating reads and writes.

While the time to acquire the FIFO is reduced to no more than one cycle, the guaranteed peak PUSH/ POP rate is also reduced. In the limit, PUSH and POP may only operate at half the RAM cycle rate. The average data through the FIFO is unaffected, however. In the long term, it is obvious that no more than half the RAM cycles can be PUSHes. Attempting to achieve more will fail when the FIFO becomes full. Similarly, no more than half the cycles can be POPs, since the FIFO will become empty.

A third option permits both burst reads and burst writes, although either PUSH or POP may experience a long delay acquiring the FIFO if it is busy. Priority is connected to Write. As a result, the FIFO repeats its last operation whenever there is a conflict. A burst read or write will continue, and lock out the other operation until the burst is complete.

High-Performance FIFO

The RAM block for the high-performance FIFO is shown in Figure 5. In this design, the RAM has simple input and output registers. The input register captures data on the falling edge of the clock which, in addition, marks the start of a RAM cycle. Data is captured in the output register at the end of the read phase, when the clock goes high.

Prior to the start of the RAM cycle, a selection is made between the read and write addresses, and the selected address is registered when the RAM cycle starts. The read address is only selected when a POP is to be executed. Stable output data is retained from POP to POP, however, since the output register is only enabled during POPs. If the output data is required to change on the falling edge, an additional register must be used.

Since new data is written into the RAM every cycle, the read address cannot be selected during idle cycles; valid data waiting to be read would be destroyed by the write. Consequently, the write address is selected for both PUSH and idle cycles.

In the previous design, the RAM can be filled completely. When the FIFO is full, the next write address becomes equal to the next read address. This is not a problem, provided the location is read before it is overwritten. In the current design, there must always remain at least one unused location where invalid data is written during idle cycles.

When a PUSH occurs, the data written finally becomes valid, and the write address is incremented. During subsequent idle cycles, invalid data is written to the new write address. Overwriting of this address continues until the next PUSH.

As a consequence, a maximum of 15 words can be stored in the RAM. The FIFO can, however, store two more words in the input and output registers. The total storage capacity is 17 words.

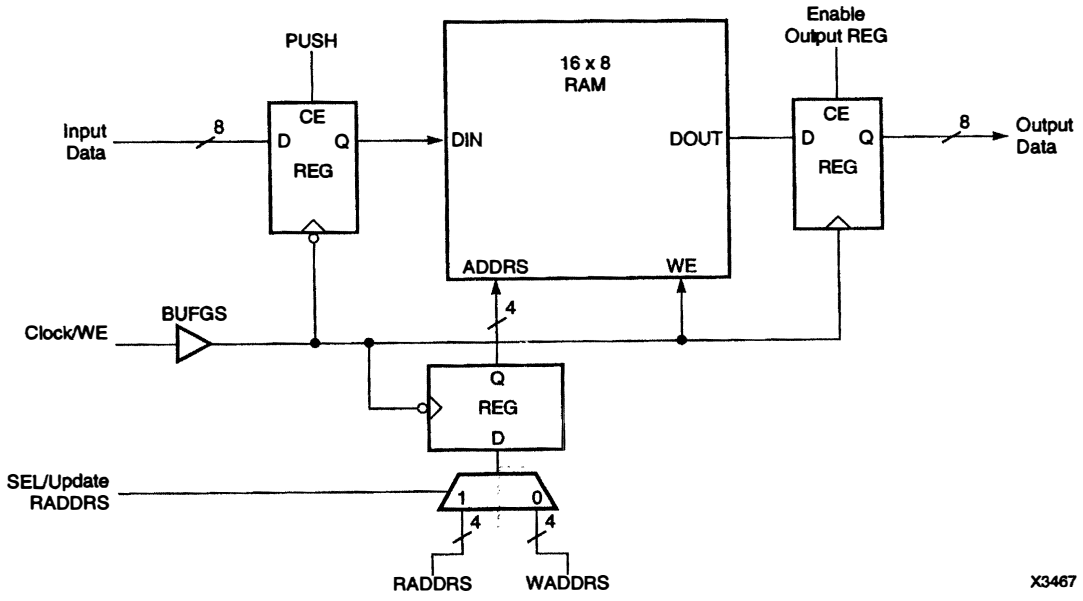

Figure 5. RAM Diagram

Figure 6 shows the address and flag logic for the FIFO. Modified 4-bit linear-feedback-shift-register (LFSR) counters are used. Conventionally, 4-bit LFSR counter use the XNOR of the last two shift-register bits as feedback the input. This results in a sequence that repeats every 15 clocks. The missing count, all-1s, can, however, be added to provide the count sequence shown in Table 2.

Normally, 1110 is followed by 0111, and 1111 would cause the counter to lock up. To include 1111 in the sequence, 111X is detected, and the shift-register input is inverted while this condition is met. Consequently, 1110 is followed by 1111 and the next count is 0111, since the input remains inverted. The remainder of the count sequence is unaffected.

A benefit of LFSR counters in this design is that adding one extra bit to the shift register permits access to two adjacent addresses, which, in turn, permits easy generation of the FULL flag. The current write address is available to the RAM, while the next write address is also available for comparison with the read address. When the next write address equals the read address the FIFO is full. The current write address is the sixteenth RAM location needed for invalid data writes during idle cycles. The FIFO is empty when the current read address becomes equal to the current write address, which is yet to be written with valid data.

Table 2. Adder-Counter Sequence

Shift Register Input ↓
0 0 0 0
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
0 1 1 1
1 0 1 1
1 1 0 1
0 1 1 0
0 0 1 1
1 0 0 1
0 1 0 0
1 0 1 0
0 1 0 1
0 0 1 0
0 0 0 1
0 0 0 0
1 0 0 0
⋮

X5281

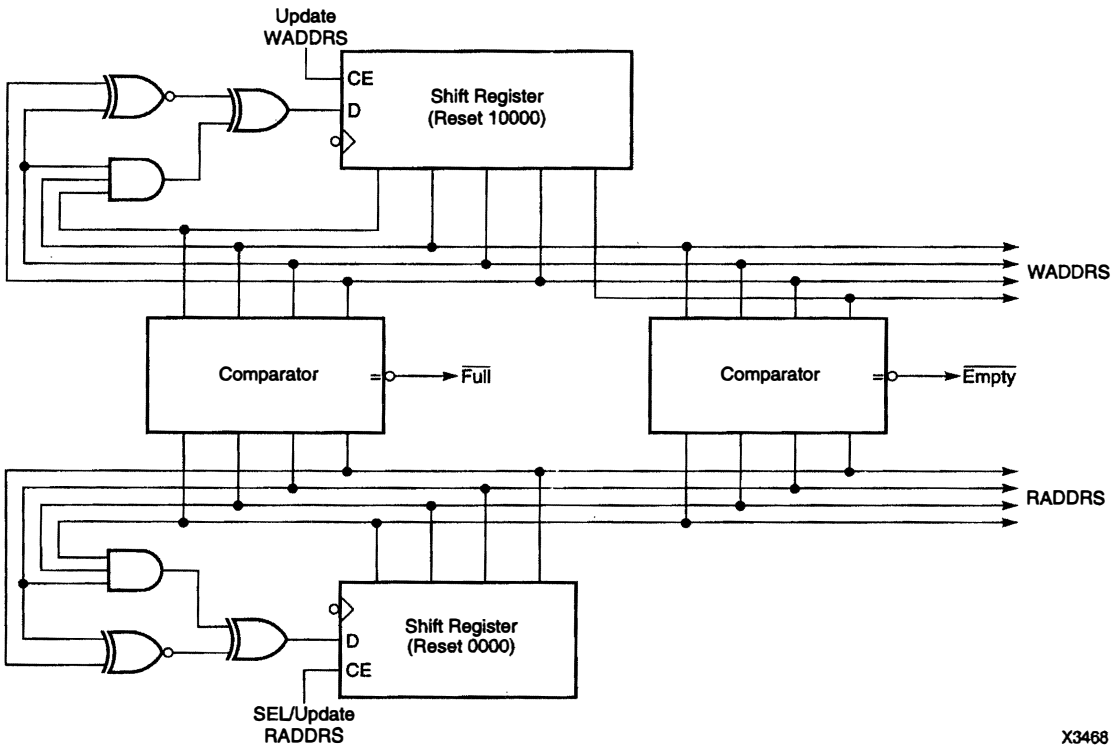

Figure 6. Address/Flag Logic

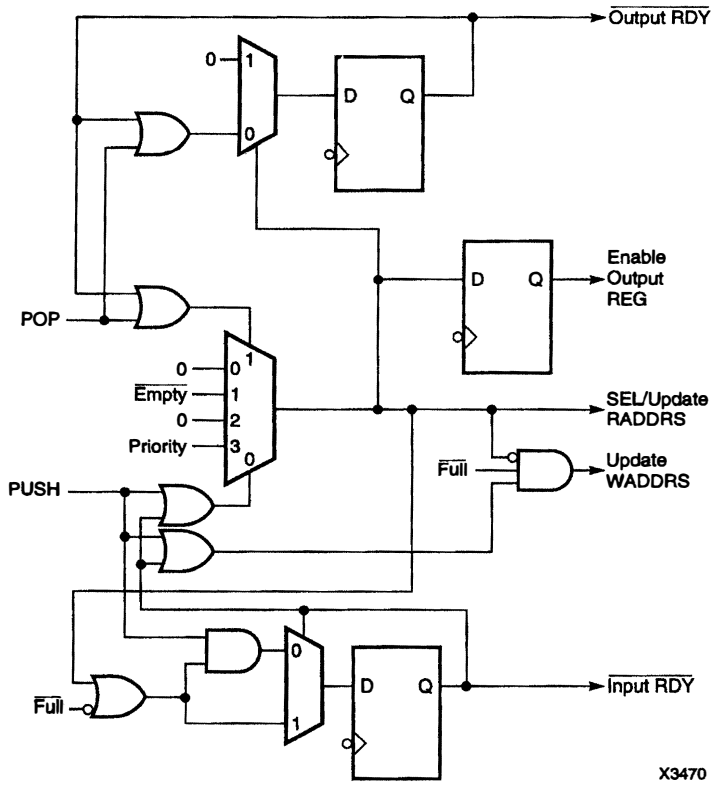

Figure 7. PUSH/POP Arbitration

The arbitration logic is shown in Figure 7. As in the previous design, the core of the arbiter is a multiplexer that selects the next operation according to the requests and the status of the RAM. The Priority input must be defined and implemented as discussed previously.

The output of the multiplexer is High when a read is to be performed. The Highfi causes the read address to be both selected for the RAM and simultaneously updated. Otherwise, the write address is selected. The write address is only updated, however, when a PUSH is requested and the UPDATE-WADDRS command is issued.

PUSH/POP requests and input data must set up to the falling edge of the clock, and POPed data becomes available on the subsequent rising edge. If a request cannot be serviced immediately, it is stored in one of two flip-flops, and a RDY is asserted on the falling clock edge at the start of the RAM cycle. If a request can be serviced, the corresponding RDY flag is never asserted.

When a PUSH is deferred, the input data is still captured in the input register, but it is not transferred to the RAM. In this case, RDY should suppress further PUSHes. The RDY flags are cleared at the start of the RAM cycle in which the request is serviced.

Megabit FIFO in Two Chips: One LCA Device and One DRAM

XAPP 030.000

Application Note By PETER ALFKE

Summary

This Application Note describes the use of an LCA device as an address controller that permits a standard DRAM to be used as deep FIFO.

Xilinx Family

XC3000/XC3100

Demonstrates

Non-linear Counters
Pseudo-random RAM Addressing

Introduction

A bit-serial FIFO buffer is a general-purpose tool to relieve system bottlenecks, e.g., in LANs, in communications, and in the interface between computers and peripherals. Small FIFOs are usually designed as asynchronous shift registers, but a larger FIFO with more than 256 locations is better implemented as a controller plus a two-port RAM, or as a controller plus a single-port RAM, either SRAM or DRAM.

SRAMs are fast and easy to use, but at least four times more expensive than DRAMs of equivalent size. Dynamic RAMs offer lower-cost data storage, but require complex timing and address multiplexing, which makes them unattractive in small designs. For FIFOs with more than 256K capacity, a DRAM offers the lowest cost solution, if the controller can be implemented in a compact and cost-effective way. An XC3020 Logic Cell Array can easily perform all the control and addressing functions with many gates left over for additional features.The XC3020 can be programmed to control one or more DRAMs for a FIFO of up to 16 megabytes, with data rates up to 16 Mbits per second serially or 16 Mbytes per second byte-parallel.

Logic Description

This FIFO DRAM controller comprises the following.

- Input/output buffer with synchronizing logic
- 20-bit Write pointer (counter)
- 20-bit Read pointer (counter)
- 20-bit full/empty comparator
- 10-bit address multiplexer
- Control and arbitration logic

Figure 1 is a block diagram of the FIFO Controller. The Write pointer defines the memory location where the incoming data is to be written, while the Read pointer defines the memory location where the next data can be read. The identity comparator between the address pointers signals when the FIFO is full or empty.

When the Write and Read pointers become identical as a result of a Write operation, the FIFO is full, and further Write operations must be prevented until data has been read out to create space in the memory. If the two pointers become identical as a result of a Read operation, the FIFO is empty and further Read operations must be prevented until new data has been written in. With a single-port RAM, Read and Write operations must be inherently sequential, and there is no danger of confusing the full and empty state, a problem that has plagued some two-port designs.

A straightforward design would use synchronous binary counters for the two pointers, but it is far more efficient to use linear feedback shift-register (LFSR) counters. Such counters require significantly less logic and are faster since they avoid the carry propagation delay inherent in binary counters. LFSR counters have two peculiarities: they count in a pseudo-random sequence, and they usually skip one state, i.e., a 20-bit LFSR counter repeats after 2^{20}-1 clock pulses. In a FIFO Controller, both these issues are irrelevant; the address sequence is arbitrary, provided both counters sequence identically.

The RAS/CAS multiplexing of the 20-bit address is performed without an explicit multiplexer. Every other bit of the shift-register counter is used to provide the 10-bit address. Before the incrementing shift, these bits are used as the Row address. After incrementing, they are used as the Column address. The Column address of any position is thus identical with the Row address of the following position, but since the binary sequence of a shift register counter is pseudo-random anyhow, this is not a problem.

The address generation logic is shown in Figure 2. With this design, two shift-register counter bits fit into one XC3000-series CLB, with the identity comparator using the combinatorial portion of the same CLB, Figure 3.

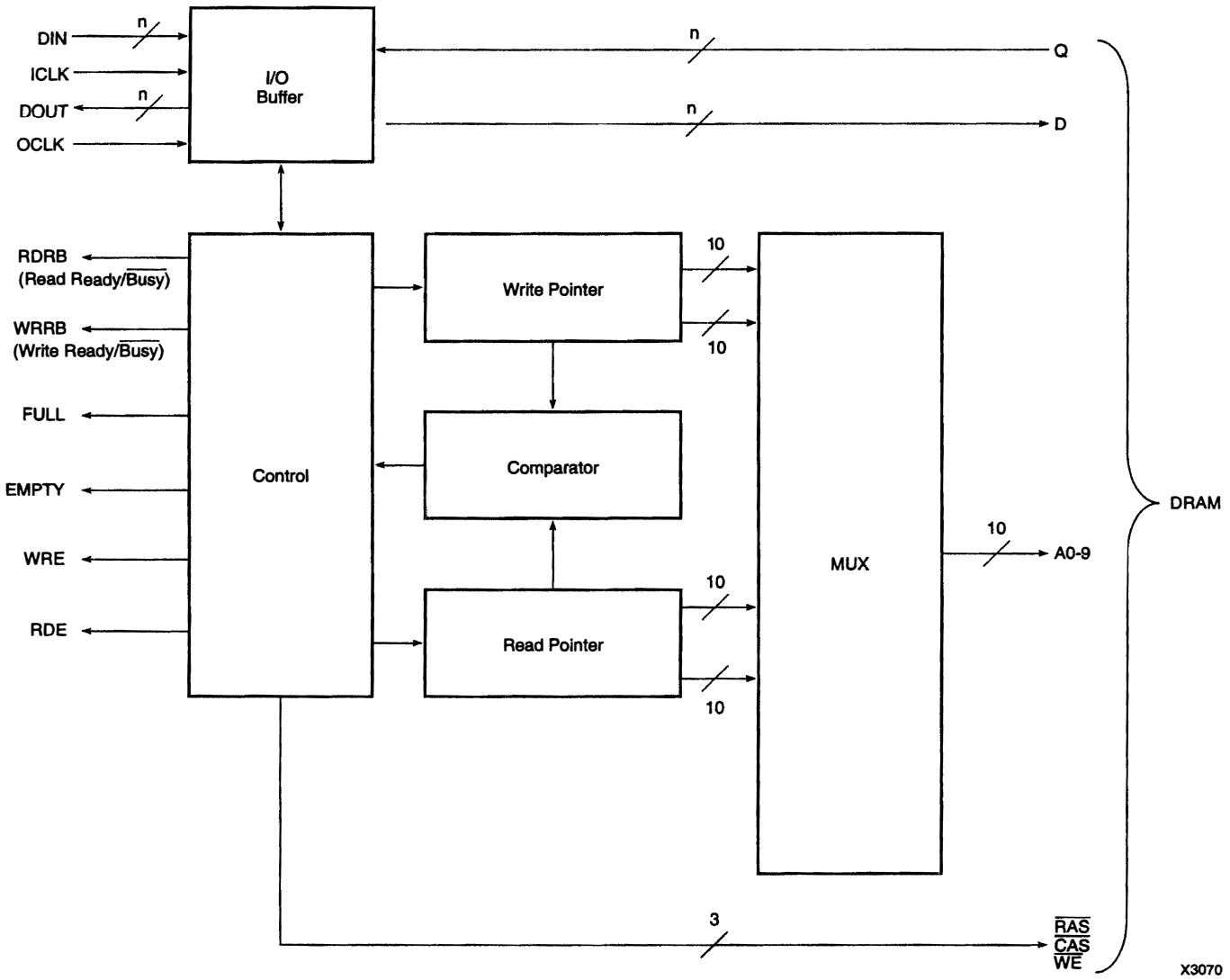

Figure 1. Megabit FIFO Controller in an XC3020

The FIFO controller permits the user to perform totally asynchronous Read and Write operations, while it synchronizes communication with the DRAM. The design takes advantage of the DRAM internal refresh counter by using CAS-before-RAS refresh/address strobes.

Both 20-bit pointers, plus their 20-bit identity comparator, plus the Row/Column multiplexer thus fit into only 20 CLBs; refresh timer and address multiplexer use another 10 CLBs and the data buffer plus control and arbitration logic take another 23 CLBs, for a total of 53, an easy fit in an XC3020.

This design can easily be modified for larger or smaller DRAMs. Other variations that might be considered are: multiple parallel bits, e.g., byte-parallel operation, interrupt-driven control, multiplexed data for multiple parallel-bit storage, and byte parallel storage with bit-serial I/O. This latter case requires special attention when the FIFO is emptied after a non-integer number of bytes has been entered, and requires direct communication between the input Serial-to-Parallel converter and the output Parallel-to-Serial converter.

This design is available from Xilinx. Call the Applications Hot Line 408-559-7778 or 1-800-255-7778.

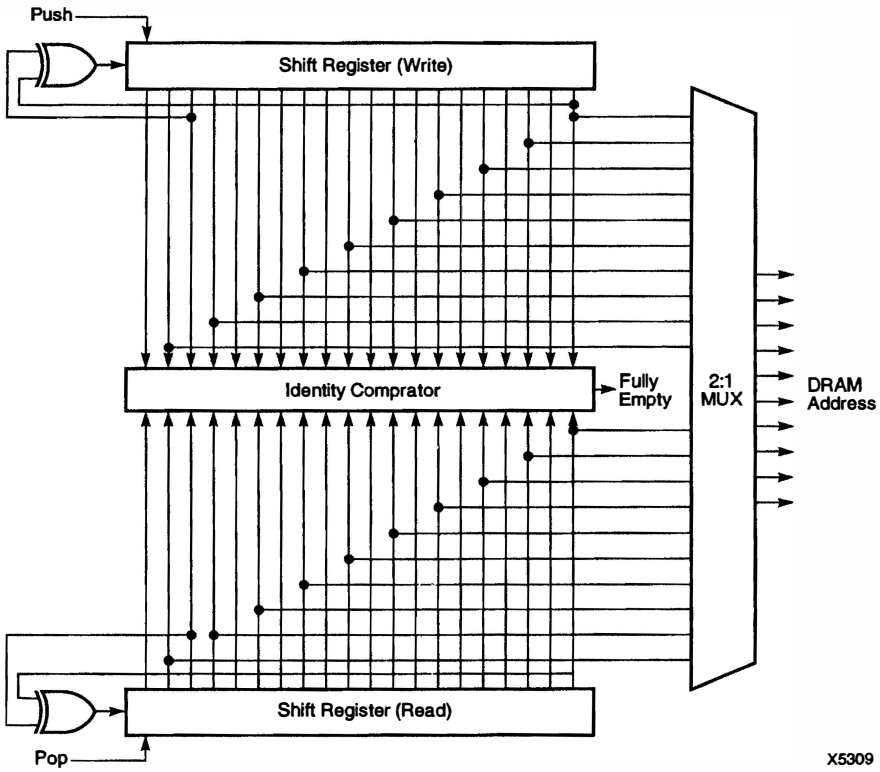

Figure 2. DRAM Address Generation

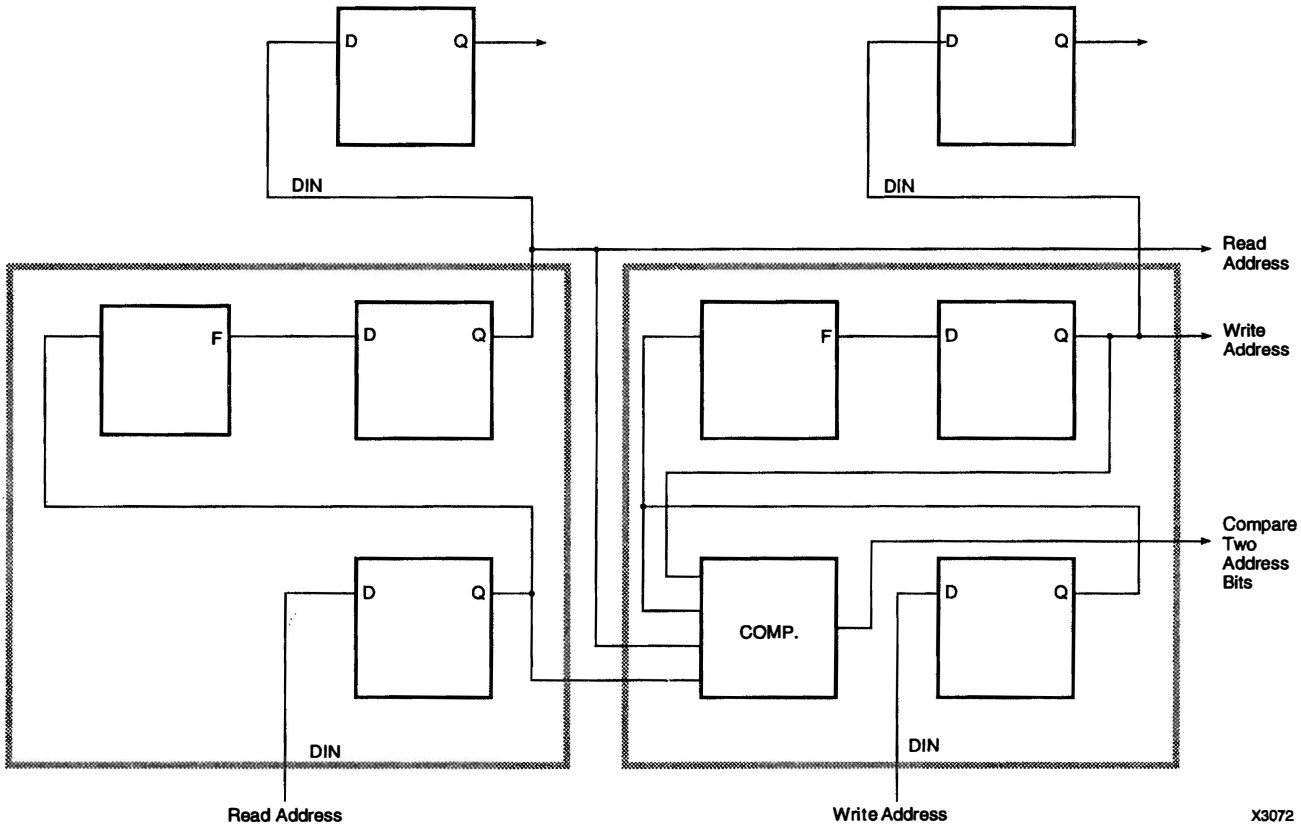

Figure 3. 2-Bit Slice of Two Counters and Comparator in Two CLBs

Four-Port DRAM Controller Operates at 60 MHz

XAPP 036.001

Application Note By JEFFREY GOLDBERG

Summary

This Application Note describes a high-performance DRAM controller implemented in a single Xilinx EPLD.

Xilinx Family

XC7200/XC7300

Demonstrates

High-speed State Machines

Introduction

Multi-port memory arrays are used in many applications, such as telecommunications, graphics and VME cards. Although these applications serve many different purposes, they share a common need: they must quickly and efficiently access a shared memory space through several different ports. The control logic must perform a complex arbitration function, yet must run at a high clock speed.

The XC7236A architecture is well suited for implementing the fast, complex state machines found in multi-port arbitration schemes. The XC7236A-16 can implement a quad-ported DRAM memory controller capable of arbitrating among four access requests in one 60-MHz clock cycle. This DRAM controller is capable of supporting 70-Mbyte/s burst transfers over a 32-bit bus, Figure 1.

The design uses 94% of the available Macrocells, yet runs at the maximum specified speed of the device. Familiar third-party tools reduce both the design effort and time, and XEPLD translator quickly compiles even the most complex designs.

Theory of Operation

The arbiter implements a round-robin algorithm, where the priorities for the four ports are arranged in circular order; the most recently served port is automatically assigned lowest priority. Each port can also lock the arbiter to retain ownership between back-to-back accesses. Such locking is necessary for semaphore reading and writing in multiprocessor systems.

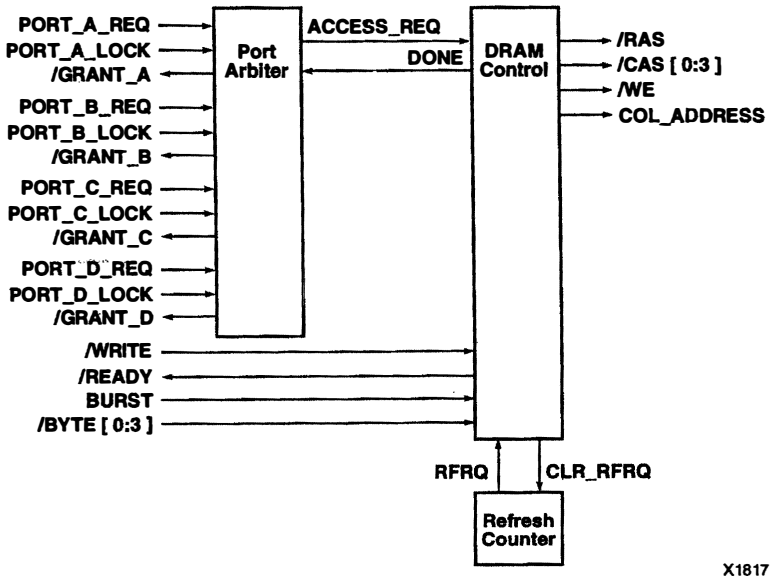

Figure 1. Quad-Port Memory Controller

The arbiter evaluates incoming access requests while it is in any of four idle states. The specific idle state depends on the last request, and determines the priority of the incoming requests. If the arbiter is not locked, it grants access to the highest priority request that is pending, and issues a memory-access request to the on-chip DRAM controller. During its transition to one of four port-access-active states, the arbiter asserts the grant signal to the appropriate port. The grant signals are used to enable the port control, address and data busses. The arbiter remains in its active state until the DRAM controller signals that it has completed the single or burst access.

The arbiter then goes to the idle state corresponding to the port that was just serviced, thus placing that port at the lowest priority level. If another access request is pending, the arbiter will issue another memory-access request to the DRAM controller. The data access will occur as soon as the DRAM controller has precharged the memory. The interaction between arbiter and DRAM controller is shown in Figure 2.

The DRAM controller also arbitrates between memory requests from the ports and refresh requests from the on-chip refresh counter, as can be seen in Figure 3. The address-multiplexer control line and the DRAM strobes are sequenced by the controller's state machine. They are enabled by the byte select and write enable outputs of the port.

The controller informs the port when there is valid data on the bus by asserting the READY output. If burst access is enabled, fast 3-clock memory accesses are performed until the port drops the burst request line. The controller then begins to precharge the memory, and asserts DONE to inform the port arbiter that the final memory access is completed.

Device Utilization

When implemented in PLDs, multi-port-arbiter state machines tend to be product-term intensive. The XC7236A is particularly well suited for such applications since each Macrocell can handle up to 17 product terms.

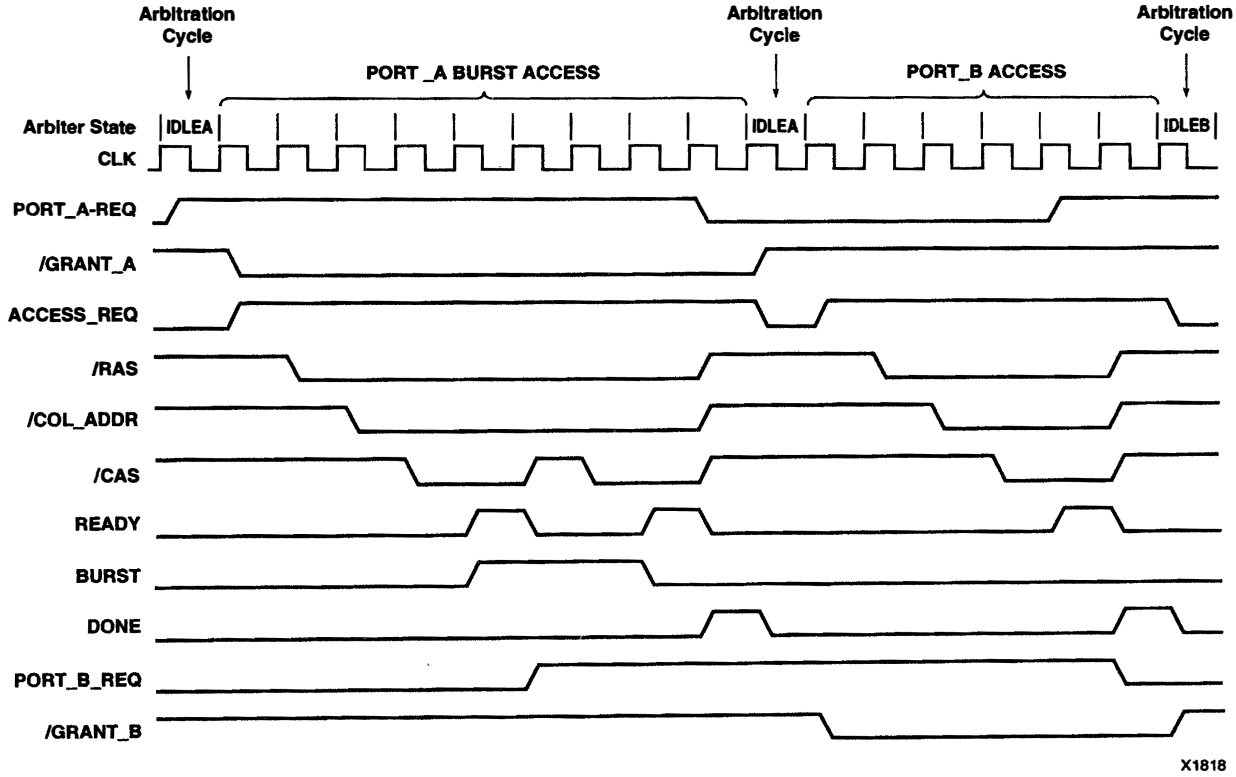

Figure 2. Quad-Port DRAM Controller Timing Diagram

Of the eight Macrocells required to implement the port arbiter, one Macrocell uses ten product terms, one uses nine terms, one uses eight terms; the remaining five Macrocells use seven product terms each. In total, 148 product terms, and 34 of the 36 Macrocells are used. The Macrocell XOR gates in the XC7236 significantly reduce the number of product terms used in the 10-bit refresh counter. In total, the DRAM controller occupies 94% of the XC7236A.

Design Methodology

The design lends itself very well to a modular behavioral description of its state machine. The ABEL 4 compiler was used to generate three Boolean equation files from high-level descriptions of the refresh counter, and the arbiter and DRAM controller state machines. A main PLUSASM file was then derived from the three equation files.

In this design, the main file only defines the external signals to and from the XC7236A. With this design approach, individual state machines and counters can be developed in a modular fashion, using the design tools most appropriate to each module. XEPLD software compiles the files in about five minutes, and generates a single file that can be downloaded into the device programmer.

Detailed design files are included with the XEPLD software, and are available from Xilinx Applications. They will soon be available from the Xilinx Technical Bulletin Board.

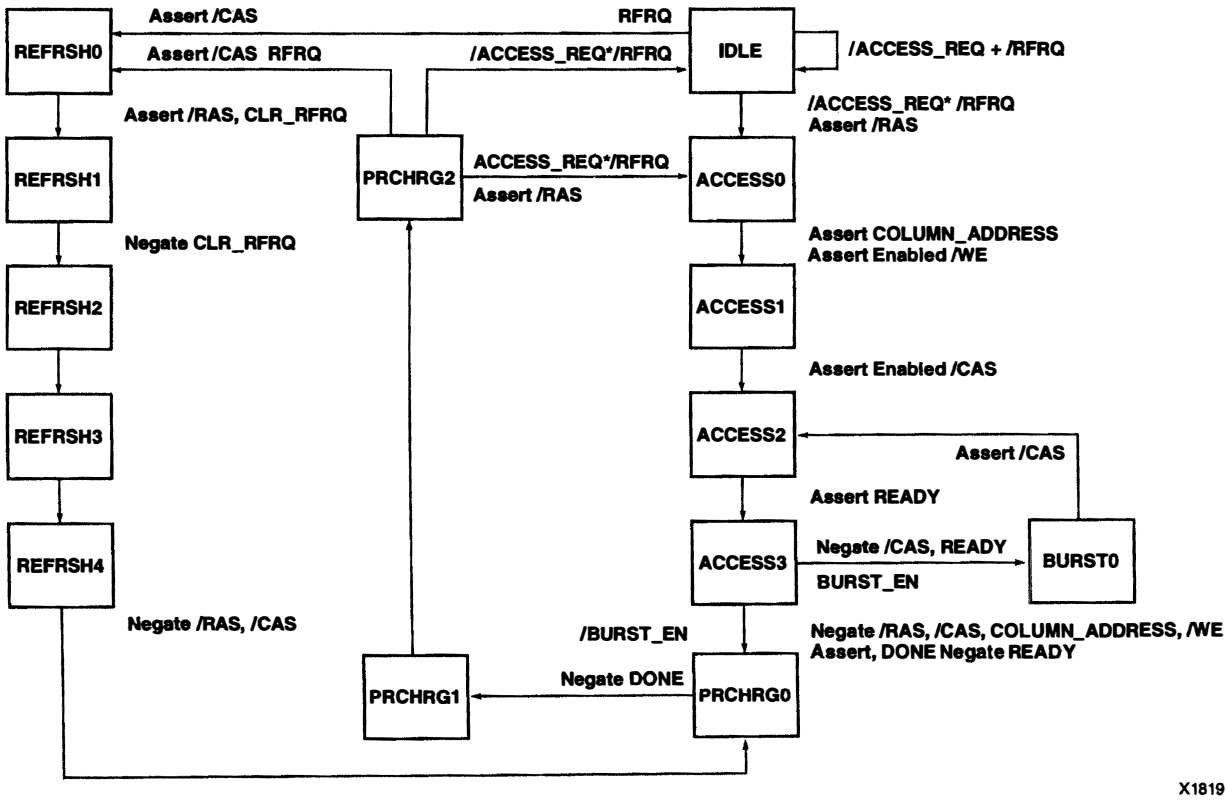

Figure 3. DRAM Controller State Diagram

The Programmable Logic Company℠

Unit 2308-2319, Tower 1, Metroplaza, Hing Fong Road, Kwai Fong, N.T. HONG KONG
(852) 410-2717 (phone) (852) 418-1600 (FAX) hongkong@xilinx.com (Internet)