XILINX®

Self-Addressing FIFO

Author: Nick Sawyer

XAPP291 (v1.2) May 27, 2003

Summary

The block memories in the Virtex™-II and Spartan™-3 architectures are capable of supporting data bus widths of up to 36-bits. A self-addressing FIFO reference design uses these block memories to store both data and address information in a single memory location. This application note (covering all Virtex-II, Virtex-II Pro™, and Spartan-3 devices) describes FIFO designs where no external counters are required. Only flag and status information logic is used. The resulting FIFOs are not fast (around 150 MHz). Their advantage is in using only one clock load. In addition, the status mechanism is very simple. These FIFOs are more suitable for data throttling in continuous data systems rather than the full or empty detection required in frame-based data systems.

Introduction

In Figure 1, the self-addressing FIFO counters are implemented as a ROM based sequencer inside the same memory being used for data storage. As there are no counters, the clock load for the incoming and outgoing data is exactly one. Thus, clock skew is no longer an issue, and none of the available 16 global clock trees need to be used.
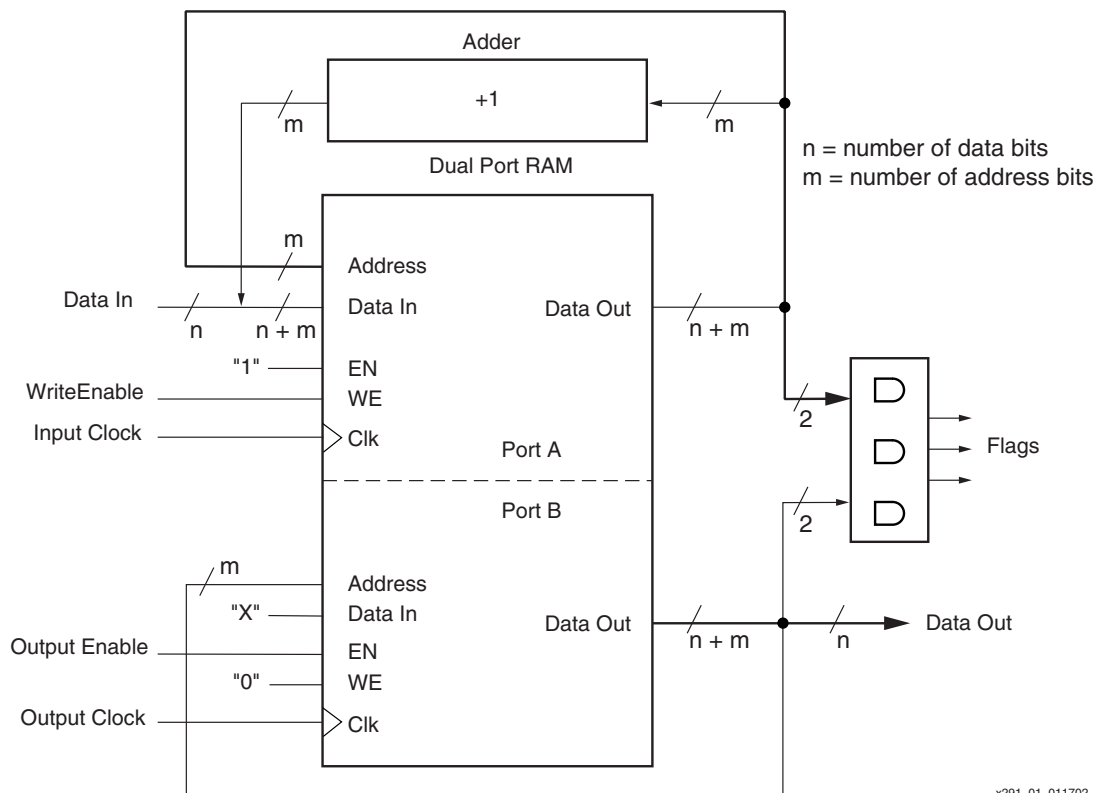

Figure 1: **Self-Addressing FIFO**

The true dual-port structure of the block RAMs includes input and output data busses plus address and control signals for each port. Data can be independently read from and written to each port. Both read and write accesses are synchronized to the appropriate port clock, with a positive or negative selectable edge. The only operational limitation is not to write data to a given address from one port while it is being read from the other port. To allow for parity, the basic array size of the block memory is 18 Kbits. Each port may be independently configured for a specific data bus width. For example, Port A can be configured as a 512 x 36 memory, and port B configured differently as a 2048 x 9 memory. Using these features, clock domains are safely crossed at the same time as the data width is converted.

As an example, the output buffer on a communication system could use 36-bit wide data written at 50 MHz, and read as 9-bit wide data at 155 MHz. All the various combinations of memory structure are shown in Table 1.

Table 1: **Possible Dual Port RAM Configurations**

Port A	Port B
512 x 36	512 x 36, 1024 x 18, 2048 x 9, 4096 x 4, 8192 x 2, 16384 x 1
1024 x 18	1024 x 18, 2048 x 9, 4096 x 4, 8192 x 2, 16384 x 1
2048 x 9	2048 x 9, 4096 x 4, 8192 x 2, 16384 x 1
4096 x 4	4096 x 4, 8192 x 2, 16384 x 1
8192 x 2	8192 x 2, 16384 x 1
16384 x 1	16384 x 1

Implementing a Counter in a Memory

There are two possible methods for implementing a counter inside a memory. Method one is presented for interest while method two forms the basis of the reference design.

By deasserting the write enable pin, a synchronous RAM becomes a synchronous ROM, or in other words, a sequencer. The contents of each of the ROMs can be initialized via the bitstream at device power-up. Essentially any sequence and therefore any state machine (within input and output pin limits) can be implemented in the ROM. The n-bit counter is a trivial case, it is a state machine with "n" inputs and "n" outputs. The count order is determined by the contents of the memory. The count speed is governed by the output enable control, effectively making it a clock enable. With counters implemented in one block memory, and data storage in another a FIFO can be built. However, this method gives a fanout of two on the read and write clocks. Although better than the classic FIFO design, there is still a possibility of clock skew.

The second method uses a feedback mechanism. M address bits are fed back from the RAM output data port to its address port. The same m bits are fed to a simple incrementer circuit. The output of the incrementer circuit is combined with the incoming data word. This next address plus data word is written synchronously into the next memory location. The only initialization needed is presetting the RAM outputs to x01 at reset. This function is available by using the block memory SSRA pin.

We can now imagine a scenario, where some of each word in the memory is used as the counter storage, and the rest is used for data storage. This is where the wide data busses comes in handy. For example, a 36-bit word can contain 32 bits of data and four bits of address, to give a 16-deep by 32-wide FIFO. Or the 36-bit word could have nine bits of address and 27 bits of data to give a 512-deep by 27-wide FIFO. The choice is up to the designer.

The top two bits of the incrementer circuit are Gray encoded. Only one of these two bits can ever change at any one clock transition. This minimizes any possibility of error when reading the flags, but it makes the incrementer slightly more complicated, and therefore slower. To maximize the speed of the design, a pure binary incrementer could be used, but the flags will need to be checked twice to ensure a valid flag value.

In Figure 1, an example data transfer is started by writing data (23) to location x00. After the clock edge, the data output of port A contains x0123. The lower data byte (23) is ignored, the upper byte (x01) is fed back as the address for the next write. A one is added to x01 for merging with the next incoming data. For example, if the next incoming data is 45, then x0245 is written into location x01. This value then appears at the output of port A. The next cycle will write to location x03 and so forth.

Reading data is less complicated. An output clock and optional output enable signal are applied to Port B. Data is read synchronous to the clock. The lower byte is the valid data that was written into Port A of the FIFO. The upper byte contains the address for the next read. The upper byte is therefore fed back to the address inputs of Port B.

Having covered the basic FIFO operation, one very important feature remains. Indicating whether a FIFO is full, empty, or somewhere in between is usually referred to as a flag. Since the top two bits of the counters are Gray coded, only one of the two bits changes at a time. It is safe to compare the top two bits of the write counter with the top two bits of the read counter, even though they are in two separate time domains. This comparison gives three flag outputs.

Flag0: The FIFO being between empty and one-half full

Flag1: The FIFO being between one-quarter to three-quarters full

Flag2: The FIFO being between one-half to full

This method gives a very simple yet elegant mechanism for handling FIFO requests. To take advantage of this method use flag1 (retimed to the receiver clock domain) as the FIFO read enable. In this way the FIFO is never near overflow, or emptied, and all asynchronous conditions are avoided. A standard system is also assumed to have continuous input data. A pictorial view of the counter and flag operation is given in Figure 2.

x291_02_062201

Figure 2: **Flag Operation**

Three different self-addressing FIFO reference designs are available from the Xilinx Web site at **ftp://ftp.xilinx.com/pub/applications/xapp/xapp291.zip.** These are written in VHDL and Verilog and are fully synthesizable. The designs describe FIFOs with equal input and output data widths, as well as versions where the input is twice as wide as the output, or the output is twice as wide as the input. Details of the available data widths and depths are given in the code.

Conclusion

The self-addressing FIFO is a small and novel mechanism for transferring data between clock domains while avoiding the necessity of using a clock tree. The only constraints are the obvious ones. Data must be valid at the clock edges and the clock period still needs to be controlled. Clock skew is not an issue and therefore general purpose routing may be used for the input or output clock depending on the system architecture.

When implemented into the Virtex-II or Virtex-II Pro devices, the circuit is capable of running at around 140 MHz for input data, and 200 MHz for the output data.

This shows the ability to build multiple self-addressing FIFOs, allowing designers to input or output data to external devices without ever using the on-board global clock resources. This is extremely useful in the design of many sorts of systems, for example where interfaces to multiple external PHY devices are required.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/02/01	1.0	Initial Xilinx release.
02/27/02	1.1	Changed Summary for clarity, revised Figure 1, and added Verilog support.
05/27/03	1.2	Added Spartan-3 family to "Summary." Made minor changes to the "Introduction" and "Conclusion."