Bit-Width Conversion Based on Asynchronous FIFO

Xianshun Song*

College of Integrated Circuit Science and Engineering, Nanjing University of Posts and Telecommunications, Jiangsu, China

* Corresponding Author Email: b19021129@njupt.edu.cn

Abstract. As digital integrated circuits become larger and larger, data transfer between different clocks and different bit widths becomes inevitable. For different clocks, asynchronous First In First Out (FIFO) can be used to solve very well. In this paper, we introduce the basic theory of asynchronous FIFO, and then implement the transfer between different bit widths based on asynchronous FIFO. By analyzing the bit-width conversion in two cases, input bit-width < output bit-width and input bit-width > output bit-width, we finally get a unified conversion idea: input data is assembled and input into asynchronous FIFO, and asynchronous FIF3O output data is split and output, and a bit-width conversion method is given by borrowing the storage function of asynchronous FIFO. Finally, the simulation is verified with the help of Verilog. By this method, an asynchronous FIFO can be implemented for conversion between various bit widths without considering the difference between input bit widths larger or smaller than output bit widths, and the conversion can be done in one step, thus improving the conversion efficiency.

Keywords: Asynchronous; FIFO; Bit-Width Conversion.

1. Introduction

With the rapid development of information technology, especially after the 1990s, the United States successfully used electronic warfare and information warfare in the South Slavic War and the two Gulf wars. Currently, the fastest way to transfer data from military computers, DMA, has a maximum transfer rate of less than 5 Mbps. With ultra-high sampling rates of tens of Mbps, the next collection process is often over before the previous data is read by the computer. Therefore, the use of traditional computer transfer processing methods will obviously lead to data loss and confusion. Therefore, for fast collection and slow processing systems, cache must be used. The FIFO is a good cache component [1].

FIFO used in the part of the data interface that needs to produce time, used to store, buffer the data transfer between two asynchronous clocks. With the development of integrated circuit technology, the design scale of the circuit is getting larger and larger, the system used more and more clocks, including many asynchronous clocks, and thus the requirement for asynchronous data transfer across different clock regions. Asynchronous clock processing has been a difficult circuit design, asynchronous clock signal processing methods, such as double latching, knotted rope method, etc., but these methods are not efficient, the best asynchronous clock processing method is to use asynchronous FIFO (first-in-first-out queue). At the same time, there are different bit-width data transmission problems between systems, and some of the articles proposed by the flag bit-width conversion method is more cumbersome [2]. In this paper, we will introduce a method of bit-width conversion using asynchronous FIFO with the unique storage function of FIFO. This will eliminate the generation of flag bits, facilitate the research of bit-width conversion, and enhance the development of digital circuits.

2. Asynchronous FIFO Theoretical Basis

2.1. Basic Introduction

It is used for sending and receiving data between asynchronous systems, and can well solve the problem of asynchronization of both.

As shown in Figure 1, it consists of a RAM as memory, a read address/pointer control module, a write address/pointer control module, 2 write pointer/read pointer synchronization modules, a full signal generation module, and a null signal generation module [3].

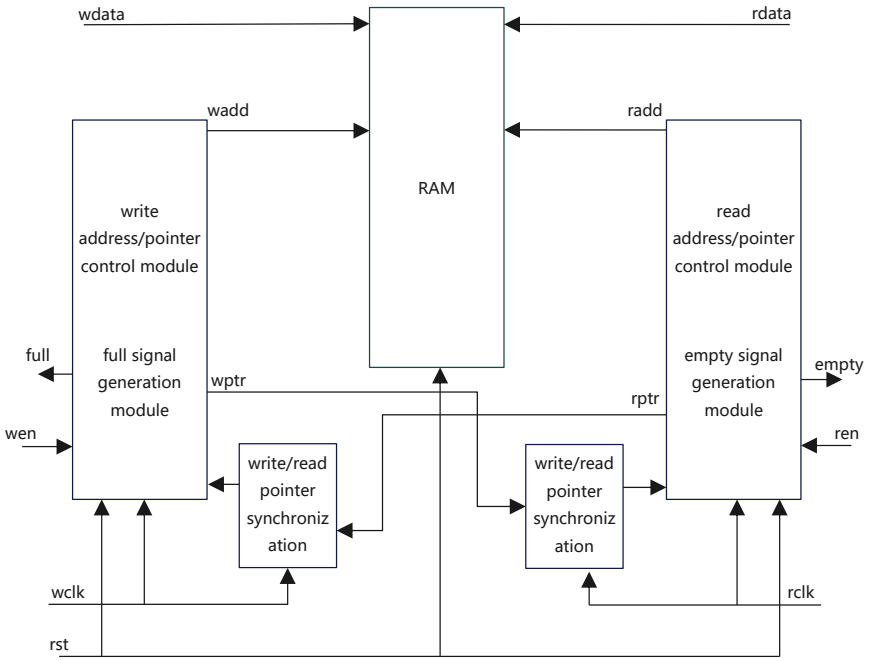

Fig. 1 Schematic diagram of asynchronous FIFO structure

The asynchronous FIFO is a first-in-first-out memory cell with read and write clocks controlled by two clocks.

The external ports are: wclk (write clock), wen (write enable), full (full signal), wdata (data input), rclk (read clock), ren (read enable), empty (empty signal), rdata (data output), rst (reset).

The purpose of each port: wclk is used to control the data writing rate; should be set to 1 when there is data writing; full becomes 1 when the memory is full, when full becomes 1, even if wen=1, the data writing stops; wdata is used for external data writing; rclk is used to control the data reading rate; ren becomes 1 when there is data to be read. empty in memory data is empty (e.g., reset, or all data are read out) [4].

2.2. Working Principle

After reset, the write address and read address become 0, and all memory data is cleared to 0. After that, the reset side is invalidated, and then data input starts, while write enable is active and read enable is also active. In the write clock field, since there is no memory data at this time, a data will be written when the write clock edge comes. In the read clock domain, since the memory is empty at this time, no data is read at this time. Assuming that the write clock is faster than the read clock, then let the FIFO keep running, there will eventually be a write full state, when the full signal should be valid output and the write address and read address should be the same at this time, before the next data is read, if the next write clock comes, then because the data - write full, so will not continue to write data, otherwise there will be data loss. On this basis, the write clock is replaced to make it slower than the read clock, then in the subsequent operation, the FIFO memory data will be read empty, when the last data is read, the read address and write address continue to be the same, before the next data is written in, there will be no more data read, and the empty signal should be output valid at this time. Therefore, the asynchronous FIFO follows the first-in-first-out work process, and when the data is written full will not write data, and when the data is read empty will not read data, so this way there is the problem of when to read empty and when to read full.

2.3. Empty/full status determination

The previous description shows that when the data is read empty, the write address is the same as the read address. When the data is full, the write address and the read address are also the same. And it is conceivable that when the write address and read address are the same, it is only possible to read the empty or full state, so based on this, we can consider solving the problem of determining the empty and full state according to the same read and write address.

The problem of whether to read empty or full when the write address and read address are the same can be solved by adding one more bit. Assuming that memory can only store 8 data, the address must be at least 3 bits, and on top of that, one more bit is added to make it 4 bits -, because the address of the call to memory cannot be called with this 4-bit address, so there is a need to keep the 3-bit address, so now there are two addresses, and then this paper calls the 4-bit address a pointer to make it easier to distinguish it from the 3-bit address. Distinguish. First reset, reset, read and write pointer consistent, then valid write enables, and keep the read enable invalid. After writing another data, the write pointer +1, and soon the memory will be written. When write is full, read and write address is the same, and read pointer is kept as 0000 because there is no data read out, but at this time write pointer is 1000 at this time because there are 4 bits. on this basis, let write enable is invalid, and read enable is valid, when read a data, read pointer +1. when read out so data, that is, read empty state, read pointer becomes 1000. from the above process can be seen, when the address bits of read and write pointer When the address bits of the read and write pointers are the same and the highest bit is opposite, the FIFO is full; and when the address bits of the read and write pointers are the same and the highest bit is the same, the FIFO is empty. From this, we can get the method of determining the empty and full states [5].

2.4. Use of Gray Code

After the previous introduction of how to perform the empty-full state determination, it should have been possible to do it with natural binary pointers alone. However, because of the sub-stability problem in the circuit, the actual comparison is done using a pointer in the form of gray code. The sub-stable problem is that the sampled signal does not stabilize to 0 or 1 when the sampling clock edge arrives, so that the picked signal is a value between 0 and 1. So here the advantage of switching to Gray's code comes into play. Since there is only a 1-bit difference between two adjacent numbers, only 1 bit of the pointer will change when the pointer changes, while for natural binary numbers, there will be a number of simultaneous changes, for example, 0111 plus 1 becomes 1000, so all the bits change, so the probability of sub-stability is greatly reduced compared to natural binary numbers [6]. In addition, because the signal between 0 and 1 picked up by the sub-stable state will eventually become 0 or 1, for the gray code is to complete the expected change or not, for example: 0011->0010, if the final stable to 0, then the expected jump is completed; if stable to 1, then no change will occur, then the pointer will not move, which does not cause much impact, at most, will only lead to the appearance of false empty or false full situation. Combining the above two points, we will eventually take the gray code for comparison.

The specific determination method: the highest two opposite, the rest of the empty is full; all bits are the same is empty.

Conversion method: The natural binary number can be converted to gray code by shifting the natural binary number logically to the right by one bit and then doing a bitwise iso-or with itself.

2.5. False empty/False full

In the asynchronous FIFO pointer transfer will be synchronized with the write pointer to the read clock domain and the read pointer to the write clock domain. The classical way of synchronization is to add a 2-stage D flip-flop, as shown in Figure 2 [7].

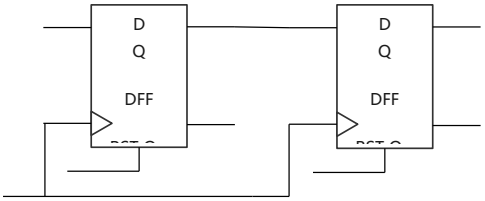

Fig. 2 write/read pointer synchronization module

This a two-level flip-flop will lead to the existence of false empty, false full phenomenon. In terms of the false empty phenomenon, as shown in Figure 1, you can see that the write pointer does not immediately pass to the read clock domain after the change, but also need to wait for the read clock to run twice to pass to the read clock domain, so the write pointer passed to the real write pointer at this time is not consistent (wptr is the write pointer in the write clock domain at this time, wptr2 is the write pointer synchronized at this time, you can see that the write pointer synchronized to the read clock domain is not the same as the The write pointer synchronized to the read clock domain is not the same as the actual write pointer), as shown in Figure 3 [8].

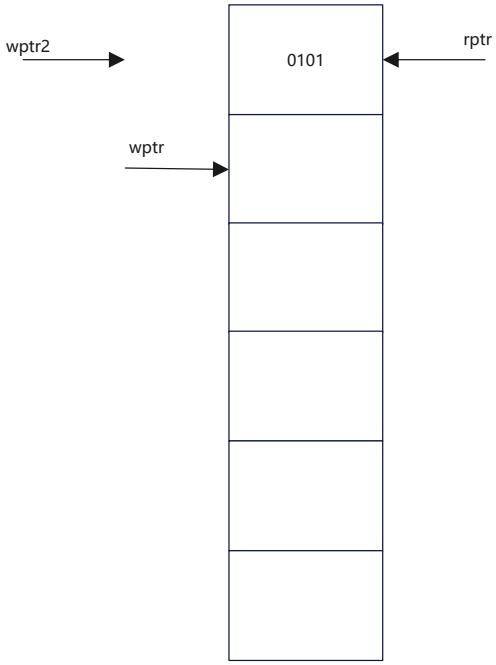

Fig. 3 Illustration of false empty

So, at this time, the FIFO judgment in the empty state, but in fact there has been data written.

For the false full phenomenon is similar, the read pointer passed to the write clock domain is actually not consistent with the read pointer at this moment, as shown in Figure 3, although at this time there is no write full, but the FIFO judgment in the write full state, so it will still output a full signal and terminate data writing. As shown in Figure 4 [9].

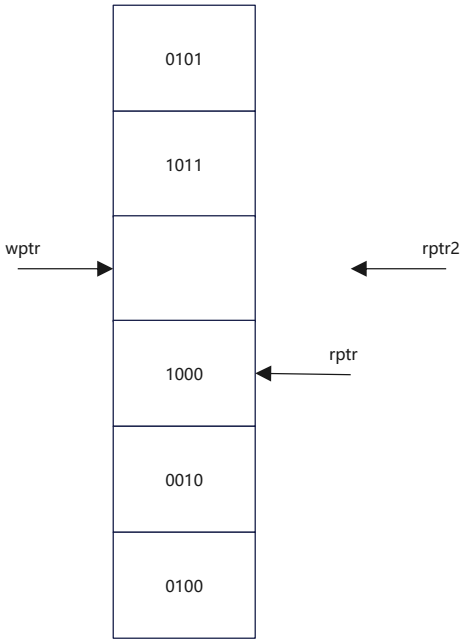

Fig. 4 Illustration of False Full

3. Asynchronous FIFO bit-width conversion

3.1. Input bit width m > output bit width n

Take a 64-bit wide FIFO as an example, there are two specific cases: input bit width > output bit width and input bit width < output bit width.

For input bit width > output bit width, the input bit width is 32 bits and the output bit width is 16 bits, for example. As the FIFO width is 64 bits, so we have to splice a number of input data before we can input into the FIFO (here is 2), after the splicing so that the FIFO wen is valid, the data will be collected, after this pointer synchronization, empty is invalid, if the receiver side is allowed to receive data, the FIFO ren is valid, the data output, after the output of a 64-bit data, as the receiver side After outputting a 64-bit data, since the receiver can only receive 16-bit data at a time, it is necessary to separate the output, here the data is divided into 4 parts and the ren is invalidated during the output to avoid data loss. After all the data is output, ren is valid again and the FIFO outputs the next data. This is shown in Figure 5.

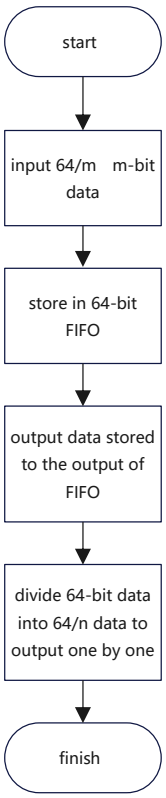

Fig. 5 m > n conversion diagram

3.2. Input bit width m < Output bit width n

For input bit width < output bit width, the input bit width is 16 bits and the output bit width is 32 bits as an example. Because the FIFO width is 64 bits, so we have to splice a number of input data before input into the FIFO (here 4), after the spelling so that the FIFO wen valid, the data will be collected, after this pointer synchronization, empty invalid, if the receiver allows the data, the FIFO ren valid, the data output, after the output of a 64-bit data, because the receiver After outputting a 64-bit data, since the receiver can only receive 32-bit data at a time, it is necessary to separate the output, here the data is divided into 2 parts, and ren is invalidated during the output to avoid data loss. After all the data is output, ren is valid again and the FIFO outputs the next data. This is shown in Figure 6.

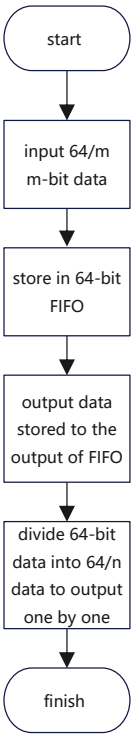

Fig. 6 m < n conversion diagram

3.3. Bit-width conversion Summary

It can be seen that whether m>n ignores m<n, the conversion idea is the same, first determine the input grouping, determine the number of groups, input the input data continuously until the spelling into 64 bits, in the spelling into 64 bits before the data will not be input into the FIFO, until the spelling into 64 bits into the FIFO. then determine the output grouping, the FIFO output 64 bits of data, a group of data for output, until 64 bits all output, and then the next from the FIFO exists in the FIFO. The FIFO outputs 64 bits of data and then outputs the data one group at a time until all 64 bits are output, and then outputs the next data in the FIFO from the FIFO that exists in the FIFO. So using the FIFO-style bit-width conversion proposed in this paper can no longer be as different as other articles to discuss m>n and m<n separately, to achieve a more unified, concise and convenient design.

4. Simulation results

The simulation is performed using Verilog for code simulation [10].

4.1. Bit-width conversion m>n and m<n

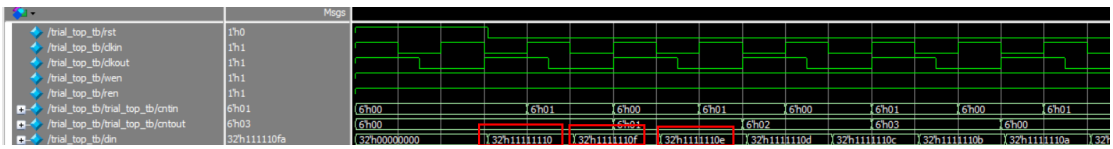

Fig. 7 m < n simulation input

From Figure 7, it can be seen that the inputs are 1111_1110, 1111_110f, 1111_110e.

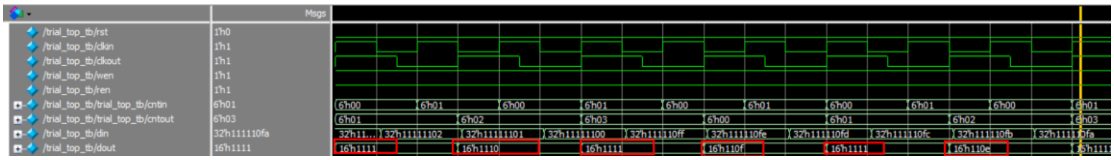

Fig.8 m < n simulation output

From Figure 8, the outputs are 1111, 1110, 1111, 110f, 1111, 110e, which shows the completion of the 32 to 16 bit-wise conversion.

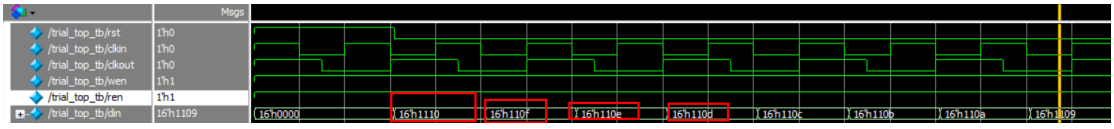

Fig.9 m > n simulation input

From Figure 9, the inputs are 1110, 110f, 110e, 110d.

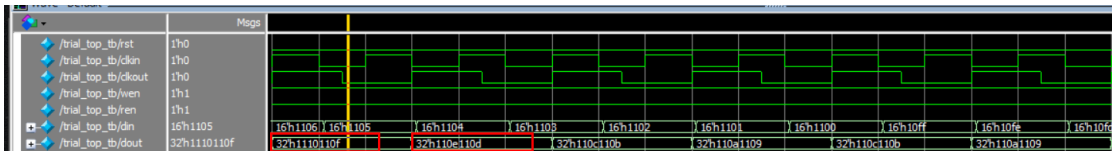

Fig.10 m > n simulation output

From Figure 10, the output is 1110110f, 110e110d, which shows the completion of the 16-to-32-bit conversion.

4.2. Bit-width conversion 16 to 64

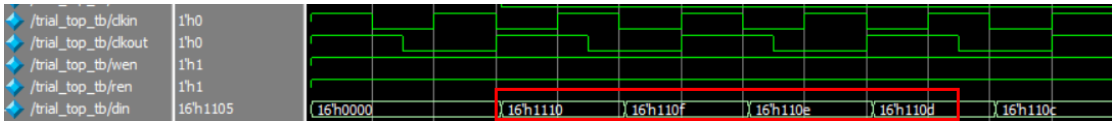

Fig.11 16 to 64 simulation input

As can be seen in Figure 11, the inputs are 1110, 110f, 110e, 110d.

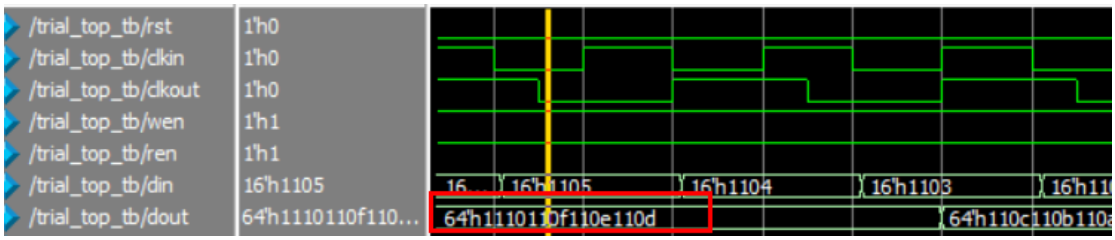

Fig.12 16 to 64 simulation output

As can be seen from Figure 12, the output is 1110_110f_110e_110d, which shows the completion of the 16 to 64-bit conversion

5. Summary

This paper gives a brief explanation of the background of the development of asynchronous FIFO, further introduces why asynchronous FIFO is used and some shortcomings of asynchronous FIFO, and addresses this shortcoming to get the core of this paper: to realize the transfer between data of different bit widths with the help of asynchronous FIFO. The basic theory of asynchronous FIFO is introduced in more detail, on top of which the method of using asynchronous FIFO to realize the conversion between data of different bit widths is explained - the input data is pieced together, and when output, the data is split and output, and then the data that cannot be output in time is cached by the storage function of FIFO itself. The data that cannot be output in time is cached by the FIFO storage function. Through the bit-width conversion method proposed in this paper, it is no longer

necessary to discuss the cases of m<n and m>n separately, and the process is also simpler; at the same time, the conversion efficiency is improved, and instead of step-by-step bit-width conversion (e.g., 16 to 64, to 16 to 32 and then 32 to 64) as is common in engineering, a one-step conversion can be achieved directly (you can directly convert from 16 to 64 bits). Finally, the simulation is verified with the help of Verilog HDL, and the simulation results show that the asynchronous FIFO conversion method proposed in this Paper is effective.

References

[1] Dereli S, Uç M. Exponential Computing Digital Circuit Design Developed for FPGA-based Embedded Systems. Academic Perspective Procedia. 2020;3(1):291-300.

[2] Xiao Y, Zhou R. Low latency high throughout circular asynchronous FIFO. Tsinghua Science and Technology. 2008;13(6):812-816.

[3] Liu FF. Research on power load forecasting based on Improved BP neural network. Harbin Institute of Technology, 2011.

[4] Yu J. Dual-Clock Asynchronous FIFO in System Verilog. Verilog Pro. Published December 7, 2015. https://www.verilogpro.com/asynchronous-fifo-design/

[5] Zipcpu. Crossing clock domains with an Asynchronous FIFO. zipcpu.com. Published 2018. Accessed December 7, 2022. http://zipcpu.com/blog/2018/07/06/afifo.html

[6] Liu BQ, Liu MZ, Yang G, Mao XB, Li HL. Research and Design of Asynchronous FIFO Based on FPGA. Applied Mechanics and Materials. 2014;644-650:3440-3444.

[7] Kim HK, Wang LT, Wu YL, Jone WB. Testing of Synchronizers in Asynchronous FIFO. Journal of Electronic Testing. 2013;29(1):49-72.

[8] Yadlapati A, Kishore Kakarla H. Design and Verification of Asynchronous FIFO with Novel Architecture Using Verilog HDL. Journal of Engineering and Applied Sciences. 2019;14(1):159-163.

[9] Park JM, Kim SM, Oh MH, Cho KR. Design of an Asynchronous Data Cache with FIFO Buffer for Write Back Mode. The Journal of the Korea Contents Association. 2010;10(6):72-79.

[10] Yadlapati A, Hari Kishore K. Low power synthesis for asynchronous FIFO using unified power format (UPF). International Journal of Engineering & Technology. 2018;7(2.8):7.