A Verilog-Based Configurable Multi-Width Asynchronous FIFO Design

Yuchen Han [1], Kexin Xiong [2,*]

[1] School of network and communication engineering, Jinling Institute of Technology, Nanjing, 211199, China

[2] College of Electronics and Information Engineering, Shanghai University of Electric Power, Shanghai, 201203, China

* Corresponding Author Email: kexin_xiong@outlook.com

Abstract. With the rapid development of integrated circuits, asynchronous FIFO, as a core component in digital system design, plays a significant role in data transmission and storage across clock domains. This paper designs a configurable bit-width asynchronous FIFO system on the Verilog platform that supports data widths of 8, 16, 32, and 64 bits. The system is partitioned into several key modules, including a write control module, a read control module, a storage module, a cross-clock pointer synchronization module, a bit width configuration module, and a top-level module. Functional verification and performance evaluation are carried out employing functional simulation, with comparative tests conducted under various write/read clock frequencies and bit-width configurations. The results show that under different write clock and read clock frequencies, this design can complete data transmission without loss or out-of-order, with the delay within a controllable range, and it can still maintain a high throughput efficiency even at a high write rate. This design can maintain the advantage of cross-clock domain transmission of asynchronous FIFO while flexibly adjusting the data width through hardware parameterization or runtime configuration, thereby adapting to more application requirements and improving system universality and resource utilization.

Keywords: Asynchronous FIFO, FIFO Design, Configurable Bit Width.

1. Introduction

In recent decades, innovative System-on-Chip (SoC) design has become a critical area of research, driven by emerging trends and complex application demands [1]. An SoC integrates a variety of IP cores, such as multi-core CPU, GPU, DSP, dedicated accelerators, various peripheral controllers, memory, and so on. The efficiency of the CPU is crucial in SoC designs as it directly impacts system performance [2]. The GPU, which was originally designed for graphical data processing, is increasingly being used in parallel computing in general engineering and science because of the efficiency arising by having thousands of computing cores [3]. A DSP works well in signal processing applications because it is optimized to process signals efficiently, is relatively not expensive, and has a well-defined development path and fixed hardware configuration [4]. Meanwhile, asynchronous FIFO is widely used in digital systems to temporarily store and transfer data between components that operate under different clocks [5]. Nowadays, there are many performance optimization schemes for Asynchronous FIFO. For example, designer use Gray code to solve the substability problem, increase the bit width and the number of bits of the empty-full warning flag [6]. The design quality of the asynchronous FIFO determines the data throughput, transmission delay, and other aspects of the SoC. The configurable bit-width design of asynchronous FIFO can maintain the advantage of cross-clock domain transmission of the asynchronous FIFO, while realizing flexible adjustment of data width through hardware parameterization or runtime configuration, adapting to more application requirements, improving system versatility, and optimizing resource utilization.

With the continuous expansion of integrated circuit design scale and the increasing complexity of systems, the design of asynchronous FIFO is facing multiple technical challenges, and scholars at home and abroad have conducted in-depth research on this. In the design of asynchronous FIFOs, the problem of metastability is particularly prominent. When data signals cross clock domains, the

asynchrony of clock phases and frequencies may cause flip-flops to enter an uncertain state, which in turn leads to system errors. Many domestic researchers have proposed various solutions to this problem. Synchronization of FIFO pointers into the opposite clock domain is safely accomplished using Gray code pointers [7]. The article studies and analyzes the influence of memory cells and sense amplifiers on access speed, which greatly shortens the access time of FIFO [8]. The FPGA-based asynchronous FIFO buffer data overflow control system performs excellently in both data monitoring and control performance. By establishing a storage overflow model and designing a data overflow control algorithm, the system can effectively address the issue of data overflow [9]. A delay-controllable asynchronous FIFO circuit that achieves high-precision delay control through integer delay and fractional delay. When multiple chips are working simultaneously, this circuit can eliminate the offset between data sources and achieve multi-chip synchronization [10]. Meanwhile, foreign researchers have brought new possibilities to the design of asynchronous FIFO in terms of new materials and new technologies. low-power design methodologies like DVFS for efficient energy use.Explores novel materials such as carbon nanotubes for improved data transfer and efficiency [11]. A high-speed and large-capacity asynchronous FIFO based on an FPGA and DDR2 SDRAM is also designed. FIFO access control of DDR2 memory based on FPGA is realized by WFIFO, RFIFO, and FIFO controller. This design supports parallel data reading and writing, has fixed access cycle, and can provide high access rate [12].

Although significant progress has been made in the research of asynchronous FIFO, there are still many challenges. This paper aims to design and implement an asynchronous FIFO structure with dynamic bit-width configuration to address the limitations of traditional asynchronous FIFOs, such as fixed data width and insufficient flexibility.

2. Methods

2.1. Design of Configurable Asynchronous FIFO System Architecture

The asynchronous FIFO designed in this paper adds a switchable bit-width module based on the traditional structure to adapt to the data throughput requirements in different application scenarios. The overall system architecture primarily consists of a write control module, a read control module, a data storage module, a pointer synchronization module, and a bit-width switching control module. Among them, the write control module is responsible for receiving data from the write clock domain and performing storage operations; the read control module completes data retrieval under the read clock domain; the storage unit uses a dual-port RAM to achieve parallel data access; the pointer synchronization module utilizes a two-stage register for synchronization; and the bit-width switching control module enables flexible transitions between different bit-widths through data splicing and segmentation technologies. The program running flowchart is shown in Figure 1.

This design implements an asynchronous FIFO with configurable data width (supporting 8/16/32/64 bits). Its core lies in using Gray code pointers and a synchronization mechanism to safely transfer states across clock domains, and adapting to different data widths through data segmentation and recombination. The following is its overall operation mode. The algorithm which is aynchronous FIFO of configurable width is shown in Algorithm 1.

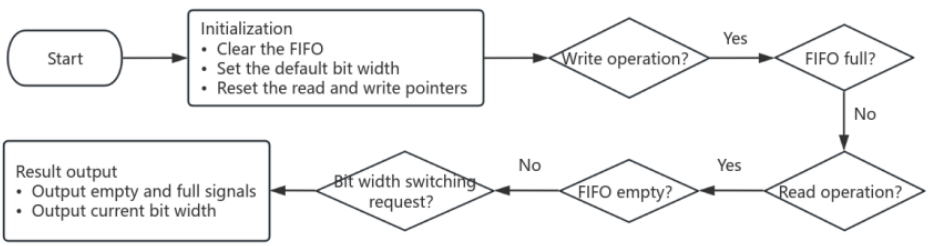

Fig 1. Program running flowchart

Algorithm 1: Asynchronous FIFO Operation with Configurable Width

Input: wr_clk, rd_clk, rst_n, wr_en, wr_data[W-1:0], rd_en, cfg_width_in[1:0], cfg_valid
Output: rd_data[W-1:0], rd_data_valid, wr_full, rd_empty, cfg_ready

```
// Initialization
upon rst_n asserted do
    wr_ptr_bin ← 0; wr_ptr_gray ← 0
    rd_ptr_bin ← 0; rd_ptr_gray ← 0
    wr_full ← 0; rd_empty ← 1; rd_data_valid ← 0
    current_width ← 2'b00 // Default to 32-bit
    cfg_ready ← 1

// Width Configuration Process
if cfg_valid and cfg_ready then
    current_width ← cfg_width_in
    cfg_ready ← 0
    //  Wait for safe context switch
    cfg_ready ← 1
end if
// Write Domain Operation
if wr_en and ¬wr_full then
    // Data decomposition & storage based on current_width
    case current_width
        2'b00: // 8-bit
            mem[wr_ptr_bin] ← wr_data[7:0]
        2'b01: // 16-bit
            mem[{wr_ptr_bin, 1'b0}] ← wr_data[7:0]
            mem[{wr_ptr_bin, 1'b1}] ← wr_data[15:8]
        //  Cases for 32-bit and 64-bit
    end case
    next_wr_bin ← wr_ptr_bin + 1
    wr_ptr_gray ← next_wr_bin ⊕ (next_wr_bin >> 1) // Binary to Gray
end if

// Read Domain Operation
if rd_en and ¬rd_empty then
    // Data read & reassembly based on current_width → rd_data
    case current_width
        2'b00: rd_data[7:0] ← mem[rd_ptr_bin]
        2'b01: rd_data[15:0] ← {mem[{rd_ptr_bin, 1'b1}], mem[{rd_ptr_bin, 1'b0}]}
        // Cases for 32-bit and 64-bit
    end case
    rd_data_valid ← 1
    next_rd_bin ← rd_ptr_bin + 1
    rd_ptr_gray ← next_rd_bin ⊕ (next_rd_bin >> 1)
else
    rd_data_valid ← 0
end if

// Pointer Synchronization
// Synchronize wr_ptr_gray to rd_clk domain:
on every rd_clk rising edge do
    wr_sync_reg1 ← wr_ptr_gray
    wr_sync_reg2 ← wr_sync_reg1
    synced_wr_ptr ← wr_sync_reg2

// Synchronize rd_ptr_gray to wr_clk domain:
on every wr_clk rising edge do
    rd_sync_reg1 ← rd_ptr_gray
    rd_sync_reg2 ← rd_sync_reg1
    synced_rd_ptr ← rd_sync_reg2

// Status Flag Generation
next_wr_gray ← (wr_ptr_bin + 1) ⊕ ((wr_ptr_bin + 1) >> 1)
wr_full ← (next_wr_gray == synced_rd_ptr)

next_rd_gray ← (rd_ptr_bin + 1) ⊕ ((rd_ptr_bin + 1) >> 1)
rd_empty ← (next_rd_gray == synced_wr_ptr)
```

2.2. Simulation and Verification Methods

The design was tested via functional simulation. A testbench written in Verilog is used for functional verification of the asynchronous FIFO. The top-level module async_fifo_top and the testbench fifo_tb are loaded to run the simulation. The test focuses on the following aspects.

(1) Write Operation: Verify whether wr_data is successfully written into the RAM when wr_en is valid;

(2) Read Operation: Verify whether rd_data outputs correct data when rd_en is valid;

(3) Empty and Full Status: Verify whether the wr_full and rd_empty flags are correctly set and reset when the FIFO reaches the full or empty state.

By changing the value of cfg_width_in, test the write and read functions of the FIFO under different bit-widths (8-bit, 16-bit, 32-bit, 64-bit), and observe whether there is data loss, duplication, or out-of-order phenomena during the write and read processes.

3. Test Results and Analysis

3.1. Function Simulation Results

Figure 2 presents the overall waveform diagram. In the simulation waveform diagram, the operation process of the system can be divided into the following four stages:

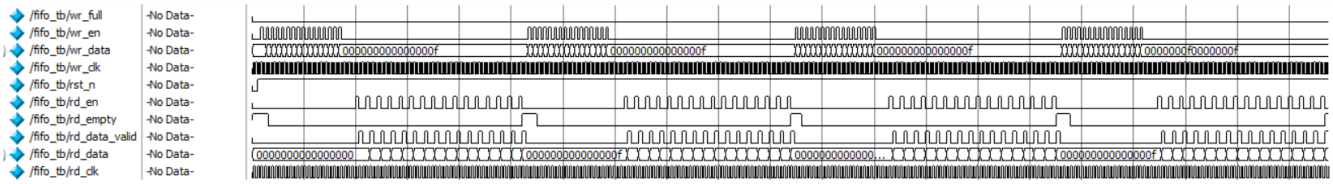

Fig. 2 Simulation waveform diagram

(1) Reset stage (0 ps–20000 ps)

Figure 3 presents the reset stage clearly in the simulation process. At the initial time, the reset signal rst_n is at a low level, and the read and write pointers inside the FIFO are reset to zero. The rd_empty signal is at a high level, and the wr_full signal is at a low level. When 20000 seconds pass, the rst_n signal will be released from reset, changing from a low level to a high level, and the system enters the normal work state.

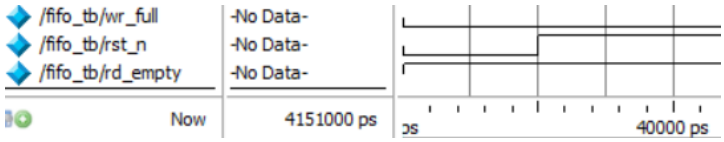

Fig. 3 Reset stage

(2) Writing stage (approximately 35000 ps-345000 ps)

Figure 4 presents the writing stage clearly in the simulation process. When the rst_n signal is released from reset, the write clock wr_clk will start to drive, and the write enable signal wr_en is first changed into high level at approximately 35000 ps, and at the next rising edge of wr_clk, which is at 45000 ps, wr_data begins to write the test data (0, …, 9, a, …, f) sequentially. The rd_empty signal drops to a low level at approximately 63000 ps, indicating that there is already data inside the FIFO.

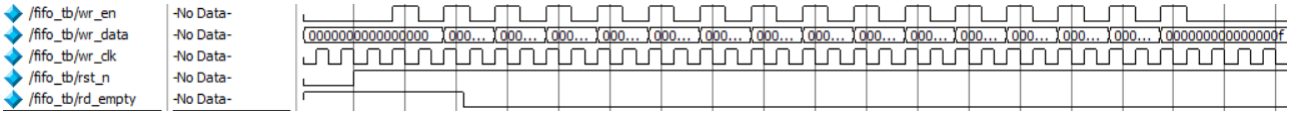

Fig. 4 Writing stage

(3) Reading stage (approximately 400000 ps-1057000 ps)

Figure 5 presents the reading stage clearly in the simulation process. After the writing operation is completed, the read clock rd_clk will start to drive, and the read enable signal rd_en is pulled high at approximately 400000 ps, rd_data_valid is pulled high at 413000 ps, and at the next rising edge of rd_clk, which is at 427000 ps, rd_data begins to read the test data sequentially, and the sequence of output data are as same as the write sequence (0, …, 9, a, …, f). After the last data is read out, the rd_empty signal is pulled high again at 1057000 ps, indicating that the FIFO is empty.

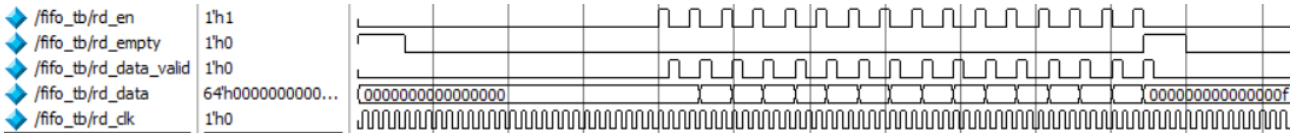

Fig. 5 Reading stage

(4) Switching between empty and full states

Figure 6 presents the switching between empty and full states clearly in the simulation process. When the FIFO is full, wr_full is at a high level. The waveform shows that it remains at a low level throughout the simulation, indicating that the amount of writing data has not reached the capacity limit of the FIFO. When the FIFO is empty, rd_empty is at a high level. The waveform shows that it becomes low after the data are written, and it returns to a high level after all the data have been read out.

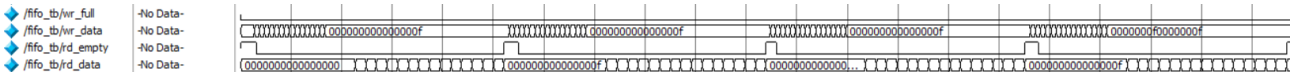

Fig 6. Switching between empty and full

Combining the logs and waveform diagrams, we can get the following conclusions:

(1) Bit widths are configurable: We test the write and read functions of the FIFO under different bit widths (8-bit, 16-bit, 32-bit,64-bit) by switching the values of DATA_WIDTH in the fifo_tb. All four tests write and read data successfully, and the read data are completely consistent with the written data, without situations that result in data loss or disorder. This proves that the configurable bit widths feature of the design is correct and valid, which verifies the function of configurable bit widths in the ram_memory module.

(2) Empty and full states: The wr_full and rd_empty signals can correctly indicate the empty and full states of the FIFO, ensuring the effectiveness of the read and write operations and preventing data overflowing or reading invalid data.

3.2. Performance Test and Analysis

When the rst_n signal is released from reset, the write clock wr_clk will start to drive. After the write enable signal wr_en is pulled high, the wr_data will begin to write the test data at the next rising edge of the write clock wr_clk. This indicates that the time of write delay from the wr_en signal being pulled high to when the data is actually written into memory is about half a wr_clk cycle, which is 5 ns. The write path is very short and there is almost no additional delay, so the performance of FIFO is excellent. After the write operation is completed, the read clock rd_clk will start to drive. After the read enable signal rd_en is pulled high and rd_data_valid is then pulled high, the rd_data will begin to read out the test data at the next rising edge of the read clock rd_clk. This indicates that the time of read delay from the rd_en signal being pulled high to when the data is actually read out is about two wr_clk cycles, which is 28 ns. The read path is slightly longer than the write path, but still very short, and the delay is acceptable. Overall, the synchronization logic inside the FIFO is very efficient. Both the write and read delays are controlled within a few clock cycles. It is very important for a high-frequency or real-time system to have low read and write delays.

The throughput rate calculated by dividing the total data volume by the transmission time is shown in the following Table 1. Here, the total data volume is determined by the product of the number of data items, and the bit width and the transmission time is obtained as the difference between the end time and the start time. As the bit width of data increases, the throughput rate (MB/s) significantly improves. When the bit width is 8 bits, the throughput rate is approximately 25.40 MB/s. When the bit width is 16 bits, the throughput rate increases to 50.79 MB/s, which is almost twice that of 8 bits. When the bit width is 32 bits, the throughput rate further increases to 101.59 MB/s, which is almost four times that of 8 bits. When the bit width is 64 bits, the throughput rate reaches a peak at 203.17 MB/s, which is almost eight times that of 8 bits. This indicates that the FIFO can make full use of the bit width of data for each transmission, and the larger the bit width, the higher the throughput rate. The write clock is fixed at 100 MHz, and the read clock is 71.4 MHz. So the actual speed of data

output is controlled by the read clock. Although the frequency of the read clock limits the amount of data transmission per second, through flexible bit width configuration, the total amount of data transmitted by the FIFO in unit time increases significantly. This proves that this design has high practical value in data-intensive applications.

The results verified the advantages of configurable bit widths in the ram_memory module. It enables this FIFO design to choose the appropriate bit width according to different application scenarios, balancing throughput rate and resource occupation without designing new hardware, which can reduce costs. We can use a smaller bit width in scenarios of small data volume or low speed to save FPGA resources and use a wider bit width in high throughput rate scenarios to make full use of the bus and clock frequency.

Table 1. Four Schemes comparing

Test	Bit width (Bit)	Write clock (MHz)	Read clock (MHz)	Throughput rate (MB/s)
1	8	100	71.4	25.4
2	16	100	71.4	50.8
3	32	100	71.4	101.6
4	64	100	71.4	203.2

4. Conclusion

This paper designs and realizes an asynchronous FIFO of configurable bit widths, supporting a switch of 8, 16, 32, and 64 bits, and possessing data transmission capability of clock domain crossing synchronization and a high utilization rate of hardware resources. This design use the minimum granularity storage architecture, dynamic pointer stepping, and Gray code real-time encoding technology to realize data transmission of low delay and high throughput. Simulation results show that the FIFO can ensure data integrity and time stability under different bit widths, and the throughput rate increases linearly with the bit width. The read and write delay is low, and the empty and full signals reflect the storage status accurately, verifying the reliability of the design. Compared with other schemes of fixed bit width or methods based on software, this design realizes higher flexibility at the hardware level.

In the future, we can further optimize clock domain crossing synchronization and handshake mechanisms to improve delay performance, introduce a configurable depth and multi-channel system to enhance the throughput capacity of the system, and conduct actual tests combined with an FPGA to verify power consumption and resource occupation. At the same time, we can strengthen metastable state protection and synchronization optimization to further improve the reliability and universality of the design in complex SoC.

References

[1] Savithri G, Saritha T, Bhargavi B M, et al. Synergized Mixed-Signal System-on-Chip (SoC) Design and Development Using System-Level Modeling and Simulation[J]. SAE International Journal of Advances and Current Practices in Mobility,2024,7(2):974-985.

[2] Wang H, Zou Y, Wen G, et al. Memory Access Acceleration Through Architecture Design for Edge SoCs[C]//2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP). IEEE, 2024: 30-31.

[3] Park S H, Jo Y B, Ahn Y, et al. Development of multi-GPU–based smoothed particle hydrodynamics code for nuclear thermal hydraulics and safety: potential and challenges[J]. Frontiers in Energy Research, 2020, 8: 86.

[4] Diouri O, Gaga A, Ouanan H, et al. Comparison study of hardware architectures performance between FPGA and DSP processors for implementing digital signal processing algorithms: Application of FIR digital filter[J]. Results in Engineering, 2022, 16: 100639.

[5] Ye R. Application of Asynchronous FIFO in High-Speed Interface Data Transmission[C]//2025 2nd International Conference on Mechanics, Electronics Engineering and Automation (ICMEEA 2025). Atlantis Press, 2025: 939-947.

[6] Wang Q. A Study of Advances in Asynchronous FIFO Design[J]. Applied and Computational Engineering, 2025, 121: 14-23.

[7] Lincy D F, Thenappan S. Asynchronous FIFO design using Verilog[J]. Int. Res. J. Eng. Technol, 2020, 7(9): 2147-2151.

[8] Zhan Yongzheng, Li Tuo, Hu Qingsheng, et al. A high-speed asynchronous FIFO for 100 Gbit/s Ethernet PCS[J]. Microelectronics, 2022, 52(5): 886-892.

[9] Zhang Wei. An overflow control system for asynchronous FIFO cache data based on FPGA[J]. Ordnance Industry Automation, 2024, 9(43): 09.

[10] Chen Tingting, Lu Feng, Wan Shuqin, Shao Jie. A delay-controllable asynchronous FIFO circuit design[J]. Microelectronics, 2022, 52(1): 42.

[11] Chandu H S. Advancements in asynchronous FIFO design: A review of recent innovations and trends[J]. International Journal of Research and Analytical Reviews, 2023, 10(2): 573-580.

[12] Mingji Wang. Design and implementation of asynchronous FIFO[C]. 2nd International Conference on Functional Materials and Civil Engineering, 2024.