Scientific Research Publishing

Engineering, 2024, 16, 61-82
https://www.scirp.org/journal/eng
ISSN Online: 1947-394X
ISSN Print: 1947-3931

Review on the Usage of Synchronous and Asynchronous FIFOs in Digital Systems Design

Dongwei Hu[1], Yuejun Lei[2], Linan Wang[1*]

[1]The 54th Research Institute of CETC, Shijiazhuang, China
[2]MUC School of Information Engineering, Beijing, China
Email: hudw1980@sina.com, lyj@muc.edu.cn, *wln99@sina.com

How to cite this paper: Hu, D.W., Lei, Y.J. and Wang, L.N. (2024) Review on the Usage of Synchronous and Asynchronous FIFOs in Digital Systems Design. *Engineering*, **16**, 61-82.
https://doi.org/10.4236/eng.2024.163007

Received: December 28, 2023
Accepted: March 26, 2024
Published: March 29, 2024

Copyright © 2024 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

First-Input-First-Output (FIFO) buffers are extensively used in contemporary digital processors and System-on-Chips (SoC). There are synchronous FIFOs and asycnrhonous FIFOs. And different sized FIFOs should be implemented in different ways. FIFOs are used not only for the pipeline design within a processor, for the inter-processor communication networks, for example Network-on-Chips (NoCs), but also for the peripherals and the clock domain crossing at the whole SoC level. In this paper, we review the interface, the circuit implementation, and the various usages of FIFOs in various levels of the digital design. We can find that the usage of FIFOs could greatly facilitate the signal storage, signal decoupling, signal transfer, power domain separation and power domain crossing in digital systems. We hope that more attentions are paid to the usages of synchronous and asynchronous FIFOs and more sophististicated usages are discovered by the digital design communities.

Keywords

First-Input-First-Output, System-on-Chip, Network-on-Chip, Advanced eXtensible Interface, Asynchronous

1. Introduction

First-Input-First-Output (FIFO) buffer is a traditional module in digital systems [1]. It is extensively used in contemporary digital processors and SoCs. In digital processor design, the pipeline stages are separated by one-slot FIFOs [2]. In multi-core or many-core systems, the synchronous or asynchronous FIFOs are used as mailing-box [3] for inter-processor communications, or router of Network-on-Chips (NoCs) [4] [5]. At the SoC level, synchronous or asynchronous

*Corresponding author.

are used to buffer the transmitted or received data. And asynchronous FIFOs are used for clock-domain crossing bridges.

However, though extensively used, because of its simplicity, there is never a paper or a textbook summarized the usage of synchronous or asynchronous FIFOs thoroughly, which leads to the neglect of its importance in digital design communities.

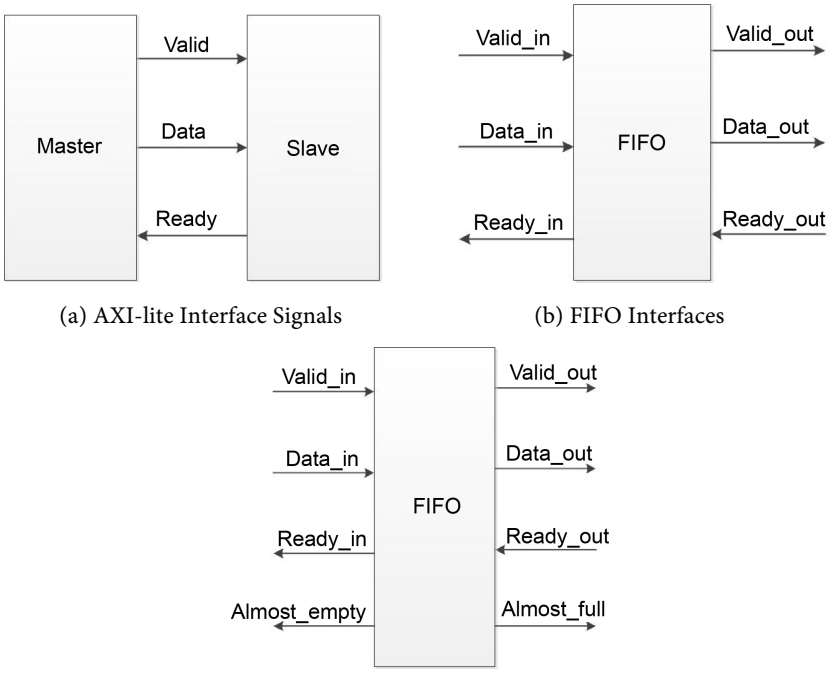
In recent years, the Advanced eXtensible Interface (AXI) bus is extensively used in digital systems [6]. The interface signals of AXI bus could perfectly match the signals of a FIFO. This leads to the resurgence of FIFO usages. This paper summarizes the usage of FIFOs at all level of digital systems, aiming to attract more attentions from the communities, and help people better understand their usages and importance.

2. Synchronous and Asynchronous FIFO

2.1. The Interface of Synchronous and Asynchronous FIFO

In this paper, all the FIFOs are with the AXI-lite interface [6] [7]. The signals of AXI-lite interface are shown in **Figure 1(a)**. There are "Data" signals from the master to the slave; a "Valid" signal indicating the "Data" are present, also from the master to the slave; and a "Ready" signal indicating that the slave is ready to accept the "Data" from the master. When both the "Valid and "Ready" are high, the transfer, the "Data" go from the master to the slave, is accomplished.

FIFO is a kind of digital module with both the master and slave interfaces. One side of the FIFO is presented as the slave, receiving "Data" in, and the other side as the master, sending "Data" out. **Figure 1(b)** shows the block diagram



(a) AXI-lite Interface Signals

(b) FIFO Interfaces

(c) FIFO Interfaces with Ints

Figure 1. AXI and FIFO Interfaces.

and interface signals of FIFO.

FIFOs only cache the "Data" temporally. They don't modify the content of the "Data". In addition, they keep the order of the input and output "Data" streams. That is, the data-slots coming into the FIFO earlier will go out off the FIFO earlier—this is why we name it FIFO: First-Input-First-Output.

When big-sized data are transferred through the FIFOs, interrupt signals could be used to facilitate the transfer. There are usually 2 interrupts, "Almost_empty" and "Almost_full", as shown in **Figure 1(c)**. When the FIFOs are almost empty, the signal "Almost_empty" interrupts the sending-side processor, and this processor puts more data into the FIFOs. When the FIFOs are almost full, the signal "Almost_full" interrupts the receiving-side processor, and that processor reads data out. When the FIFOs are big enough, these processors could be adequately alleviated from frequent interrupts.

2.2. The Implementation of Synchronous and Asynchronous FIFOs

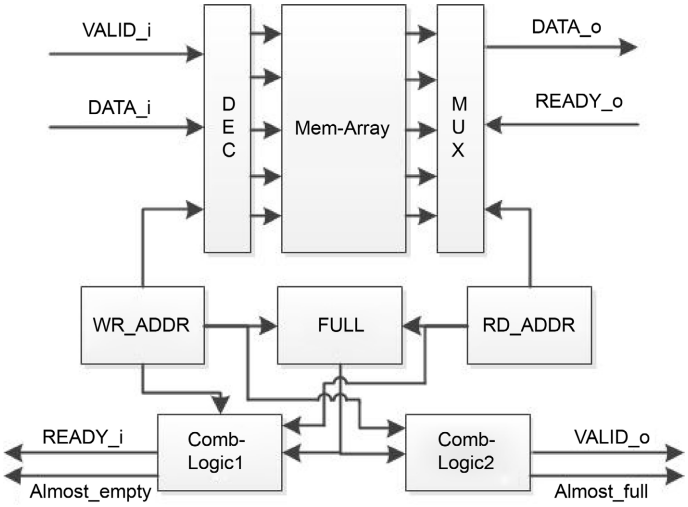
Figure 2(a) shows the schematic of synchronous FIFO. In **Figure 2(a)**, there are 2 address registers, one at the input side and the other at the output side. When the two address pointers are equal, the FIFO is either full or empty. A 'FULL' register, which should be cleared at reset (indicating empty but not full), could be employed to differentiate and track the full and empty state.

Figure 2(b) shows the schematic of asynchronous FIFO. The difficulty of asynchronous FIFO is that the address registers (read address, RA_ADDR, write address, WR_ADDR) are at different clock domains, so they need to be gray encoded (RA_GRAY and WA_GRAY) and delayed by two clock cycles (RA_d1, RA_d2 and WA_d1, WA_d2) at the other side (the other clock domain) before being used [8]. In addition, a "FULL/EMPTY" register is used in each clock domain.

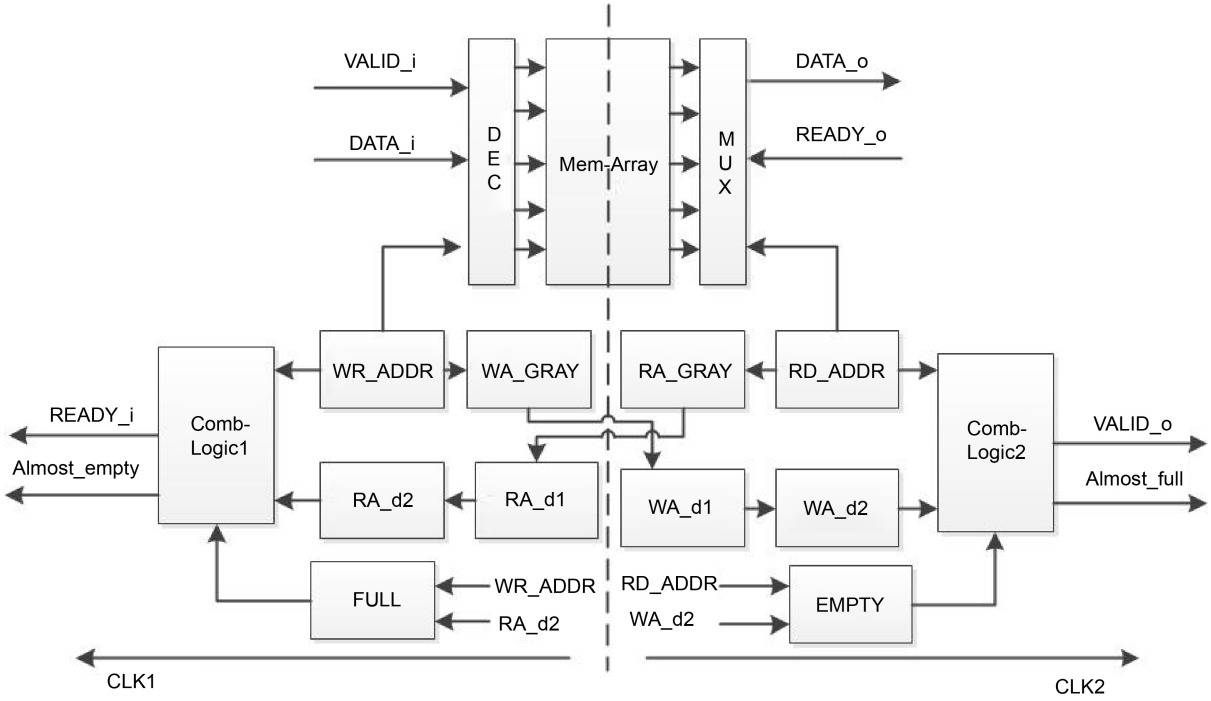
The main body of synchronous and asynchronous FIFOs is the memory array, which is responsible for storing information. The memory array could be implemented in a lot of ways. When it is small, it could be composed with D-flipflops. When it is medium-sized, it could be realized with latches [4]. When the size is large, it needs to be implemented with customized circuits.

However, in many circumstances, customized FIFOs are not provided but Dual-Port Static Random-Access-Memories (DPRAM) are available. In these cases, we need to construct FIFOs with DPRAMs. The main difference of FIFOs and DPRAMs is that they are with different read timing. For DPRAMs, the output data is one-clock delayed after the read command, while for FIFOs with AXI-lite interfaces, the data are output at the same time with the "Ready" signal.

A pipelined architecture [9] [10] could be used to transform the read timing of DPRAMs to that of FIFOs. **Figure 3** shows this architecture. In **Figure 3**, a small-sized synchronous FIFO (SyncFIFO) follows the DPRAM. The two clocks of DPRAM could be synchronous or asynchronous. The "Empty" and "Full" flags of DPRAM are generated in the same way as D-flipflop-based FIFOs, as



(a) Synchronous FIFO



(b) Asynchronous FIFO

Figure 2. Circuit implementation of synchronous and asynchronous FIFOs.

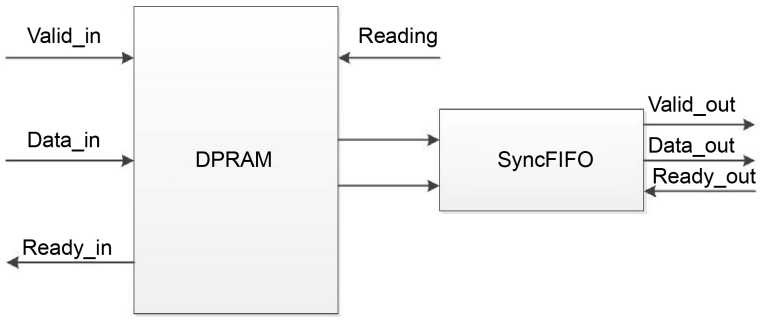


Figure 3. Constructing FIFOs with DPRAMs.

shown in Figure 2. As long as DPRAM is not empty and the following SyncFIFO is ready, the data in DPRAM are read out and put into the SyncFIFO. The output of the SyncFIFO, whose timing aligns with that of AXI-lite interface, serves as the top-level output of the pipelined architecture. In order to maximize the throughput and minimize the latency of the reading, the SyncFIFO should be implemented with a skip buffer (see following sections of this paper).

Due to the gray encoding and/or pipelined implementation, asynchronous FIFOs are with larger latencies then synchronous FIFOs. For synchronous FIFO, the minimum input-output latency is one-clock-tick. If we connect the two sides of asynchronous FIFO with the clocks with the same clock frequency, the latency of D-flipflop-based asynchronous FIFO would be 3 clock-ticks, while DPRAM-based asynchronous FIFO is 4 clock-ticks. Table 1 summarizes the latencies of synchronous and asynchronous FIFOs with different implementation styles.

2.3. Four Special Cases of Synchronous FIFO

2.3.1. One-Slot FIFO

If there is only one-slot data storage in the FIFO, we call the FIFO as One-slot FIFO. The one-slot data storage is usually realized with a register stage. The input "Data" always go into the register stage, and then go out. Therefore, the Valid_out, Data_out and Valid_in, Data_in are cut by the register stage.

However, in order to make the one-slot FIFO's throughput as high as possible, when the "Data" in the register stage are going out, new "Data" could be loaded in at the same time (the same clock cycle). This is to say, "Ready_out" would enable "Ready_in". Exactly we have

$$\text{Ready\_in} = \text{Ready\_out or (not Valid\_out)}.$$

"(not Valid_out)" indicates that the register stage is empty. Unlike the "Valid" and "Data" signals, "Ready_in" is coupled with "Ready_out", which leads to long combinational logic.

The schematic of one-slot FIFO is shown in Figure 4.

2.3.2. Skip Buffer

Skip buffer is one way to cut the "Ready" signal of One-slot FIFO. Figure 5 shows the architecture of skip buffer. It is composed by a mux and an one-slot FIFO. When the one-slot FIFO is empty and "Ready_out" is asserted, the "Data" go directly from input to output, skipping the FIFO. When the one-slot FIFO is

Table 1. Latencies of different implementations of synchronous and asynchronous FIFOs.

Implementation Styles	# of clock-ticks
Flip-flop-based Sync-FIFO	1
Flip-flop-based Async-FIFO	3
DPRAM-based Sync-FIFO	2
DPRAM-based Async-FIFO	4

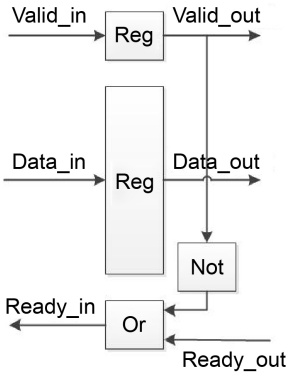


Figure 4. One-slot FIFO.

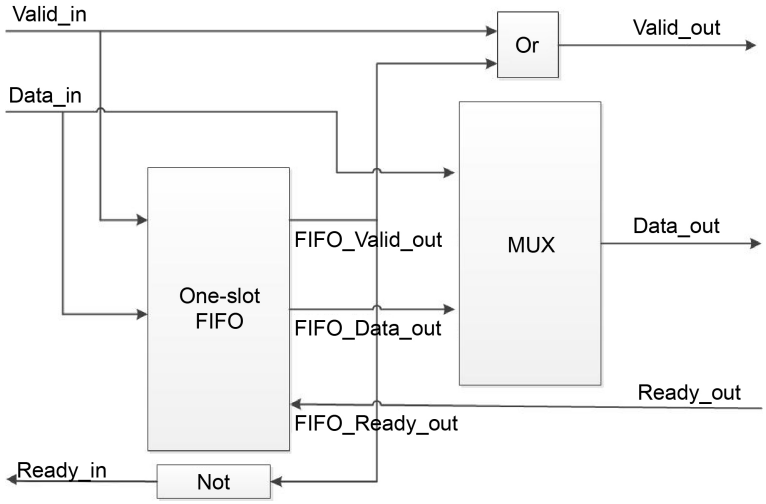


Figure 5. Skip Buffer.

empty, but "Ready_out" is not ready (desserted), one-slot "Data" could still be accepted and stored into the FIFO. When the FIFO is full, input "Data" could not be accepted until the "Data" in FIFO is gone first, thus keeping the order of the "Data" stream. Obviously, "Ready_in" depends on either the FIFO is full or not, that is,

$$\text{Ready\_in} = \text{not FIFO\_Valid\_out}$$

It's decoupled from "Ready_out". However, as there is a direct pathway from the input to the output, skip buffer couldn"t decouple "Valid_out", "Data_out" from "Valid_in", "Data_in".

2.3.3. Decoupling Skip Buffer

One-slot FIFO decouples "Data" and "Valid" signals, but fails in decoupling "Ready" signal. Skip buffer decouples "Ready" but couldn't decouple "Data" and "Valid" signals. If we combine them together, all signals could be decoupled. We name this kind of circuit as Decoupling Skip Buffer.

There are two types of decoupling skip buffer as shown in **Figure 6**. From the application point of view, there are no differences for the two. Therefore, when we refer to decoupling skip buffer, either type could be applied.

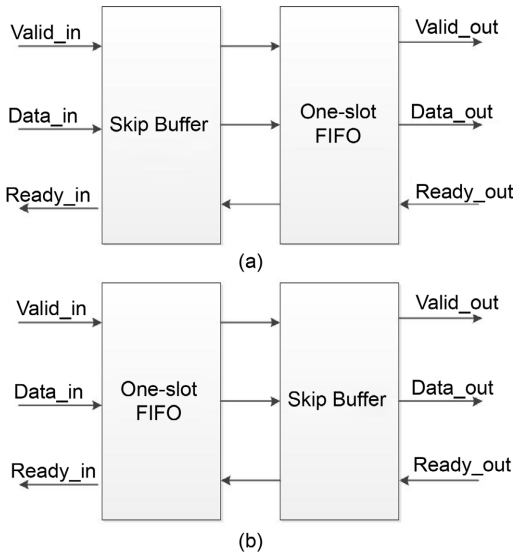


Figure 6. Decoupling Skip Buffer.

2.3.4. Two-Slot FIFO

There are two One-slot FIFOs in decoupling skip buffer, one to decouple the "ready" signal, and the other to decouple "Valid" and "Data" signals. Actually we could directly use a Two-slot FIFO for timing decoupling. The structure of Two-slot FIFO is the same as that shown in **Figure 2**, with just 2 entries. In **Figure 2**, the output "Valid", "Data" and "Ready" signals are all generated from (read/write address pointers and "full") registers, so they are decoupled from the input side of the FIFO.

3. The Usage of Synchronous FIFOs for Intra-Processor Pipelines

Look into the pipeline design of a processor, we can find that synchronous FIFOs are extensively used.

3.1. Using Skip Buffer for Memory Reading Pipelines

Figure 7 shows the pipeline model of a digital processor [2]. There are two memory reading blocks in the pipeline stages, one for instruction fetching and the other for data reading. We only discuss the instruction fetching as an example.

At the instruction fetching stage, Program Counter (PC) is issued out to the program memory (PMEM) as address. In the meantime, the fetching command is delayed to the next stage, that is, the instruction queue. If both the read-back instruction, indicated by "RB_VLD" signal, and the fetching command, indicated by "RC_VLD" signal, are available, the read-back instruction is captured by the instruction queue. However, the PMEM is outside of the processor core and the read-back delay is out of control. It may come much later than the fetching command. In this case, the instruction queue will wait until both "RB_VLD" and "RC_VLD" are available.

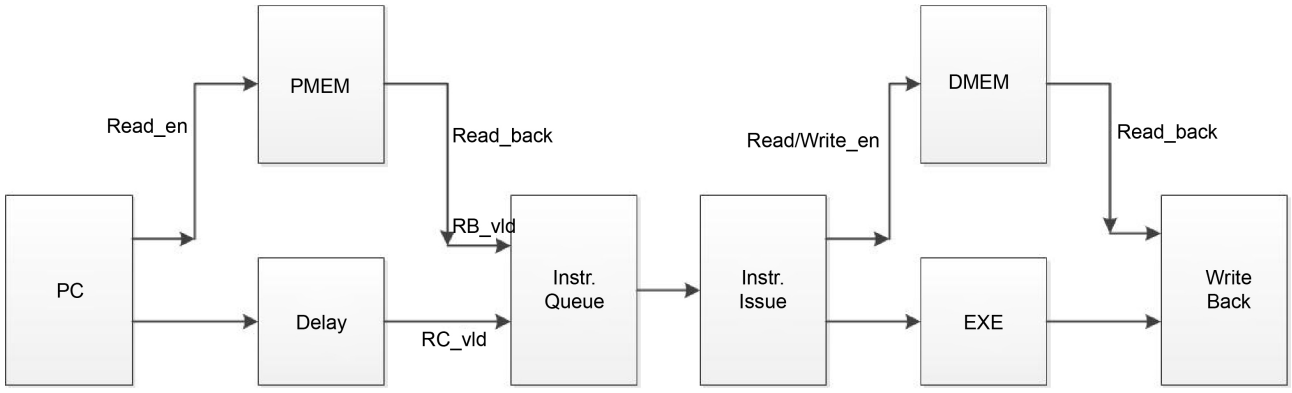


Figure 7. Pipeline model.

There is another case that makes things even difficult. The pipeline maybe blocked by later stages of the pipeline, e.g. by reading data memory blockage. In this case, the instruction queue may become full. If both "RB_VLD" and "RC_VLD" are coming at this time, the instruction queue cannot capture the read-back instruction. However, the address has been issued out and the PC has changed. In this case, the read-back data must be hold by the PMEM. This is to say, the PMEM cannot accept new addresses, which leads to 2 shortcomings: firstly there will be longer delay after the later stages recover; secondly the PMEM cannot be read by other modules (e.g. the Direct Memory Access, DMA module).

A skip buffer could be employed to overcome these shortcomings. As shown in **Figure 5**, in normal conditions, the one-slot FIFO in skip buffer is empty, and the read-back instruction bypasses the skip buffer and goes to the instruction queue directly. When the later stages of the pipeline are blocked, the back-pressure "RDY" signal from instruction queue will stop the PC being issued. However, there will be one fetching command has been issued out and the read-back instruction is on the fly. In this case, when the read-back instruction becomes available, it will be stored into the One-slot FIFO of the skip buffer and the PMEM will be released. In this way, the previous shortcomings are overcome. Of course, a two-slot FIFO could also serve this purpose [11].

3.2. Using Multiple One-Slot FIFO for Out-of-Order Issue

For high performance processors, there are lots of executive modules, e.g. integer multiplier, integer divider, floating adder, floating multiplier, floating divider, etc. Different modules may have different latencies (in clock cycles).

Refer to **Figure 7** again, in order to improve the performance of the processor, instructions could be issued out-of-order, and it is possible that more than one instructions are issued simultaneously [12].

Figure 8 shows the implementation schematic of out-of-order issue circuits. In **Figure 8**, instructions are orderly coming to a shifter FIFO, and are attached with a sequential identity (ID). Each FIFO in **Figure 8** is of one-slot. In the shifter FIFO, instructions either go into the executive modules (below in **Figure 8**),

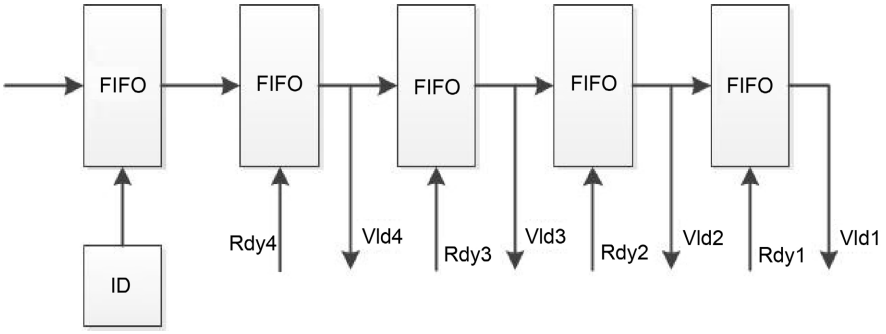


Figure 8. Out of order issue.

or go to the right slots. All the instructions in shifter FIFO are checked parallelly, any instruction whose source registers are available will be issued. In this way, instructions could be issued out-of-orderly and parallelly. After the execution of these instructions, they are committed (that is, write back the results to the register file) sequentially according to their ID.

3.3. Using Decoupling Skip Buffer or Two-Slot FIFO for Timing Decoupling

As shown in Figure 7 and Figure 8, there are many stages/modules in the pipeline. If all the stages/modules are separated by One-slot FIFOs, all the "READY" signals of every stage/module would be coupled together, which may lead to long combinational logic and critical timing path.

To overcome this circumstance, some One-slot FIFOs could be replaced with decoupling skip buffers or two-slot FIFOs. For example, it is highly recommended that the depth of the instruction queue being at least 2 slots, thus not only cutting both the forward "VALID" signal and the backward "READY" signal [11], but also facilitating PMEM reading.

4. The Usage of Synchronous or Asynchronous FIFOs for Inter-Processor Communications

In this section, the usage of synchronous or asynchronous FIFOs for inter-processor communications is discussed.

4.1. Using Synchronous or Asynchronous FIFOs as Inter-Processor mailing-box

Figure 9 shows the schematic that synchronous or asynchronous FIFO is used as inter-processor mailing-box. In Figure 9, there are 2 processors, Central Processing Unit 0 (CPU 0) and CPU 1, and a mailing-box. The mailing-box is implemented by synchronous or asynchronous FIFO, depending on the 2 CPUs are driven by the same or different clocks. CPU 0 writes data into the FIFO, the write enable signal is used as the "Valid_in" signal. When the FIFO is not empty, the "Valid_out" signal interrupts CPU 1. When CPU 1 reads out the data entry in FIFO, the reading command clears the interrupt signal simultaneously.

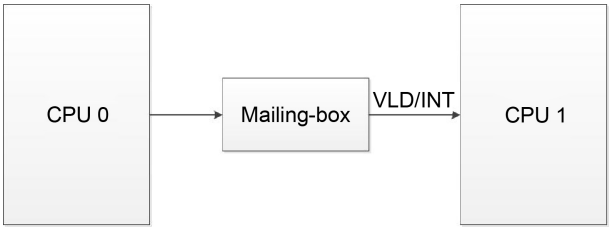


Figure 9. Mailing box.

It should be noted that mailing-box could be used not only between CPUs, but also between any master processing units (e.g. Finite-State-Machine, FSMs), or mixed CPU and other kind masters.

Figure 10 shows an example for inter-processor communications with asynchronous FIFOs. This is a baseband chip for satellite communications. In this chip, there are two CPUs, CPU 1 and CPU 2, CPU 1 controls the Transmitter (Tx), and CPU 2 controls the Receiver (Rx). There is a "Sync" signal from Rx to Tx, controlling the timing of sending signals according to the timing of the received signals. In **Figure 10**, the traffic interface is Gigabit Media Independent Interface (GMII). The received packets of GMII are stored into off-chip Synchronous Dynamic Random Access Memory (SDRAM) and CPU 1 is notified by interrupt signal "Eth_rx_int". On the other side, the Rx demodulated signals are stored into SDRAM and sent out through GMII interface. The interrupt signal "Eth_tx_int" of GMII to CPU 2 asks for more packets to be sent out.

This chip is for satellite communications. In satellite communication systems, Adaptive Modulation and Coding (AMC) is extensively employed to maximize the throughput of the wireless link. With AMC, the Rx side needs to measure the state (e.g. rain fading, length of buffered packets, etc.) of the wireless link, and returns this state to the Tx side through reverse link. Then the Tx side selects the best AMC scheme and processes the sending signals. With this mechanism, there are a few short messages from Rx module fed to the Tx module in every frame in the baseband chip. These messages are for link maintenance, not for traffic. It is called signalling. In **Figure 10**, these signalling messages are transferred from CPU 2 to CPU 1.

There are two properties for these signalling messages. Firstly, the lengths of these signalling messages are short; the average length of every message is 32 bytes in our case. Secondly, the number of these signalling messages is high. If every message interrupts CPU 1, the interrupt would be too frequent and there will be severe overhead in CPU 1 for context switch. As CPU 1 controls the Tx module and needs to be real-time, severe overhead would be dangerous.

In our implementation, an asynchronous FIFO based mailing-box is employed to transfer these short messages, as shown in **Figure 10**. The size of the asynchronous FIFO is of 1KX64, which can accommodate about 256 messages. In order to decrease the frequency of interrupt, two signals are employed as interrupt signals, one is "Vld" of the FIFO, which indicates the non-empty of the FIFO, and there are messages needing to be read, the other is the "Almost_full"

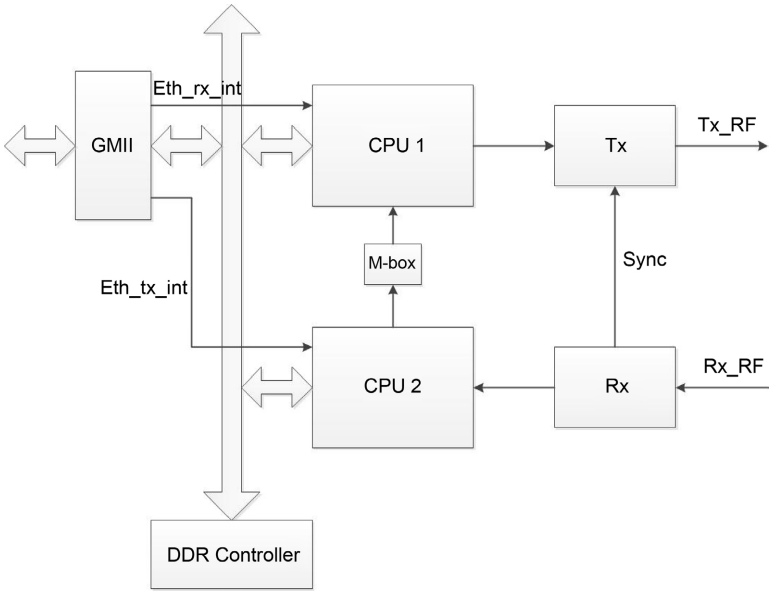


Figure 10. Example of using Mailing Box for inter-processor communications.

signal, which indicates the FIFO is almost full, and the messages should be read out emergently. Therefore, the "Almost_full" interrupt is of high priority. In out implementation, the "Almost_full" signal indicates that 75% of the FIFO is occupied. And every time it interrupts CPU 1, the CPU 1 reads all the messages in the mailing-box out.

In this implementation, the signalling bandwidth is 1 Mega-bits-per-second (Mbps), there are 4000 messages on average within one second. If there is no mailing-box, the messages need to be stored in SDRAM, and every message interrupts CPU 1, which means 4000 interrupts within one second on average, the average time interval between two interrupts is 0.25 minisecond. However, with this mailing-box and only "Almost_full" signal is interrupting CPU 1, there is only 21 interrupts within one second, and the interrupt interval is 48 minisecond on average. From this example, we can see that the usage of FIFO greatly decreased the frequency of interrupts and thus facilitated inter-processor communications.

4.2. Using Synchronous or Asynchronous FIFOs as Inter-Processor Packet Router

Figure 11 shows the architecture of a many-core processor [4] [13] [14]. In **Figure 11**, every 4 Processing Elements (PEs) are clustered together, called Quad PE (QPE). 32 QPEs are connected together with a Network-on-Chip (NoC) of 6 by 6 mesh topology. Every QPE is connected to a NoC Router, which is indicated as "R" in **Figure 11**. The links between every two NoC Routers are called L1. The links between PEs and NoC Routers are called L2, and the links between every two PEs within a QPE are called L3. In the center of the mesh NoC, there are synchronizers and shared memories facilitating on-chip inter-processor communications. At the 4 corners of the chip, 4 Double Data Rate (DDR) controllers

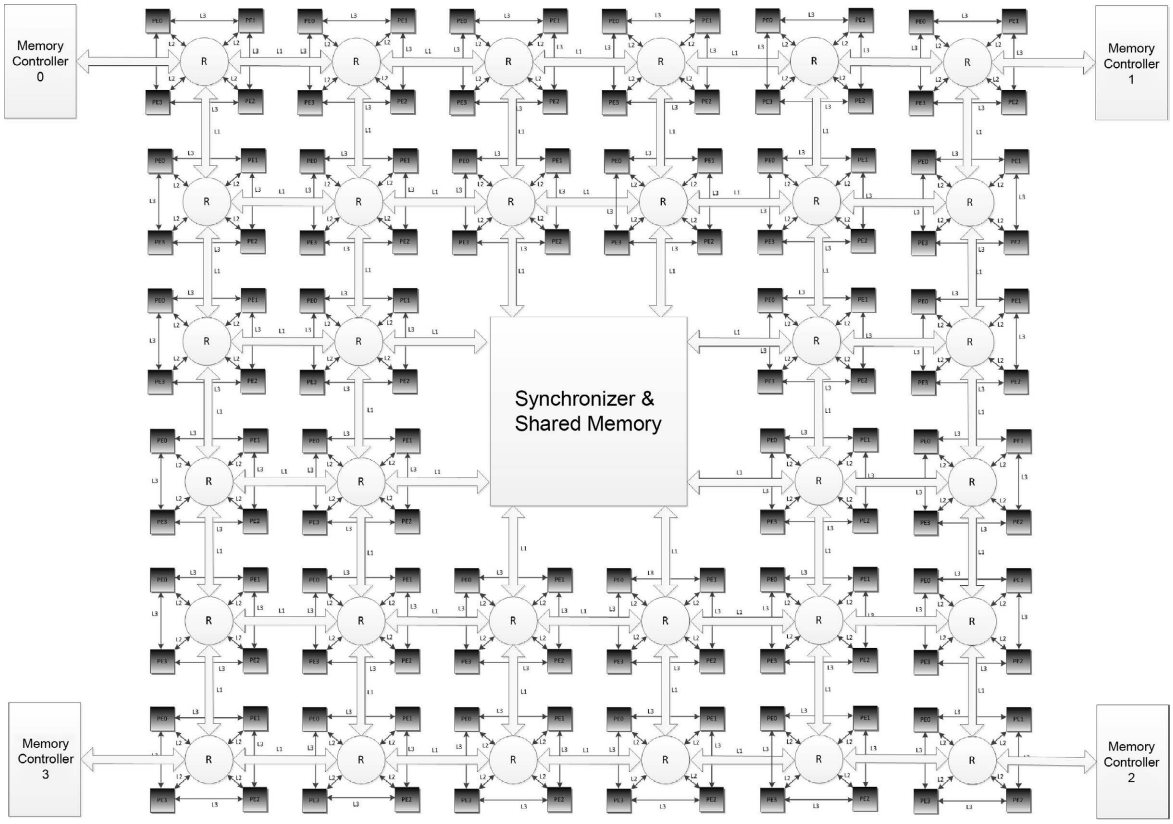
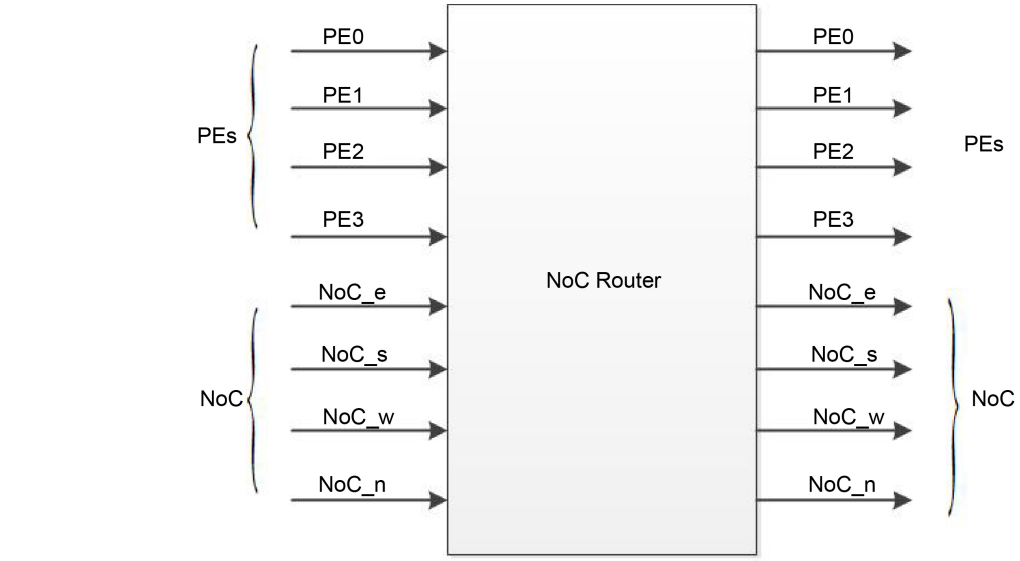


Figure 11. The architecture of a many-core processor.

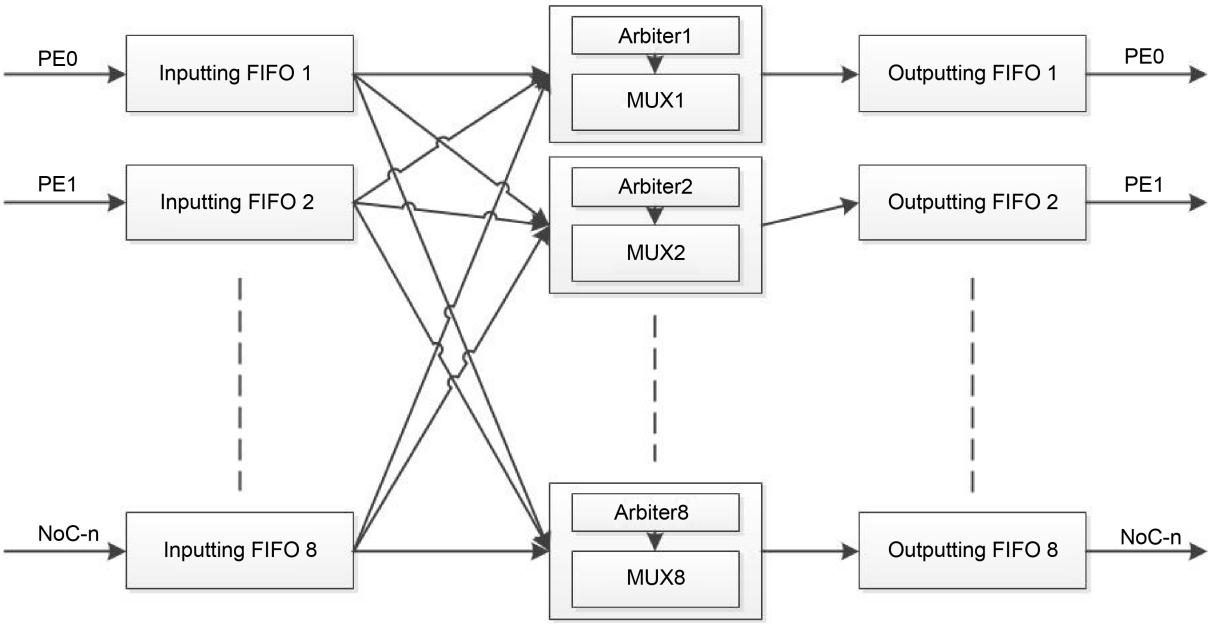
are integrated to connecting to off-chip SDRAMs, each at one corner. In **Figure 11**, 128 PEs are integrated in a chip with this mesh topology. Obviously this architecture could be extended to integrate more PEs within a die, and if necessary, more DDR controllers could be employed to even increase memory access bandwidth.

Figure 12(a) shows the interface of NoC Router, and **Figure 12(b)** shows the architecture of NoC Router [4]. In **Figure 12(a)**, there are 8 input ports, 4 for PEs and another 4 for {East, South, West and North} L1 links. Correspondingly, there are 8 output ports for the NoC Router, also 4 for PEs and 4 for L1 links. In **Figure 12(b)**, the input ports are firstly buffered by FIFOs. The output of the inputting FIFOs are then arbitrated and routed (by MUX) to the outputting FIFOs, thus sending packets to the next stage.

In **Figure 12(b)**, the FIFOs in NoC Router could be synchronous or asynchronous FIFOs. Synchronous FIFOs are with less latencies thus quicker communication (especially reading) response time. Asynchronous FIFOs are with better power efficiency and could alleviate the difficulty of clock tree routing for big chips [5]. For low power embedded many-core processors, where power is more cared about than performance, asynchronous-FIFO-based NoC is recommended [5] [13]. However, for big chips aiming at applications in High-Performance-Computers (HPCs), where latency is the main concern, synchronous-FIFO-based NoC is preferred as long as the timing is allowed [15] [16].



(a) Interface of NoC Router



(b) Architecture of NoC Router

Figure 12. Interface and architecture of NoC router.

5. The Usage of Synchronous or Asynchronous FIFOs for System-on-Chips

5.1. Usage of Synchronous or Asynchronous FIFOs for Peripherals

Figure 13 shows a picture in which synchronous or asynchronous FIFOs are used for peripherals. In Figure 13, data are put into tx_FIFOs and the peripherals take data out and transmit them out. When the tx_FIFOs are almost empty, it interrupts the CPU and asks for more data. For the receiving direction, the data from peripherals are put into rx_FIFOs. When the rx_FIFOs are almost full, it interrupts the CPU and CPU takes data out from it.

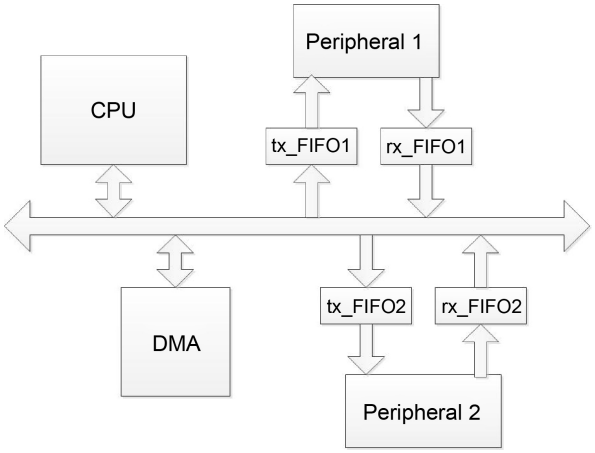


Figure 13. Using FIFOs for peripherals.

In **Figure 13**, the tx_FIFOs and rx_FIFOs could be synchronous or asynchronous FIFO. When the peripherals and the bus use the same clock, the tx_FIFOs and rx_FIFOs are synchronous, when the peripherals and the bus use different clocks, tx_FIFOs and rx_FIFOs are asynchronous.

5.2. Usage of Synchronous FIFOs for Interleaved AXI Reading and Writing

For AXI bus [7] [16], there could be multiple masters issuing reading commands or writing commands, and the commands from different masters are interleaved. Each command requests multiple reading or writing data packets. The data packets belonging to one command is called a burst. The last data packet within a burst is always indicated by a "last_data" flag. The length of the burst means the number of packets within a burst. To differentiate these masters, these masters are with different Master IDentities (MIDs). For AXI bus, the timing of the address channel and data channel are independent, which enables outstanding reading or writing. With outstanding reading, multiple reading commands could be issued before their corresponding reading-data are back. With outstanding writing, multiple writing commands could be issued before the corresponding writing-data are sent out.

However, as AXI is an in-order bus, that is, it must keep the order of data in reading and writing, it is very dangerous in implementing outstanding reading and writing. For example, if multiple reading commands with MID 2 following MID 1 are sent out, but there are not enough data buffer in MID 1, the read-back data belonging to MID 1 will be blocked at the read-data channel of the slave, thus blocking the read-back data belonging to MID 2 coming back to the master. For writing, if multiple writing commands with MID 2 following MID 1 are sent out, but there are not enough data buffer for MID 1 in the slave, the writing-data of MID 1 will be blocked at the writing-data channel of the slave, thus blocking writing-data belonging to MID 2 going to the slave. If this reading-blockage or writing-blockage occurs, the efficiency of the AXI bus will be

greatly deteriorated.

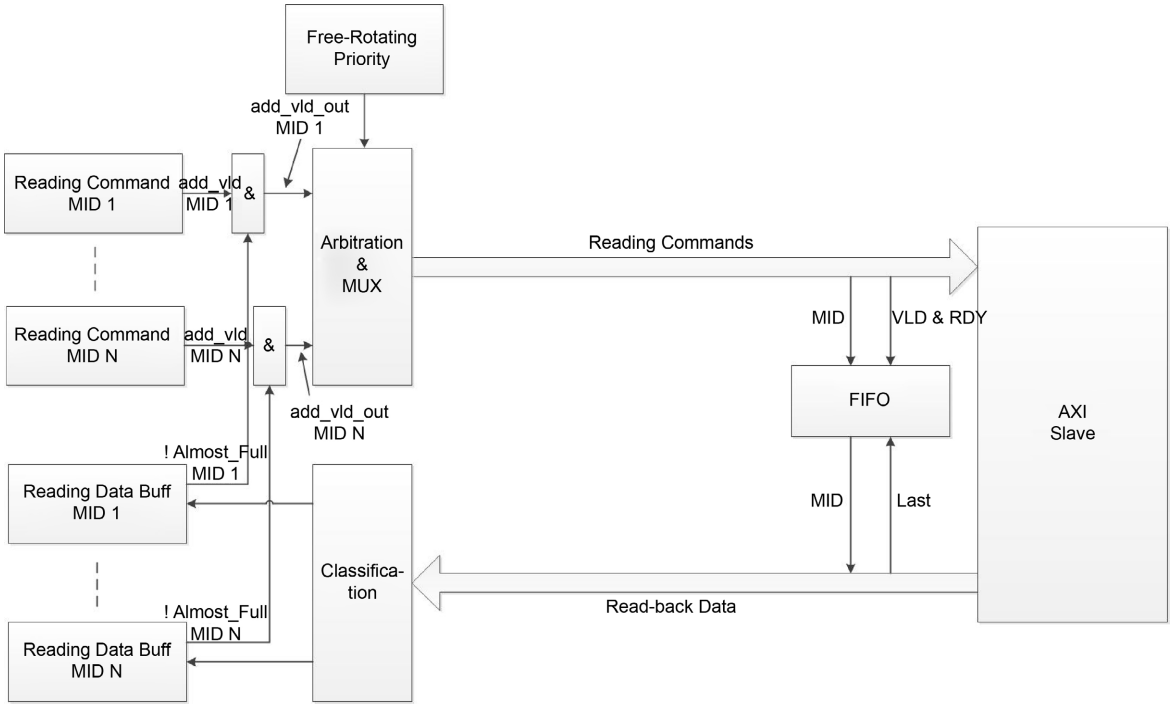
Figure 14(a) shows the idea to avoid reading blockage. In Figure 14(a), there is a flag "Almost_full" showing the status of read-back data buffer in each MID, which indicates how many slots are available to store incoming packets in the read-back data buffer. When a reading command (indicated by addr_vld MID x) is sent out, it is ANDed with its corresponding read-data buffer status. The "Almost_full" flag is inverted and ANDed with it. If we design that the "Almost_full" to be no less than the length of one AXI burst, the resulted "addr_vld_out MID x" will make sure that when the reading command is sent to the slave, the read-back data will always be accepted by the corresponding master "MID x". After the reading command is sent out, the status of the read-data buffer is updated. On the other side, when the data are taken out from the data buffer by the master, the status is updated again to release the buffer for reading more data.

In Figure 14(a), multiple valid "addr_vld_out MID x" are arbitrated, and one is selected to the slave. At the slave, the selected MID are stored in a FIFO, and attached to the read-back data. The "last_data" flag in a burst is used to pop out the MID from the FIFO. At the masters, the MID attached with the read-back data are used to recognize which master the data are belonging to. This recognization is realized by "Classification" module in Figure 14(a).

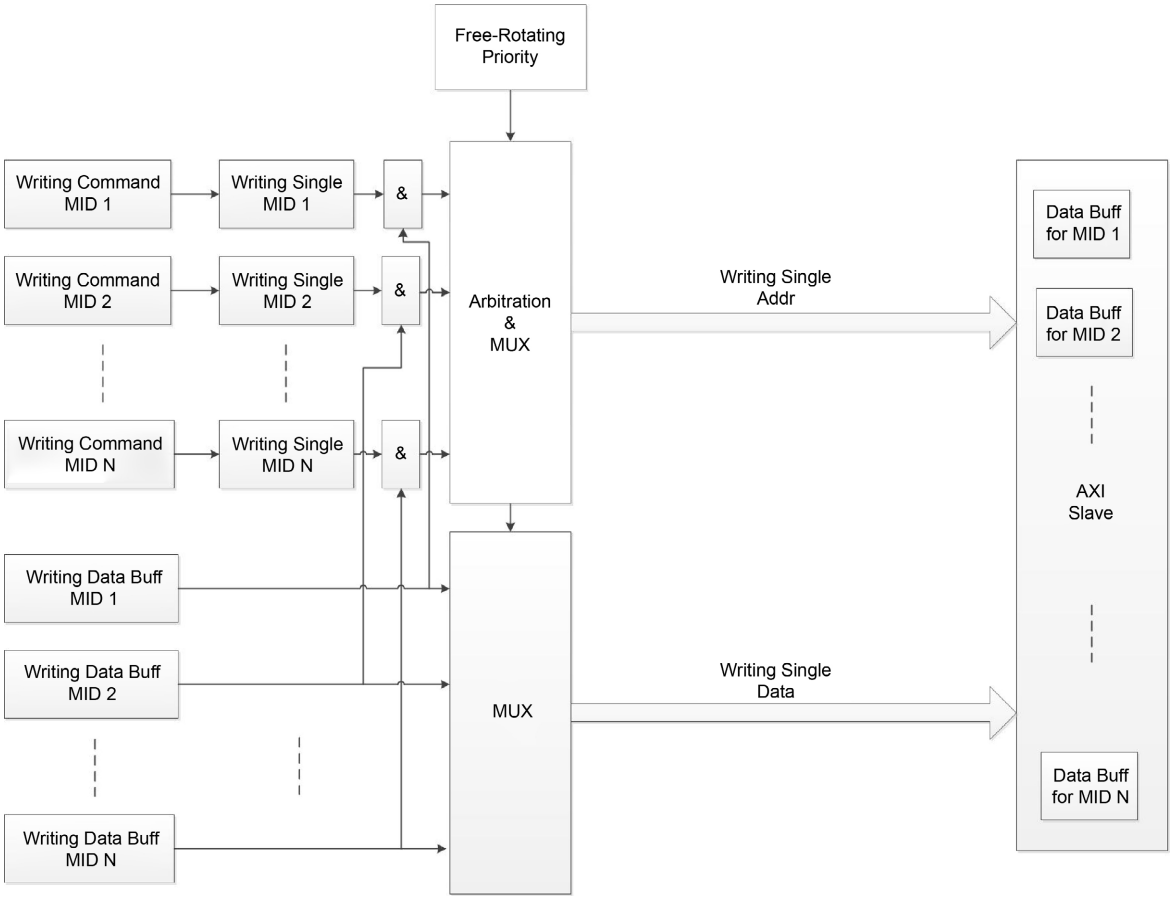
Figure 14(b) shows the idea to avoid writing blockage. In Figure 14(b), multiple masters with different MIDs are issuing writing commands. These writing commands may request writing multiple data packets. Firstly the "Writing Single MID x" module transforms the writing commands into multiple "Writing Single" commands, which only request writing single data packet. Then the "Arbitration & MUX" module selects one "Writing Single" command from multiple MIDs, and sends this command to the salve. In the mean time, one data packet corresponding to the selected MID is also selected out and sent to the slave. In order to avoid the case when a "Writing Single" command is sent out, but the corresponding data packet is not yet ready (that is, the "data_vld" is not asserted), the "vld" signal of the "Writing Single" command is ANDed with the "vld" signal of the corresponding data packet before going to the arbitrator.

It is worth noting that the priority of the arbitrator must be free-rotating, see Figure 14(a) and Figure 14(b) the "Free-Rotating Priority" module. If MID x is selected by the arbitrator, however, the data buffer for MID x in slave is full, the transfer from MID x to the slave cannot be accomplished. At this time, if the priority of the arbitrator is free-rotating, other MIDs will be selected and tried. In this way, the AXI bus will never be blocked by any one master, thus the bandwidth of the bus is sufficiently utilized. On the contrary, if the priority-rotating depends on the accomplishment of a transfer, blockages may occur.

Figure 14(a) and Figure 14(b) could be used to combine multiple AXI master ports into one AXI master port. In this case, in Figure 14(a) and Figure 14(b), after the AXI slave receives a reading or writing command, it will initiate a new reading or writing command to the next stage, acting as a new AXI master.



(a) Interleaved Reading



(b) Interleaved Writing

Figure 14. AXI interleaved Reading and Writing.

5.3. Usage of One-Slot-FIFO-Based Circular Buffer for AXI Bus Data Alignment

AXI bus supports unaligned data transfers. If the width of data channel is W bytes, but the reading or writing address is not W-byte aligned, unaligned transfer occurs. In unaligned transfer, the first data packet transfers the non-aligned part of the data stream, so that all the addresses of following packets are W-byte aligned. Figure 15 shows the data on read/write data channel of unaligned transfer. The numeric digits in Figure 15 are the addresses of the valid bytes. In Figure 15, the reading/writing address starts at 0x0003, the width of the data channel is 64-bit (8 bytes), the first data packet is of 5 bytes (address 0x0003 to 0x0007), and all the addresses of the following data packets are 8-byte aligned. Because of the length of the data stream, only 6 bytes in the last packet are valid. For writing, these valid bytes are indicated by "w_strb" signal of the write data channel. For reading, the master knows the valid bytes in the last packet by itself.

Referring to Figure 16, there is a DMA controller, who moves data between memory addresses. The DMA controller is composed of a top-level controller module "DMA Controller Top", a memory reading controller "DMA Reading", a memory writing controller "DMA writing", and a "Data Alignment" module. We assume it reads data starting from address "addr_r", and writes data starting from "addr_w", both "addr_r" and "addr_w" are unaligned with W bytes, and the reading and writing are with different sized first packet, e.g., "addr_r" starts at 0x0003 while "addr_w" starts at 0x0002, we therefore need to decompose and re-assemble every data packet so that the data packets on the read/write data channels of the bus are in accordance with the AXI specification. This decomposition and re-assembly process is called data alignment.

Figure 17 shows the implementation of this data alignment. In Figure 17, D

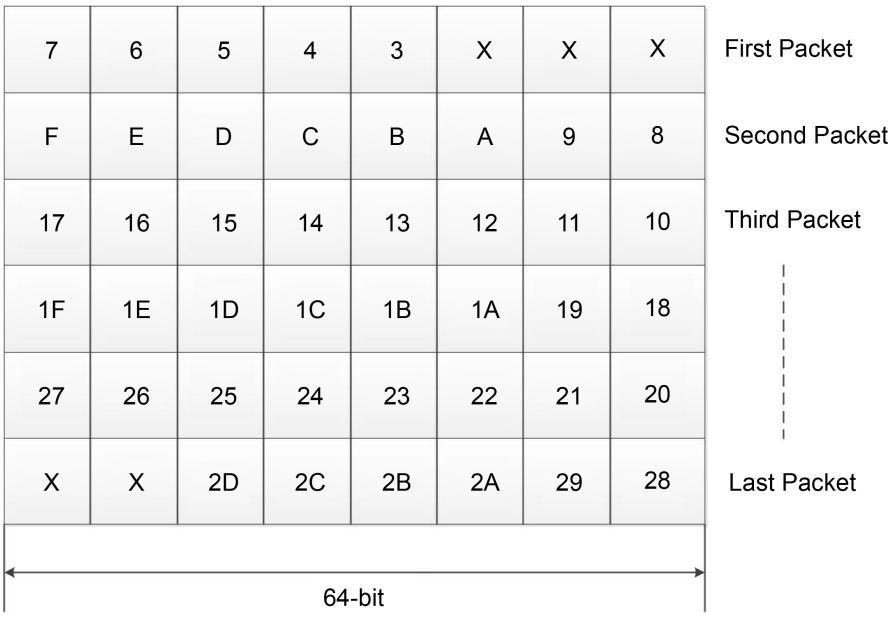


Figure 15. The data on read/write data channel of unaligned transfer.

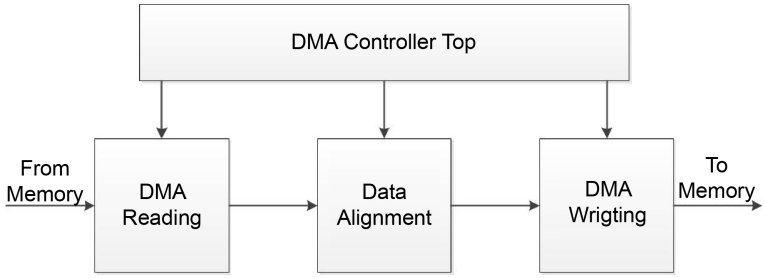


Figure 16. Data alignment in DMA controller.

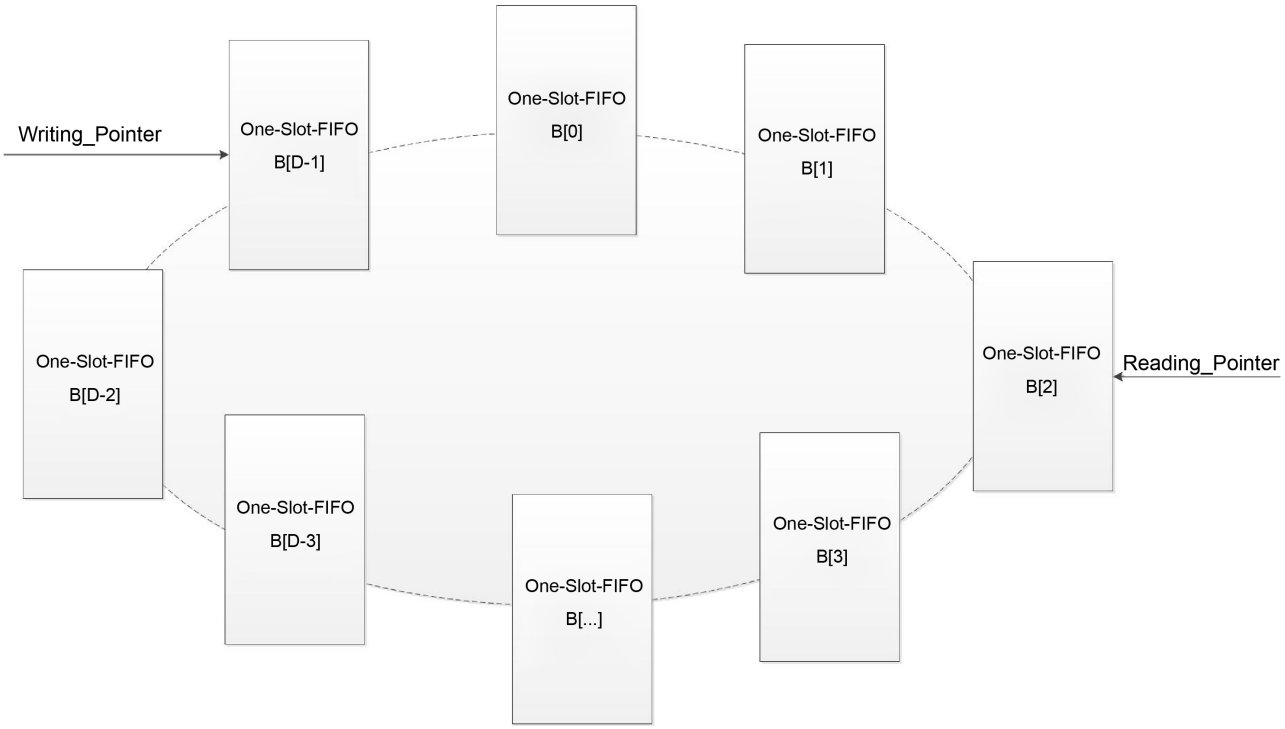


Figure 17. Using circular one-slot-FIFOs for data alignment.

One-slot-FIFOs, denoted as B[0], B[1], ..., B[D − 1], are circularly connected and form a circular buffer. Every One-slot-FIFO is of one-byte width. There are two address pointers, one for writing data into the circular buffer, named as "Writing_Pointer", and one for reading data from the circular buffer, named as "Reading_Pointer". The length between "Writing_Pointer" and "Reading_Pointer" indicates the number of slots (that is, number of bytes) being occupied, this is denoted as "Len_used". The length of buffer which is available for incoming data is denoted as "Len_available", which is the total length of the circular buffer minus the length being occupied, that is,

$$\text{Len\_used} = (\text{Writing\_Pointer} - \text{Reading\_Pointer}) \% D$$

$$\text{Len\_available} = D - \text{Len\_used}$$

In the above equation, "%" means modulo.

At reset, both "Writing_Pointer" and "Reading_Pointer" point to One-slot-FIFO B[0]. When a data packet is read out from memory by the DMA control-

ler, it is written into this circular buffer, and the "Writing_Pointer" is updated. After that, when a new packet is read out by the DMA controller, the length of the packet is compared with "Len_available" of the circular buffer. Whenever "Len_available" is bigger than the length of the packet, this packet is put into the circular buffer, and the "Writing_Pointer" is updated.

At the other side, when the DMA controller is trying to write a data packet into the memory, it requests data from this circular buffer to form the data packet. If the length of the requested data packet is smaller than "Len_used", the data in circular buffer are taken out to form this data packet, and the "Reading_Pointer" is updated. After that, when forming a new packet, new data are requested from this circular buffer. Whenever "Len_used" is bigger than the requested length, the data are taken out and "Reading_Pointer" is updated. However, if "Len_used" is smaller than the requested length, we have to wait until new data packets are put into the circular buffer, and "Len_used" is updated.

There are two points worth of noting:

1) When writing or reading the circular buffer, it is possible that multiple bytes are written into or taken out from the circular buffer simultaneously, and the number of bytes being written or reading is changing. Therefore, one Static Random Access Memory (SRAM) macro block doesn't work; we therefore form the circular buffer with multiple One-slot-FIFOs of one-byte width.

2) The minimum depth of the circular is 2*W − 1, that is,

$$\text{Min}\{D\} = 2*W - 1$$

If the depth of circular buffer is less than 2*W − 1, it is possible that there are not enough data to form the requested data packet, while new data packet can't be put into the circular buffer because there are not enough buffer for this incoming data packet, thus the circular buffer can't either be read or written, and therefore deadlock occurs.

5.4. Usage of Asynchronous FIFOs for Clock-Domain Crossing

5.4.1. Usage of Asynchronous FIFOs for AXI Bus Bridge

As mentioned in the previous section, for AXI bus, the read address channel, the read data channel, the write address channel, the write data channel and the write response channel are independent. And as shown in section II.A, the data flow of FIFO is unidirectional. Therefore, an asynchronous FIFO could be used for every channel for clock domain crossing. In this way, clock domain could be easily separated.

5.4.2. Usage of Asynchronous FIFOs for AHB Bus Bridge

For Advanced High Performance Bus (AHB bus), the timing of address and data are tightly coupled. When an address is issued for a writing, the data should tightly follow it with one-clock-tick delay. Therefore, we use an one-slot FIFO to delay the address, so that it is aligned with the data. Then the aligned address and data are put into an asynchronous FIFO to cross the clock domain. At the other side, an one-slot FIFO is used again to delay the data for one-clock-tick after

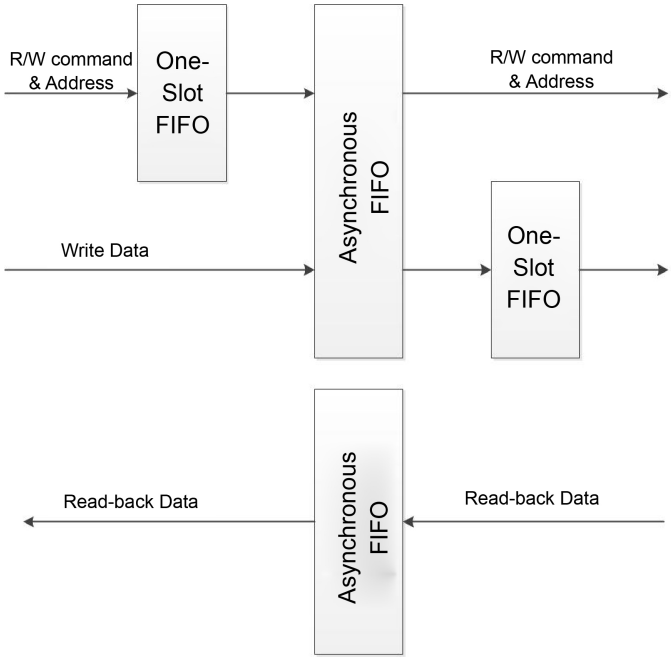


Figure 18. Using asynchronous FIFOs for AHB clock bridge.

the address.

For AHB reading, only address is put into the asynchronous FIFO, and after the FIFO, the read-back data are put into another asynchronous FIFO whose data flows in another direction. After the reading data cross the asynchronous FIFO, the "Valid_out" signal serves as the "hready" signal of the AHB reading command.

The block diagram of AHB clock bridge is shown in **Figure 18**.

6. Conclusions

Synchronous FIFOs and asynchronous FIFOs are extensively used in contemporary digital systems. In this paper, we addressed the various implementations of synchronous and asynchronous FIFOs. We also addressed the four special cases of synchronous FIFOs: One-slot FIFO, Skip Buffer, Decoupling Skip Buffer, and Two-slot FIFO. We showed their sophisticated usage in the pipeline design of processors. For inter-processor communication networks, we showed that synchronous and asynchronous FIFOs could be used as mailing-box, and in NoCs, asynchronous FIFOs could not only undertake inter-processor communications, but also facilitate clock-domain-crossing. At the whole SoC level, synchronous and asynchronous FIFOs are not only used in peripherals, but also used for AXI masters combining, and for AXI and AHB clock-domain crossing. From the review of this paper, we can find that FIFOs play an important role in digital design, and are used at all levels of nowadays digital systems. Therefore, we should pay more attentions to the usage of FIFOs in our future work.

In the future, more design patterns will be explored and more advanced design methodologies will be proposed.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Mano, M.M. and Ciletti, M.D. (2017) Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog. 6th Edition, Pearson, New York.

[2] Hennessy, J.L. and Patterson, D.A. (2017) Computer Architecture: A quantitative Approach. 6th Edition, Morgan Kaufmann, Cambridge, MA.

[3] Gordon-Ross, A., Abdel-Hafeez, S. and Alsafrjalni, M.H. (2019) A One-Cycle FIFO Buffer for Memory Management Units in Manycore Systems. *IEEE Computer Society Annual Symposium on VLSI* (*ISVLSI*), Miami, FL, 15-17 July 2019, 265-270. https://doi.org/10.1109/ISVLSI.2019.00056

[4] Hu, D., Shang, D., Zhang, Y., *et al.* (2022) Timing and Area Optimized Re-Configurable Network-On-Chip Router. *Journal of Xidian University*, **49**, 125-134.

[5] Hoppner, S., Yan, Y.X., Dixius, A., *et al.* (2021) The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing. https://arxiv.org/pdf/2103.08392.pdf

[6] (2017) ARM, Amba axi 5.0. https://developer.arm.com/architectures/system-architectures/amba/amba-5

[7] Jiang, Z., Audsley, N., Shi, D.Y., *et al.* (2021) Brief Industry Paper: AXI-Interconnect^{RT}: towards a Real-Time AXI-Interconnect for System-on-Chips. *IEEE 27th Real-Time and Embedded Technology and Application Symposium* (*RTAS*), Nashville, TN, 18-21 May 2021, 437-440. https://doi.org/10.1109/RTAS52030.2021.00046

[8] Wang, S., Xu, Y., Tang, J., *et al.* (2021) Design of Asynchronous FIFO Controller Based on FPGA. *International Core Journal of Engineering*, **7**, 153-159.

[9] Wielage, P., Marinissen, E.J., Altheimer, M., *et al.* (2021) Design and DFT of a High-Speed Area-Efficient Embedded Asynchronous FIFO. *International Core Journal of Engineering*, **7**, 153-159.

[10] Shibata, N., Watanabe, M., Tanabe, Y., *et al.* (2002) A Current-Sensed High-Speed and Low-Power First-In-First-Out Memory Using a Wordline/Bitline-Swapped Dual-Port SRAM Cell. *IEEE Journal of Solid-State Circuits*, **37**, 735-750. https://doi.org/10.1109/JSSC.2002.1004578

[11] https://github.com/openhwgroup/cv32e40p/blob/master/rtl/cv32e40p_prefetch_buffer.sv

[12] Moreshet, T. and Iris Bahar, R. (2004) Effects of Speculation on Performance and Issue Queue Design. *IEEE Transactions on Very Large Scale Integration* (*VLSI*) *Systems*, **12**, 1123-1126. https://doi.org/10.1109/TVLSI.2004.834226

[13] Yu, Z.Y., Meeuwsen, M.J., Apperson, R.W., *et al.* (2008) AsAP: An Asynchronous Array of Simple Processors. *IEEE Journal of Solid-State Circuits*, **43**, 695-704. https://doi.org/10.1109/JSSC.2007.916616

[14] Abdelhadi, A.M.S. and Li, H. (2021) Enabling Mixed-Timing NoCs for FPGAs: Reconfigurable Synthesizable Synchronization FIFOs. *International Conference on Field-Programmable Logic and Applications*, **43**, 312-318. https://doi.org/10.1109/FPL53798.2021.00062

[15] Fariborz, M. and Ben Yoo, S.J. (2022) High Throughput Memory with Silicon Photonics in Chiplet-Based Architectures for Irregular Workloads. 27*th OptoElectron-*

ics and Communications Conference (*OECC*) *and* 2022 *International Conference on Photonics in Switching and Computing* (*PSC*), Toyama, 3-6 July 2022, 1-3. https://doi.org/10.23919/OECC/PSC53152.2022.9849864

[16] Fischer, T., Rogenmoser, M., Cavalcante, M., *et al.* (2023) FlooNoC: A Multi-Tb/s Wide NoC for Heterogeneous AXI4 Traffic. *IEEE Design & Test*, **40**, 7-17. https://doi.org/10.1109/MDAT.2023.3306720