Computer Organization and Architecture II: Parallel Output Controller

Leo

March 3, 2025

Contents

1 Purpose

The purpose of this project is to design and simulate a parallel output controller (POC) which acts an interface between system bus and printer. The Xilinx Vivado is provided for simulation.

2 Task

POC is one of the most common I/O modules, namely the parallel output controller. It plays the role of an interface between the computer system bus and the peripheral (such as a printer or other output devices).

The handshaking process is described as follows: When the printer is ready to receive a character, it holds RDY=1. The POC then hold a character at PD (parallel data) port and generate a pulse at the port TR (transfer request). The POC will change TR to 0 when detecting printer has responded TR, i.e., RDY has been changed from 1 to 0. When detecting the effective TR signal, the printer will change RDY to 0, take the character at PD and hold the RDY at 0 until the character has been printed (e.g., 5 or 10ms), then set RDY=1 again when it is ready to receive the next character. (Suppose the printer has only a one character "buffer" register, so that each character must be printed before the next character is sent).

The buffer register BR is used to temporarily hold a character sent from the processor, which character will be transferred to the printer later. The status register SR is used for two control functions: SR7 serves as a ready flag to indicate POC is ready or not to receive a new character from the processor, and SR0 is

used to enable the interrupt requests sent by POC. In interrupt mode, If SR0=1, then POC will send an interrupt request signal to processor when it is ready to receive a character (i.e., when SR7=1). If SR0=0, then POC will not interrupt. The other bits of SR are not used and empty.

In polling mode, SR0 is always 0. The processor selects SR by accessing the relative address, then reads SR register. If SR7=1, the processor selects BR and writes a character into BR, then processor clears SR7 to 0 to indicate that the new character has been written into BR and not printed yet. When POC detects that SR7 is set to 0, POC then proceeds to start the handshaking operations with the printer. After sending character to printer, POC sets the SR7 to 1, which indicates POC is ready to receive another character from the processor. The transfer cycle can now repeat. During the handshaking operations between POC and printer, the processor continues to fetch and execute instructions. If it happens to read SR, it will find SR7=0 and hence will not attempt to send another character to the POC.

In interrupt mode, SR0 is always 1. After sending character to printer, POC sets the SR7 to 1, since SR0=1, the interrupt request signal (IRQ) is set to 0, which indicate an effective interrupt signal to the processor. When the processor detects the effective IRQ signal, the processor directly selects BR and writes a character into BR, and then the processor sets the SR7 to 0, which indicates that the new character has been written into BR and not printed yet. When POC detects that SR7 is set to 0, POC then proceeds to start the handshaking operations with the printer. After sending character to printer, POC sets the SR7 to 1, which indicates POC is ready to receive another character from the processor. The transfer cycle can now repeat. During the handshaking operations between POC and printer, the processor does not try to access POC until it receives the interrupt request signal.

3 Design Description

3.1 Design Analysis

3.1.1 Processor

According to the description above, the finite state machine of the processor can be designed as follows:

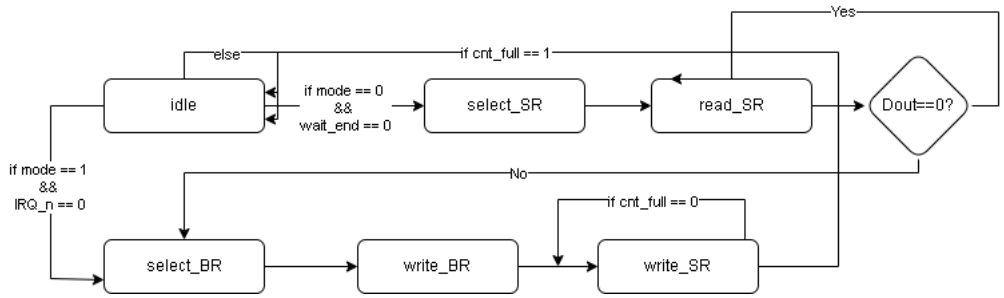

Figure 1: The finite state machine of the processor.

The states from *idle* to *write_SR* can be encoded by 0 to 5 for sake of brevity. Between one state and another are certain actions, which are shown below:

1. State 0 to state 1: set RW to 0, ADDR to 0. If reset signal is given, set polling wait flag to 0, else set it to 1.

2. State 1 to state 2: set polling wait flag to 0, RW to 0 and ADDR to 0. If bus wait delay is over, set bus wait enable flag to 0, else set it to 1.

3. State 3 to state 4: set RW to 1 and ADDR to 1. If bus wait delay is over, set bus wait enable flag to 0, else set it to 1. Start to get the letter to be transferred.

4. State 4 to state 5: If bus wait delay is over, set bus wait enable flag to 0, else set it to 1. Put the letter to port Din.

5. State 5 to state 0: If bus wait delay is over, set bus wait enable flag to 0, else set it to 1. Set the left-most 7 bits of SR to 0 and the right-most bit to its previous mode. Set RW to 1 and ADDR to 0.

Note that from state 2 to state 3 there is no action to be executes since reading from SR only acts as an interval state before POC is ready.

Beside the finite state machine, some extra registers should also be declared so that the flow control can be much easier. The Processor should be able to switch between two modes, i.e., the poll mode and the interrupt mode. So a one-bit mode register is necessary. In poll mode, the processor attempts to transfer a byte to POC periodically after a certain amount of time. So a counter should be assigned to the polling process, which gives a positive signal every time the predetermined delay is over. Besides, to mimic real world printing behavior, I designed a content generator called *out_src*, which gives out a character whenever being activated. The content starts at letter 'a' and ends at letter 'z', after which a new period begin again. Finally, to improve the robustness of the process, the signals on the bus should be maintained for a period of time, which yields the design of the counter called *bus_wait*.

3.1.2 POC

The finite state machine of POC can be depicted as the following flow chart diagram.

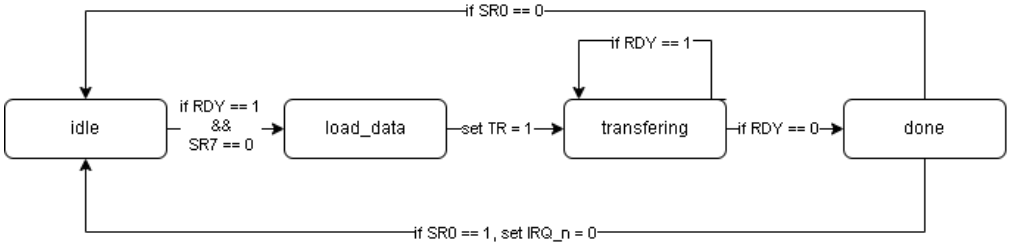

Figure 2: The finite state machine of POC.

Similar to the processor, the POC also needs some internal registers to facilitate the design. The SR is designed to record the state of POC. The left most bit indicates if the POC is ready to receive a byte from the processor; the right most bit indicates the operation mode, which is read-only for the processor. Besides, I also instantiated a counter to assure sufficient bus time for every signal.

The actions between states are defined in the following list.

1. State *idle* to state *load_data*: set $IRQ\_n = 1$. If RW == 0 and ADDR == 0, put the content in SR onto Dout; if RW == 0 and ADDR == 1, put the content in BR onto Dout; if RW == 1 and ADDR == 0, update the left most 7 bits in SR with the content in Din[7:1]; else, update BR with Din.

2. State *load_data* to state $transfering$: If bus wait delay is over, set bus wait enable flag to 0, else set it to 1. Put the content in BR to PD, set TR to 1.

3. State *done* to state *idle*: If bus wait delay is over, set bus wait enable flag to 0, else set it to 1. Set SR[7] to 1, TR to 0. If in poll mode, set $IRQ\_n = 0$, else set $IRQ\_n = 1$.

3.1.3 Printer

The finite state machine of the printer shown in figure 3 is much easier.

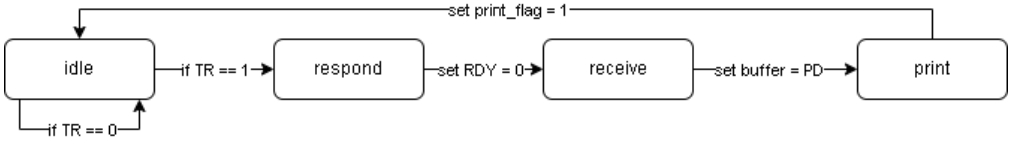

Figure 3: The finite state machine of the printer.

Additionally, I defined a buffer to store the byte to be printed, which is updated by PD. A print flag is also provided so that the printer can determine if the print operation is done.

1. State *respond* to state *receive*: set $RDY = 0$. Update buffer with PD.

2. State *receive* to state *print*: raise the print flag.

3.2 Overall Connection

The overall connection of the design is shown in figure 4.

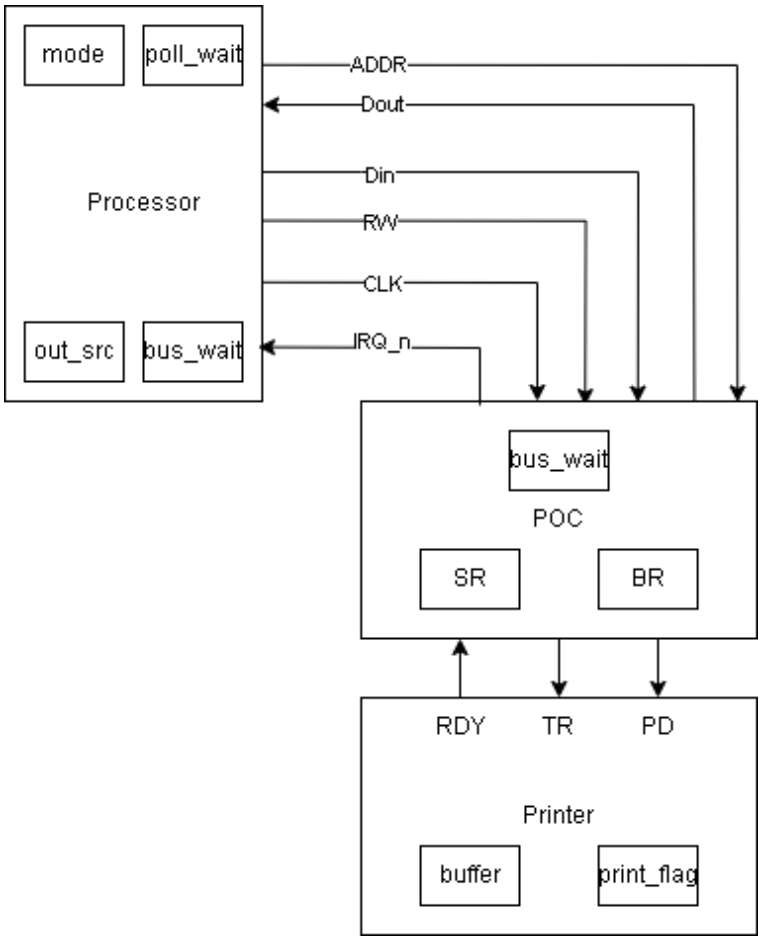

Figure 4: The overall connection of the design.

4 Simulation

The simulation follows a bottom-up rule. That is to say, every submodule is independently tested before assembly. The submodule simulation is presented first in this section and top module simulation is given subsequently.

4.1 Submodule Simulation

4.1.1 Processor

Note that the processor is able to work in both modes, but cannot switch during a certain task period. So for convenience, I instantiated two processors, namely proc1 and proc2, one works in poll mode and one works in interrupt mode. They are fed with identical input signals and their outputs are inspected respectively.

The waveform of proc1 is shown in figure 5.

Some special time points have been marked, and their meanings are explained as follows

1. In poll mode, the predetermined wait cycle is over, so processor actively queries SR7.
2. SR7=1 (according to Dout), indicating that POC is idle and processor continues to operate.
3. Select BR.

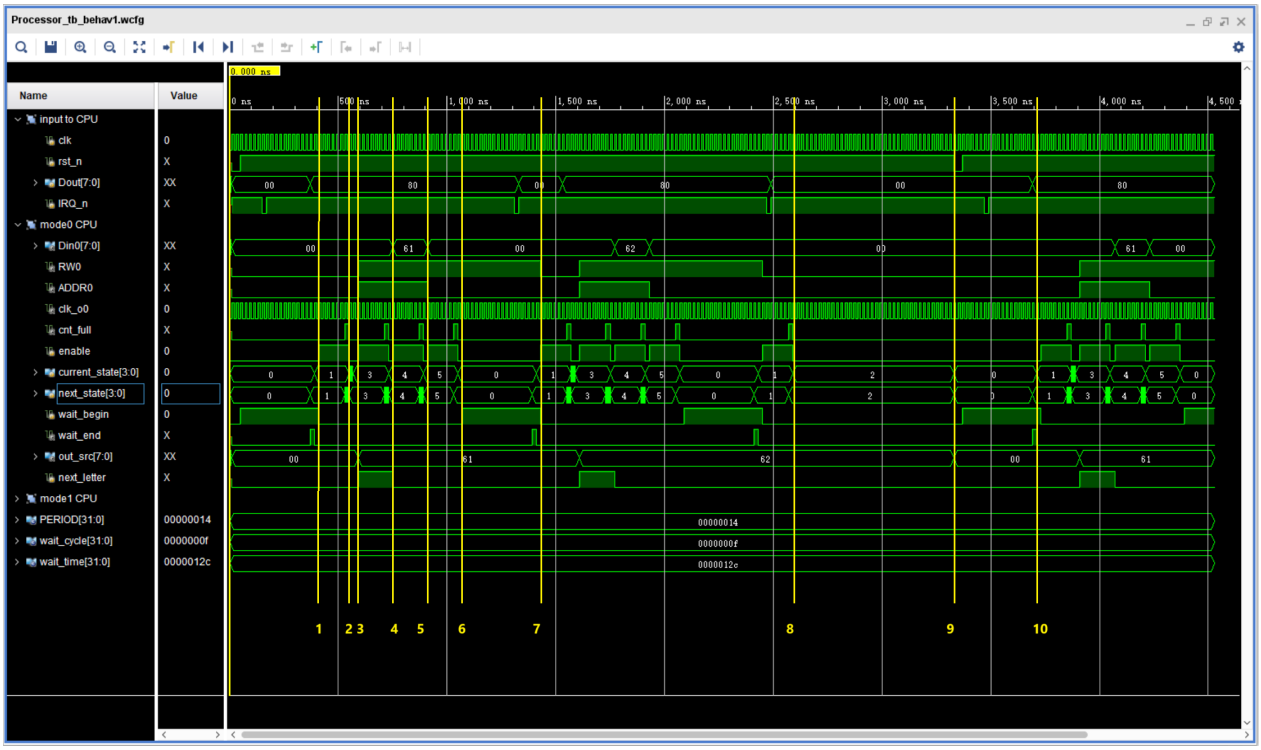

Figure 5: The waveform of the processor working in poll mode.

4. Write BR, Din='a', which is the word that the processor wants to write to POC.
5. Set SR7 to 0.
6. Return to idle state, start to wait.
7. Timing ends and a new round of transmission begins.
8. The processor attempts to execute the third round of transmission, but SR7 was found to be 0, so the processor cannot continue.
9. Reset.
10. The fourth round of transmission is proceeding normally.

The interrupt signal is also given, showing that in poll mode, IRQ_n has no influence in the behavior of the processor.

The waveform of proc1 is shown in figure 6.

1. Driven by $IRQ\_n = 0$ without waiting for $wait\_end$. Ready to manipulate BR.
2. Select BR.
3. Write the letter 'a' to BR.
4. Write SR to have POC handle the current BR content.
5. Once the process is completed, return to idle.
6. It can be seen that even if $wait\_end = 1$, it will not activate the processor. In this mode, the processor is only driven by IRQ_n.
7. Start a new round of transmission.

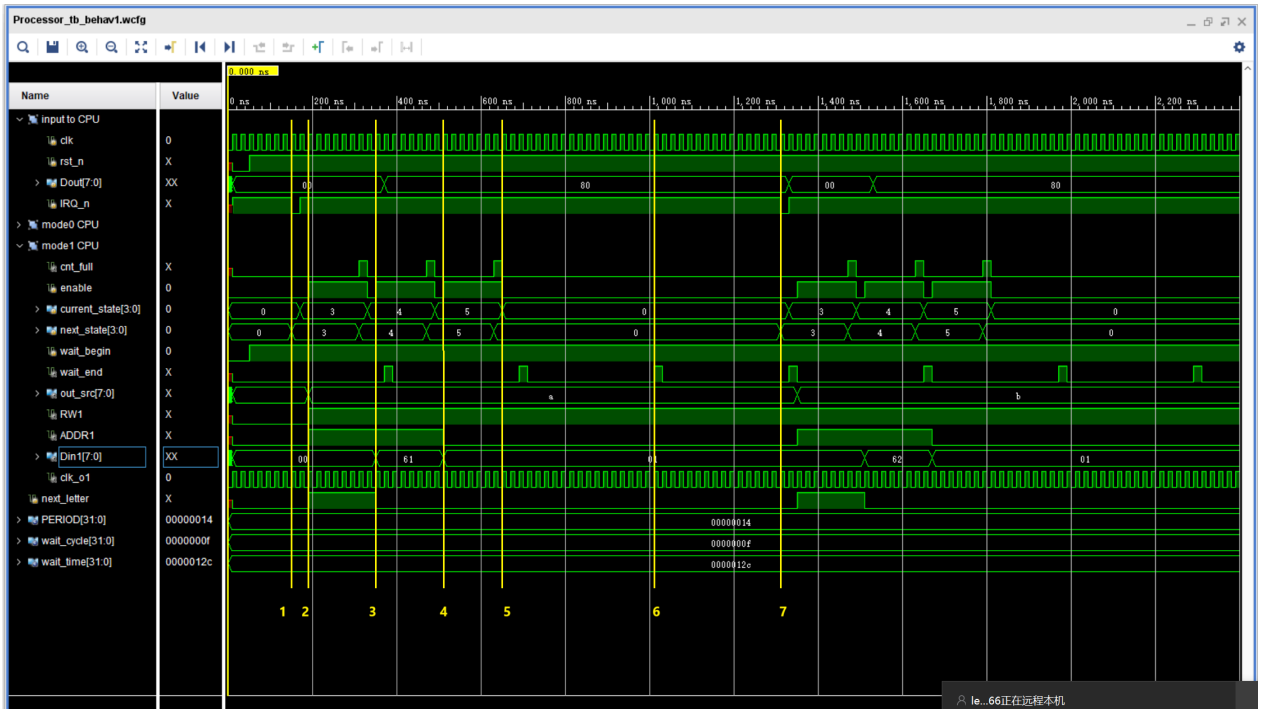

Figure 6: The waveform of the processor working in interrupt mode.

4.1.2 POC

The waveform of POC test bench is given in figure 7.

1. The processor writes the content into BR.
2. Write SR7=0 and wait for POC's response. However, since the printer have not given RDY signal to POC, the POC has no action.
3. Set the processor to 'read' mode to avoid affecting BR.
4. RDY signal from printer.
5. The POC responds to the RDY signal, passes the BR content to the PD. At the same time, POC raises TR to pass the content to the printer. After the printer responds (RDY=0), TR also responds (TR=0). In interrupt mode, POC will issue IRQ_n=0 to inform the processor, while in poll mode it will not.
6. Check if the reset function is effective.
7. Perform the second transmission.

4.1.3 Printer

The waveform of printer test bench is given in figure 8.

1. The reset is over, the printer returns to idle state, giving out RDY signal.
2. The POC detects RDY, giving out TR =1.
3. The printer detects TR =1,giving out RDY = 0,indicating that printer is receiving the content provided by POC.
4. The POC detects RDY is down, so set TR =0. Note that the buffer of the printer is safely updated, and the print flag is raised, indicating that the print operation begins.
5. The print operation is done.

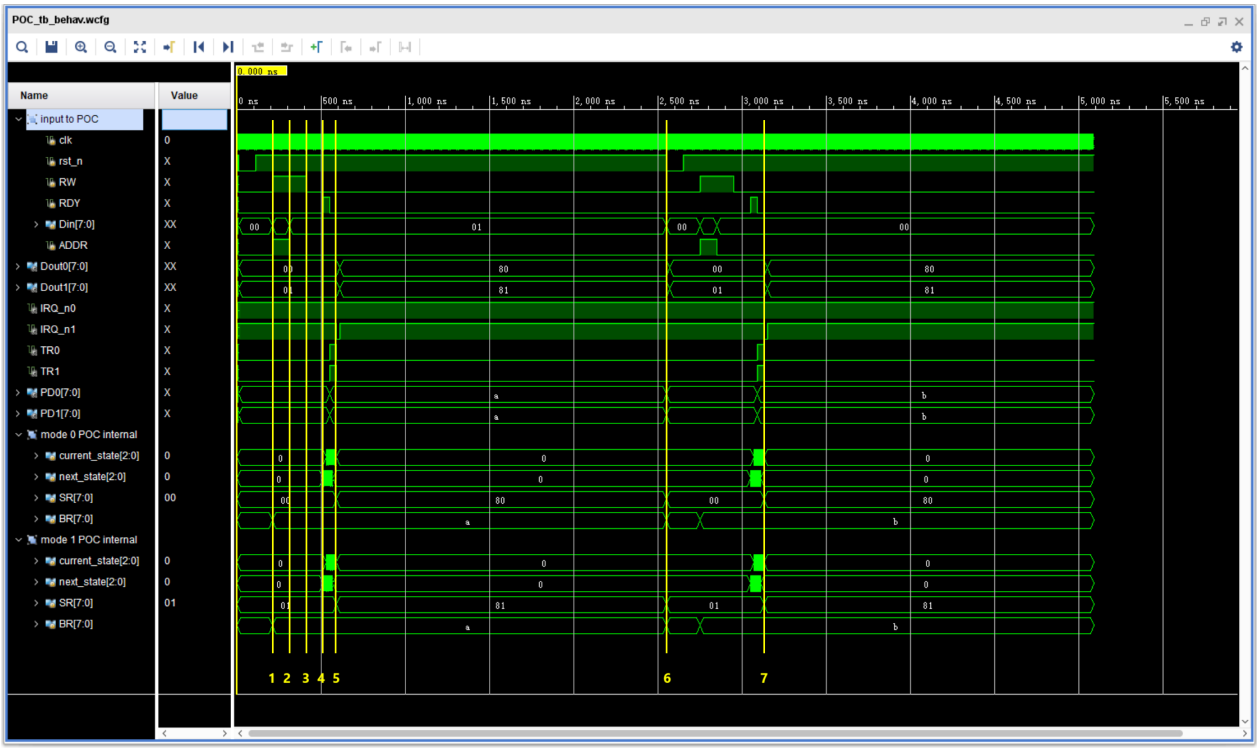

Figure 7: The waveform of the POC test bench.

6. Raise the RDY signal.
7. POC detects RDY, the second round of transmission begins, similar to the first round.
8. The second round of printing ends.
9. Test the reset function.
10. The third round of transmission and printing (from mark 10 to mark 12).

4.2 Top Module Simulation

Similar to the simulation of processors, the top module simulation can be divided into two parts: the poll mode and the interrupt mode. Before presenting the waveform, it's necessary to introduce the basic logic, process, and parameter setup of the test bench. Suppose that under poll mode the processor wants to print characters at a speed of one byte every 20 cycles. But the printer can only print at a speed of 35 cycles per byte. Note that the real-world situation is much worse since the operation speed of processors is way faster than printers. The parameter design here is only for the convenience of inspecting waveform diagrams.

The general simulation process is as follows. SR7=1 for POC indicates POC is ready. The processor is waiting at the time, so the transfer does not start immediately. After processor wait end=1, it selects SR immediately. If the RW and ADDR of the processor are set to 00, it indicates that SR is selected. The Dout of the processor appears with data transmitted from the SR of the POC. From Dout the processor knows that POC is idle, so it selects BR and continues for a period of time, while *out_src* also generates the content to be printed and maintains it. The processor writes data to BR, and at this time, it should check whether the BR register of POC is updated. The processor selects SR and maintains it. The processor writes a status word to the SR so that the highest bit is 0, and hands it over to the POC for processing, which basically end the interactive procedure between the processor and the POC.

Assuming that the POC receives data and wants to pass it to the printer, but the printer has not completed the previous task, the POC just waits. The printer ends the previous task and give RDT =1. POC's PD data appears and remains for a period of time. After that, POC's TR=1 and remains constant. The printer responds with RDY=0 and keeps it constant. The POC holds TR down and TR remains low as a response

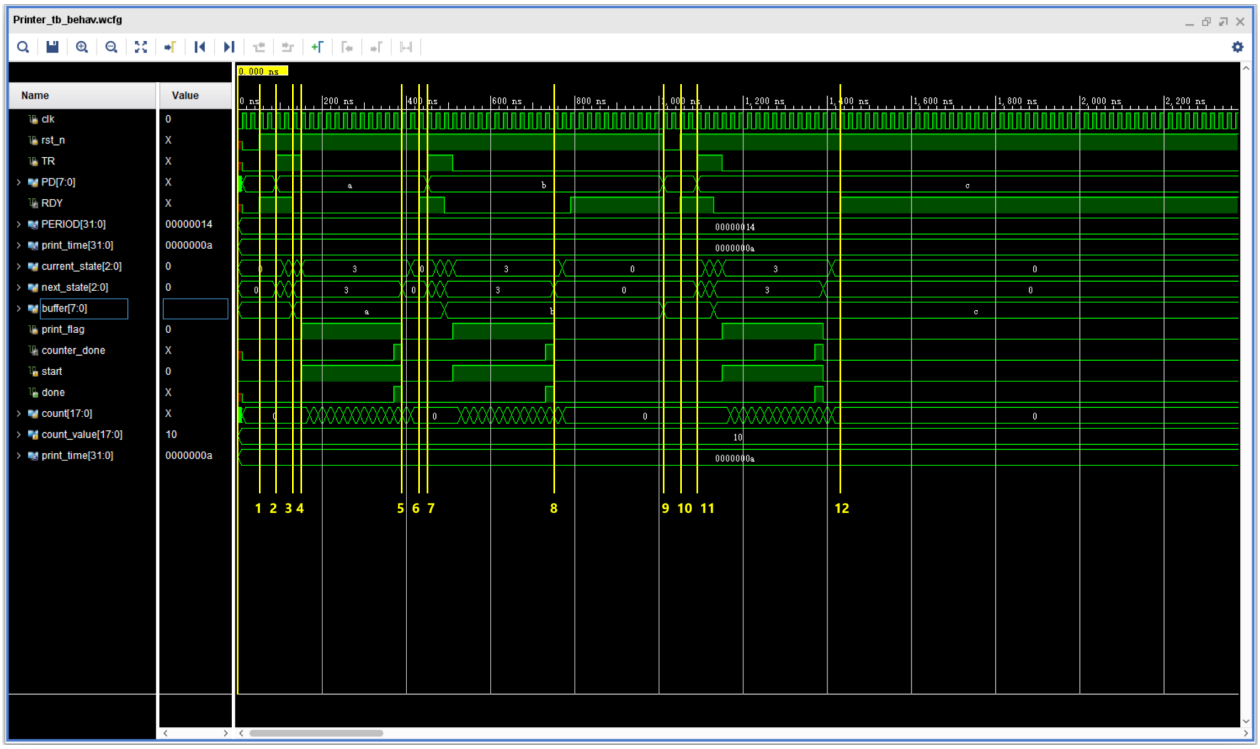

Figure 8: The waveform of the printer test bench.

to RDY=0. Meanwhile, POC sets SR7 to 1 to indicate availability for the processor. After delay indicated for printing, the printer will provide the RDY again.

Next, we take a closer look at the simulation under interrupt mode. The SR7=1 of POC indicates that POC is ready, and a signal of $IRQ_n = 0$ is immediately sent to the processor. The processor responds immediately upon receiving an interrupt request signal, performing the following routine:

1. Selects BR directly and keeps it.
2. Writes BR and keeps it.
3. Selects SR and keeps it.
4. Write the SR status word to set the highest bit to 0.

The handshake process between POC and printer is the same as above and is omitted for brevity.

Note: The readiness of POC and printer are not directly related since they are asynchronous, which also shows the significance of POC's existence.

Now we can inspect the waveform more clearly.

The explanation of figure 9 and figure 10 are given as follows.

Figure 9 mainly illustrates the interactive operation between the processor and the POC.

1. At a certain moment, the processor waits for the end and begins a new round of attempted transmission. Since SR7=1, it indicates that POC can receive content from processor.
2. The processor generates the content that needs to be transmitted.
3. The content is written into BR.
4. Set SR7 to 1.
5. The wait cycle of the processor has ended, but POC did not process the previous word properly, so the processor cannot transfer.

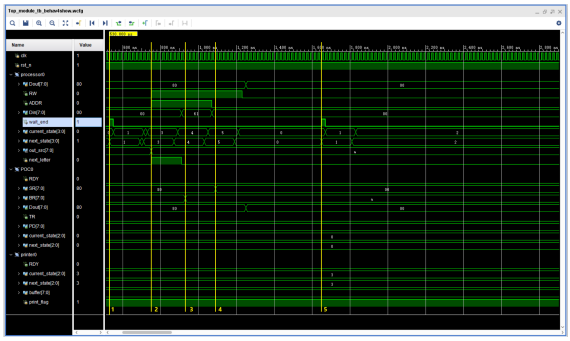

Figure 9: Top module waveform figure 1 under poll mode.

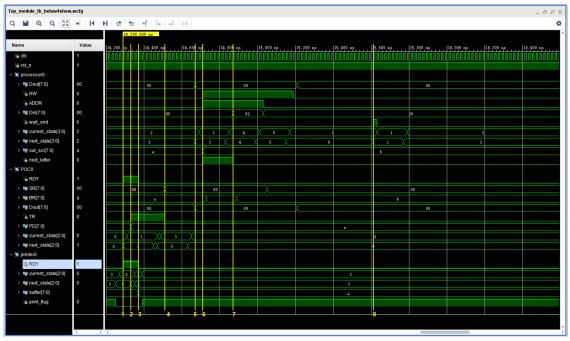

Figure 10: Top module waveform figure 2 under poll mode.

Figure 10 mainly exhibits the actions between the POC and the printer, and the behavior of processor after a successful transmission.

1. The printer has finished printing the previous byte and sent a RDY signal to the POC.
2. POC raise the TR signal, indicating a transfer attempt.
3. The printer set RDY=0 and the buffer received content from PD.
4. POC responds to RDY=0 signal, sets TR to zero, and sets SR7 to 1, indicating that the processor can pass the next byte to it.
5. POC sends SR7 content to Dout and processor receives it.
6. The processor starts a new round of content transfer.
7. The second content 'b' to be transmitted is generated and written to BR.
8. The processor has finished waiting and wants to write new words again, but the printer has not processed the printing of 'a' yet, so it did not provide POC or RDY signals. The BR of POC contains 'b' that has not been handed over to the printer for printing, so it does not accept processor transmission. Only when the printer finishes printing the previous byte, a new cycle can begin.

An overview of the whole process under poll mode is given in figure 11. It can be seen that the printer prints a-z in a loop. From mark 1 to 2, it can be seen that the printed content is always delayed by a certain period of time compared to what the processor wants to print, which is caused by the slow processing speed of the printer.

Next, we present the waveform of the top module under interrupt mode. Still, we present the figures ahead of corresponding explanations.

Figure 12 contains the following actions.

1. The printer prints the previous content and gives RDY=1.
2. POC gives TR=1, requesting transmission.
3. The printer responds by setting RDY to zero, while the buffer is also updated with PD content.
4. Printer starts printing the contents of the buffer.
5. TR is set to zero, and POC sends an interrupt request to the processor.
6. The processor responds with an interrupt and starts a new round of transmitting to the POC.
7. The processor generates new content.
8. The processor writes BR into POC.

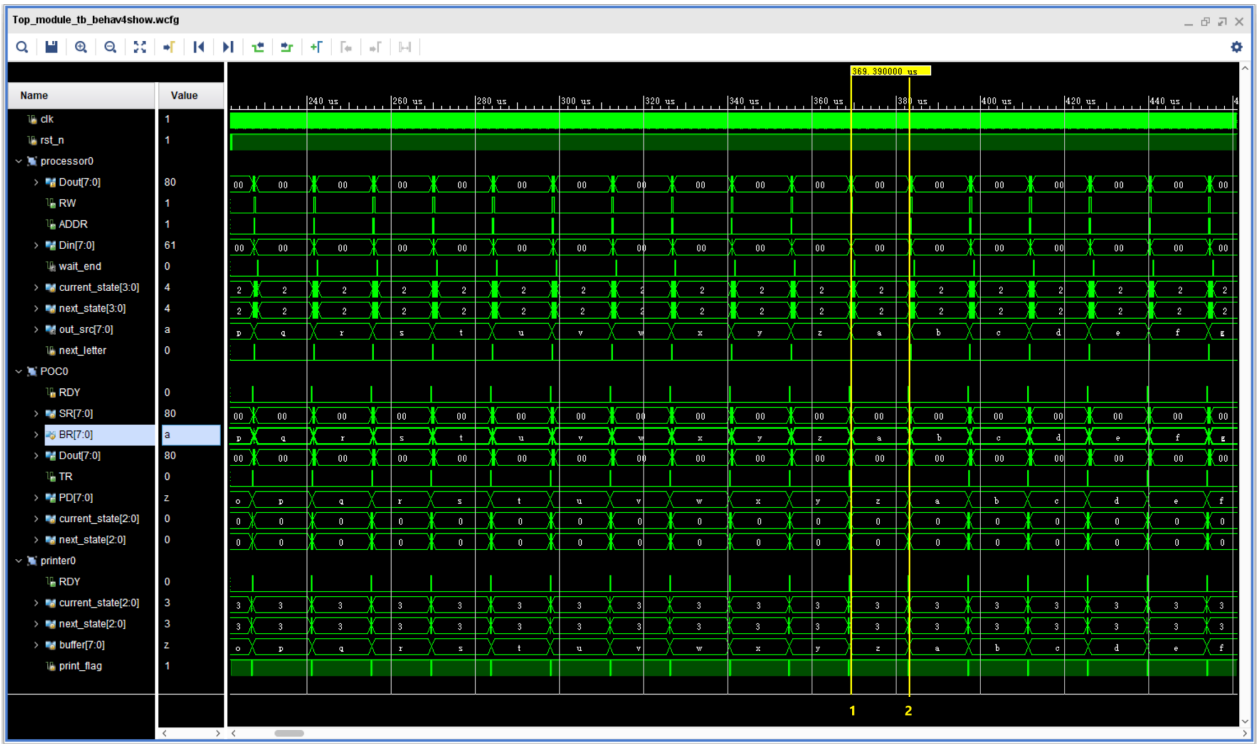

Figure 11: An overview of the whole process under poll mode.

9. The processor selects POC and sets SR7 to zero. When printer is ready, the new content can be delivered to the printer and the loop starts again.

Figure 13 is the overview of waveforms, it can be seen that the function of periodically printing 'a' to 'z' has been implemented. It's note-worthy that due to the interrupt mode, the efficiency of the processor is achieved.

5 Conclusion

The design and simulation of the parallel output controller (POC) have been successfully completed. The POC serves as an essential interface between the system bus and the printer, facilitating efficient data transfer through handshaking protocols. The simulation results demonstrate the effectiveness of the POC in both polling and interrupt modes.

In polling mode, the processor periodically checks the status of the POC and transfers data accordingly. This mode ensures continuous operation but may lead to inefficiencies due to the processor's idle time while waiting for the POC to become ready. The simulation showed that the processor's periodic attempts to transfer data were synchronized with the POC's readiness, ensuring smooth data flow. However, the delay between the processor's data generation and the printer's actual printing was noticeable, primarily due to the printer's slower processing speed.

In interrupt mode, the POC sends an interrupt request to the processor when it is ready to receive data. This mode significantly improves the processor's efficiency by allowing it to continue executing other tasks until an interrupt is received. The simulation results indicate that the interrupt-driven mechanism effectively reduces the processor's idle time and enhances overall system performance. The processor's immediate response to the interrupt request ensures that data transfer occurs promptly when the POC is ready.

Overall, the POC design meets the requirements of interfacing between the processor and the printer. The handshaking protocol ensures reliable data transfer, and the dual-mode operation (polling and interrupt) provides flexibility to optimize system performance based on specific use cases. The simulation results validate the functionality and robustness of the design, demonstrating its potential for practical implementation in computer systems.

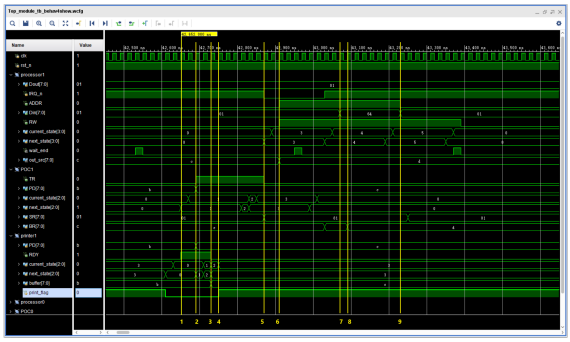

Figure 12: Top module waveform under interrupt mode.

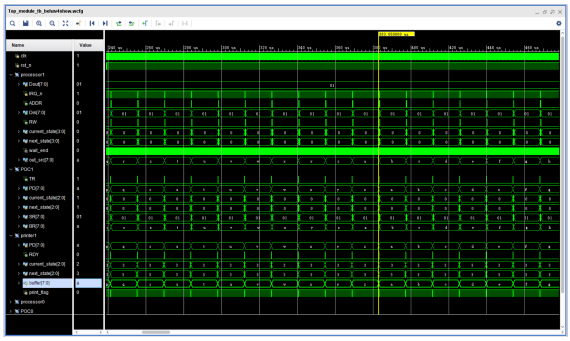

Figure 13: Top module waveform overview under interrupt mode.

6 Further Discussion

6.1 Potential Improvements

While the current design of the POC functions effectively, several areas can be explored for further improvement:

1. Enhanced Buffering Mechanism: The current design assumes a single-character buffer in the printer. Implementing a multi-character buffer in the POC could improve data transfer efficiency by allowing the processor to send multiple characters before waiting for the printer's readiness signal.

2. Error Handling and Recovery: The current design focuses on normal operation without considering potential errors such as data corruption or printer malfunctions. Adding error detection and recovery mechanisms, such as checksums or retries, would enhance the reliability of the system.

3. Dynamic Mode Switching: The processor currently operates in a fixed mode (polling or interrupt) during a task period. Implementing dynamic mode switching based on system load and performance requirements could further optimize resource utilization.

6.2 Scalability and Integration

The POC design can be extended to support additional output devices beyond printers. By generalizing the interface and handshaking protocol, the POC could serve as a universal parallel output controller for various peripherals. This scalability would require modifications to the status and control registers to accommodate different device characteristics and protocols.

6.3 Real-World Applications

The principles and mechanisms demonstrated in this POC design are applicable to various real-world scenarios, such as industrial automation, embedded systems, and data communication networks. The handshaking protocol and dual-mode operation provide a robust framework for interfacing between high-speed processors and slower peripheral devices, ensuring efficient and reliable data transfer.

6.4 Future Work

Future work could involve implementing the POC design on a hardware platform using FPGA technology. This would allow for real-world testing and validation of the design's performance and reliability. Additionally, exploring advanced synchronization techniques and optimizing the handshaking protocol could further enhance the POC's efficiency and adaptability to different system configurations.

In conclusion, the POC design presented in this report offers a reliable and efficient solution for interfacing between processors and output devices. The simulation results validate its functionality, and the potential for further improvements and real-world applications highlights its significance in the field of computer system design.